Software Architecture Specification (SAS)

Revision – 1.0.1
Last Print Date: 8/1/2002 - 1:48 PM

Linux System Software for the InfiniBand* Architecture

*Other names and brands may be claimed as the property of others.

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

This Specification as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Abstract

InfiniBand is an industry standard that defines a new input-output subsystem designed to connect processor nodes and I/O nodes to form a system area network. The specification primarily defines the hardware electrical, mechanical, link-level, and management aspects of an InfiniBand™ fabric, but does not define the lowest layers of the operating system stack needed to communicate over an InfiniBand fabric. The remainder of the operating system stack to support storage, networking, IPC, and systems management is left to the operating system vendor for definition.

This specification details a software architecture for the operating system software components needed to support an InfiniBand fabric, specifically for the Linux operating system. The architecture for several of these components is further influenced by other standards and emerging standards that define standards and protocols for components of the operating system. Examples here are emerging protocols like Internet protocol (IP) over InfiniBand (IPoIB) and the SCSI RDMA Protocol (SRP) and proposed definitions for standard InfiniBand transport and driver APIs. Taking into account the InfiniBand specification and these emerging standards, this specification details the overall architecture for the Linux operating system components.

This architecture serves as an overall umbrella for the development of these Linux operating system components by the Linux community. The goal is to have InfiniBand support completed and included in a standard Linux distribution in time for HCA product launch.

Table of Contents

1. Introduction

1.1 Purpose of this Document

This document outlines the structure and major components required to support InfiniBand in the Linux operating system.

The target audience includes software design and sustaining engineers responsible for developing InfiniBand OS and management software. Anyone interested in understanding the high-level software structure of the InfiniBand Linux Software should read this document.

1.2 Document Scope

This SAS presents the structure and behavior of the entire software stack. It only covers the first level of decomposition of the product into its major components. Further decomposition of the major components themselves is included in the High-Level Design, which also includes the APIs.

1.3 Terminology

1.3.1 Acronyms

Acronym	Description
B_Key	Baseboard Management Key
CI	Channel Interface
CQ	Completion Queue
CQE	Completion Queue Element
CRC	Cyclical Redundancy Check
DGID	Destination GID
EEC	End-to-end Context
EQ	Event Queue
GID	Global Identifier
GMP	General Services Management Packet
GRH	Global Route Header
GSI	General Services Interface
DMA	Direct Memory Access
GUID	Globally Unique Identifier
HCA	Host Channel Adapter
IBA	InfiniBand Architecture
IHV	Independent Hardware Vendor
IPv6	Internet Protocol, version 6
IPoIB	IP over InfiniBand
IOC	I/O Controller
L_Key	Local Key
LID	Local Identifier
LMC	Lid Mask Control
LRH	Local Route Header
M_Key	Management Key
MAD	Management Datagram
MSN	Message Sequence Number
MTU	Maximum Transfer Unit
P_Key	Partition Key
PD	Protection Domain

PSN	Packet Sequence Number
Q_Key	Queue Key
QoS	Quality of Service
QP	Queue Pair
R_Key	Remote Key
RC	Reliable Connection
RDMA	Remote Direct Memory Access
SD	Sockets Direct
SGID	Source GID
SLID	Source LID
SL	Service Level
SM	Subnet Manager
SMI	Subnet Management Interface
SRP	SCSI RDMA Protocol
SMP	Subnet Management Packet
TCA	Target Channel Adapter
UC	Unreliable Connection
UD	Unreliable Datagram
VL	Virtual Lane
WC	Work Completion
WQ	Work Queue
WQE	Work Queue Element
WQP	Work Queue Pair
WR	Work Request

Table 1-1: Definition of Acronyms used

For additional terminology, see “Appendix A: Definitions”.

1.4 Related Documents

The following documents are referenced by the Software Architecture Specification:

Document Name	Revision
InfiniBand Architecture Specification	1.0.a
Annex B Sockets Direct Protocol (SDP), Release	1.0.a
IP over IB IETF draft: http://www.ietf.org/ids.by.wg/ipoib.html	
Fast Sockets reference: http://www.cs.purdue.edu/homes/yau/cs690y/fastsocket.ps	
Stream Socket on Shrimp reference: http://www.cs.princeton.edu/shrimp/Papers/canpc97SS.ps	
Memory Mgmt. in User Level Network Interfaces reference: http://ww.cs.berkeley.edu/~mdw/papers	
InfiniBand Configuration Management (CFM) Annex E	
SCSI RDMA Protocol – Working Draft, http://t10.org/drafts.htm	

Table 1-2: Related Documentation

2. Architecture Overview

2.1 Identification

This document describes a software architecture for the Linux operating system and management software components needed to support the InfiniBand Architecture.

It is envisioned that the InfiniBand Linux Software will be developed by the Linux community within the context of a SourceForge project and eventually become part of a standard Linux distribution.

2.2 Goals and Objectives

The goal of the architecture is to describe, in sufficient detail, all of the software components, their responsibilities and behavior, and their interfaces to allow a more detailed design to proceed forward. The architecture will not define the exact APIs. The APIs will be documented in the High Level Design specifications for each software component.

A goal of the architecture is to provide support for multiple vendors HCAs for interoperability.

2.2.1 Compatibility and Interoperability

The architecture employs the use of industry standard interfaces, where available, for communication among the software components. This supports the greatest possible interoperability with third party software. For example, communication between the OS software and the subnet manager will use the standard protocols defined in the InfiniBand Architecture specification. Storage will use the emerging standard SRP protocol and networking will use the emerging standard IPoIB protocol.

2.2.2 Extensibility

The architecture places a requirement on each component requiring the component to provide a mechanism for the user of the interface to determine the revision of an interface.

This allows the user of an interface to determine if it is compatible with a particular implementation and allows the interface to a particular component to be extended over time.

2.2.3 Portability and Platform Independence

The architecture will promote portability to other operating systems wherever possible. However, the architecture is optimized for Linux. Attempts will be made to accommodate the requirements of other more restrictive operating systems, such as EFI, to the extent that it does not result in a sub-optimal implementation for Linux.

The architecture requires the components support multiple vendor's HCAs and requires processor and platform independence.

2.3 The Architectural Model

The architectural model used to describe the architecture utilizes functional block diagrams.

2.3.1 Architectural Concepts and Representations

In the functional block diagrams that detail the architecture of the various software components, lower level software components (closer to the hardware) are represented on the bottom of the diagram. The level of software abstraction of the hardware typically increases as one moves from the bottom to the top of the diagram.

Distinct functional software blocks are typically displayed as separate boxes in the diagrams. This does not imply or require the software that is described by the block to be implemented as a single software module, but rather is used to delineate the areas where major interfaces will be defined between layers.

Interfaces between components are represented by lines/arrows between components or dashed lines between components. Functional blocks layered on top of each other also imply an interface between the adjacent blocks.

2.4 Software Architectural Overview

This section provides a description of the overall software architecture and it's major components. The section is intended to introduce the reader to the major components of the architecture and only describes the components at a very high level. The chapters that follow provide a more detailed architecture, interfaces, and relationships of each component to the other components and the Linux operating system.

2.4.1 Architecture Block Diagram

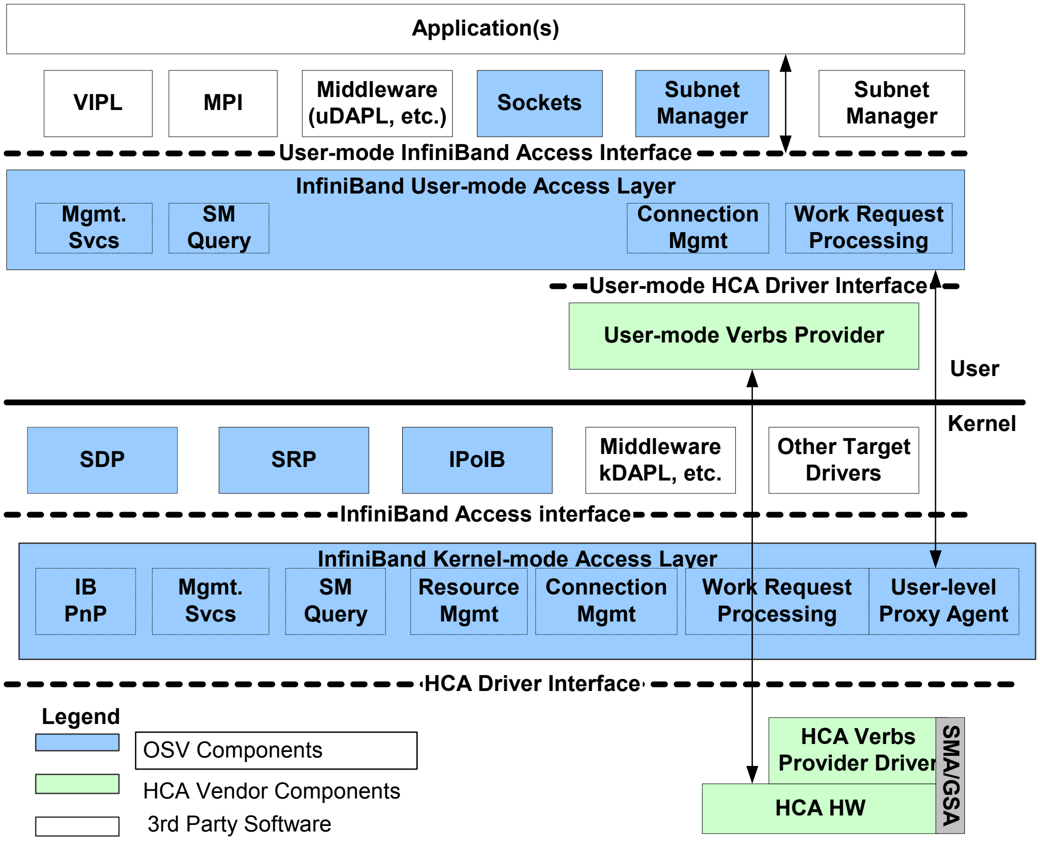


Figure 2-1: Architectural component block diagram

2.4.2 Architecture Block Diagram Description

HCA – Host Channel Adapter provided by a HW vendor.

HW Verbs Provider Driver – This is a vendor-specific piece of software that together with the vendor's HCA forms a unit capable of implementing the verbs as specified in the InfiniBand specification.

HCA Driver Interface – This interface separates vendor-specific HCA code from code that is independent of any particular HCA vendor. The code that sits on top of this interface utilizes the functionality of the verbs without being dependent upon any particular vendor's implementation of the verbs.

InfiniBand Kernel-mode Access Layer– This software exports the full capabilities of the underlying HCA implementations to higher-level software. It exists to provide useful services that would otherwise be duplicated in the independent upper-level components, and to coordinate access to shared InfiniBand resources (such as the SMI and GSI) that are needed by multiple upper-level components. The InfiniBand Access component is envisioned to provide several services:

- **Resource Management** – This tracks HCA resource usage by upper level software components. This is useful for cleaning up those resources when/if the upper level components go away, and for having a means of tracking which upper level components should have access to which HCA resources.
- **Work Request Processing –** Process incoming work requests and dispatch the work request to the appropriate HCA drivers and handles the dispatching of work request completion notification to upper-level components. It also notifies interested upper level components of asynchronous events and errors that are detected by the vendor's HCA during its processing.
- **Connection Management** – Encapsulates the connection management protocol as defined in Chapter 12 of Volume 1 of the InfiniBand specification. This significantly simplifies the process of forming connections for those upper-level components that require them, and coordinates access to the GSI amongst those components.
- **SM Query** – Provides an interface to a standard set of subnet administration queries regarding the subnet configuration.
- **Management Services (SMI/GSI QP Access**)– Coordinates access amongst multiple upper-level components to the management queue pairs provided on each port of the vendor's HCA. This also routes incoming SMPs and GMPs to the appropriate upper-level components.
- **User-level Support Services (Proxy Agent)** – Interfaces to the user-mode InfiniBand Access Layer component to support all the services appropriate for user mode that need a kernel transition. Utilizes resources provided by both the host operating system and by the HW specific HCA verbs provider driver to accomplish this. The user-mode InfiniBand Access Layer communicates with the proxy agent through a set of IO control calls.
- **IB PnP** – The plug-and-play implementation for InfiniBand provided by the host operating system.

InfiniBand Access Interface – This exports the interface that all upper-level components within the kernel use to access the functionality provided by an HCA. The Proxy Agent also uses this interface to support some of the services provided to the user mode access layer.

Subnet Manager – Provides the basic Subnet Manager functionality as defined in Volume 1 of the InfiniBand specification.

SDP – Sockets Direct Protocol driver.

Middleware – Transport and OS independent software layers that support InfiniBand and other RDMA capable transports like iWarp. Also provide an API that is portable across operating systems.

IB Target Drivers – Provides access to TCAs of various types. One example would be a driver provided by the host operating system that utilizes the SCSI RDMA Protocol (SRP) running on top of InfiniBand hardware to access InfiniBand-attached storage. Another example would be a network driver that implements Internet Protocol (IP) over InfiniBand (IPoIB).

User-mode HCA Verbs Provider - Vendor-specific software in user-mode to assist with direct user mode IO (OS bypass) for data transfer operations (DTO).

User-mode HCA Driver Interface - This interface separates user-mode vendor-specific HCA code from code that is independent of any particular HCA vendor.

User-mode Access Layer – This modules exports capabilities of all the underlying HCAs that assist in developing upper layer protocols, like VIPL, uDAPL, SM, HCA Diagnostics etc., providing access to InfiniBand primitives in user-mode. The user-mode InfiniBand Access Layer is a vendor independent shared library that can be dynamically loaded by the higher-level software components.

User-mode InfiniBand Access Interface – This is the interface that all applications running in user-mode use to access the underlying HCA. It attempts to minimize the level of abstraction of the underlying hardware while simultaneously being independent of the implementation of any particular vendor's HCA. It also provides the facilities to allow multiple applications to share access the HCAs.

VIPL, MPI, and other messaging interfaces – user-mode implementations of all of these messaging interfaces can be built using the User-mode InfiniBand Access Interface.

Applications – Applications that wish to live "close to the metal" in order to fully exploit the capabilities of the underlying InfiniBand hardware and are willing to use a lower-level interface for optimum performance can access the User-mode InfiniBand Access Interface directly.

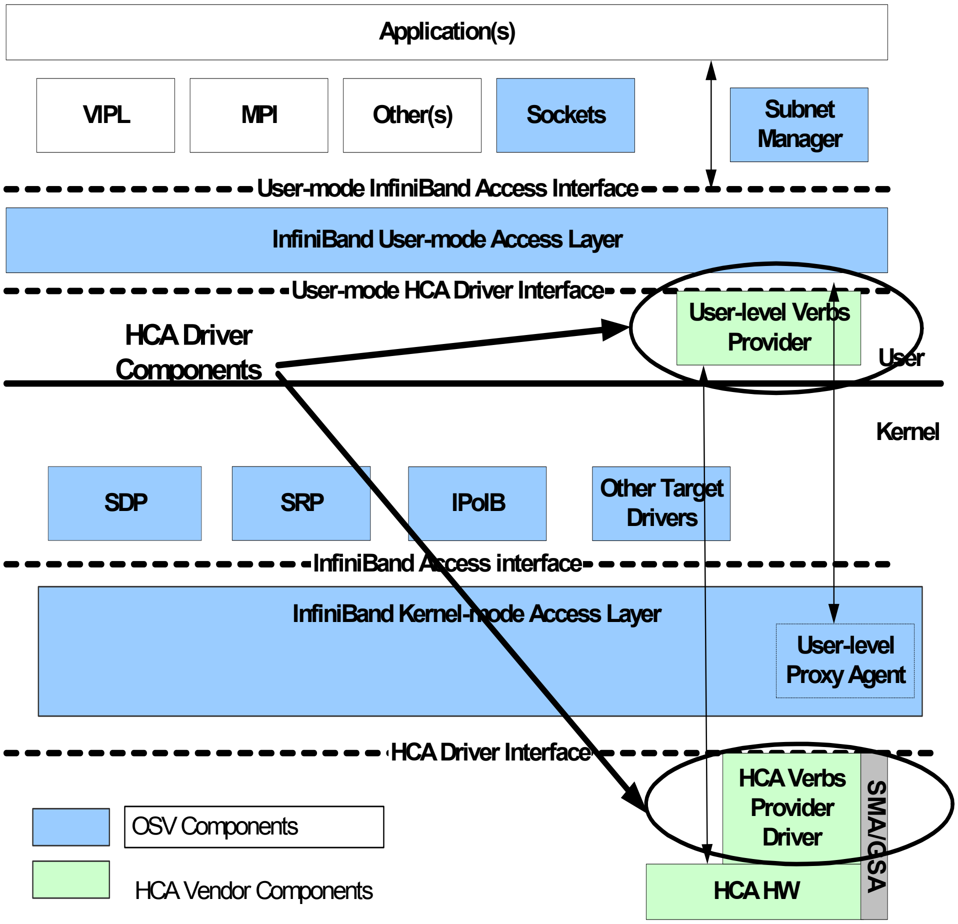
3. Architectural Specification – HCA Verbs Provider Driver

3.1 Introduction

This chapter defines the architecture for the HCA specific, or “verbs provider”, driver software, circled in the figure below. This chapter also contains detailed pictorial representations of the driver software and how it interfaces with other components. The chapter also describes how the driver software interfaces with the hardware, the operating system, and other components of the InfiniBand Linux software stack.

3.2 System Structural Overview

3.2.1 Software Component View



Application(s)
VIPL
MPI
Other(s)
Sockets
Subnet Manager
User-mode InfiniBand Access Interface
InfiniBand User-mode Access Layer
User-mode HCA Driver Interface
HCA Driver Components
User-level Verbs Provider
User
Kernel
SDP
SRP
IPoIB
Other Target Drivers
InfiniBand Access interface
InfiniBand Kernel-mode Access Layer
User-level Proxy Agent
HCA Driver Interface
HCA Verbs Provider Driver
SMA/GSA
HCA HW
OSV Components
HCA Vendor Components

Figure 3-1: Architectural component block diagram

The HCA Verbs Provider Driver is the lowest level of software in the operating system and interfaces directly with the HCA hardware. The driver software is usually developed and supplied by an HCA hardware vendor. There is typically a separate driver for each different HCA type, although a single driver manages multiple instances of the same HCA type.

The HCA driver has two pieces, a loadable kernel mode device driver (HCA Verbs Provider Driver) and a dynamically loadable user mode library (User-mode Verbs Provider), as shown in the figure above. The user mode library is optional

3.2.2 Relationship between InfiniBand Access and HCA Components

The figures below provide more detail about the interfaces between the HCA driver and the other components in the InfiniBand software stack.

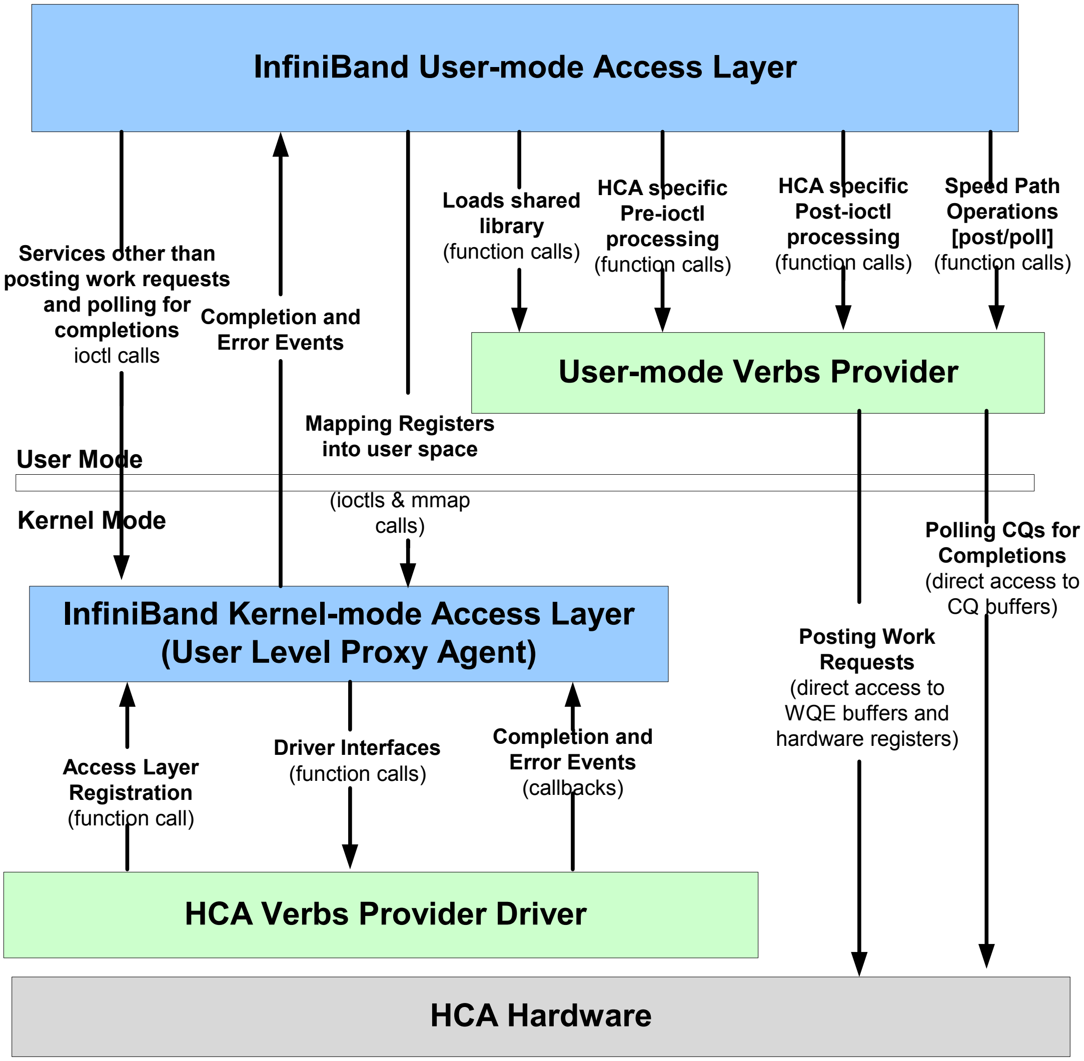
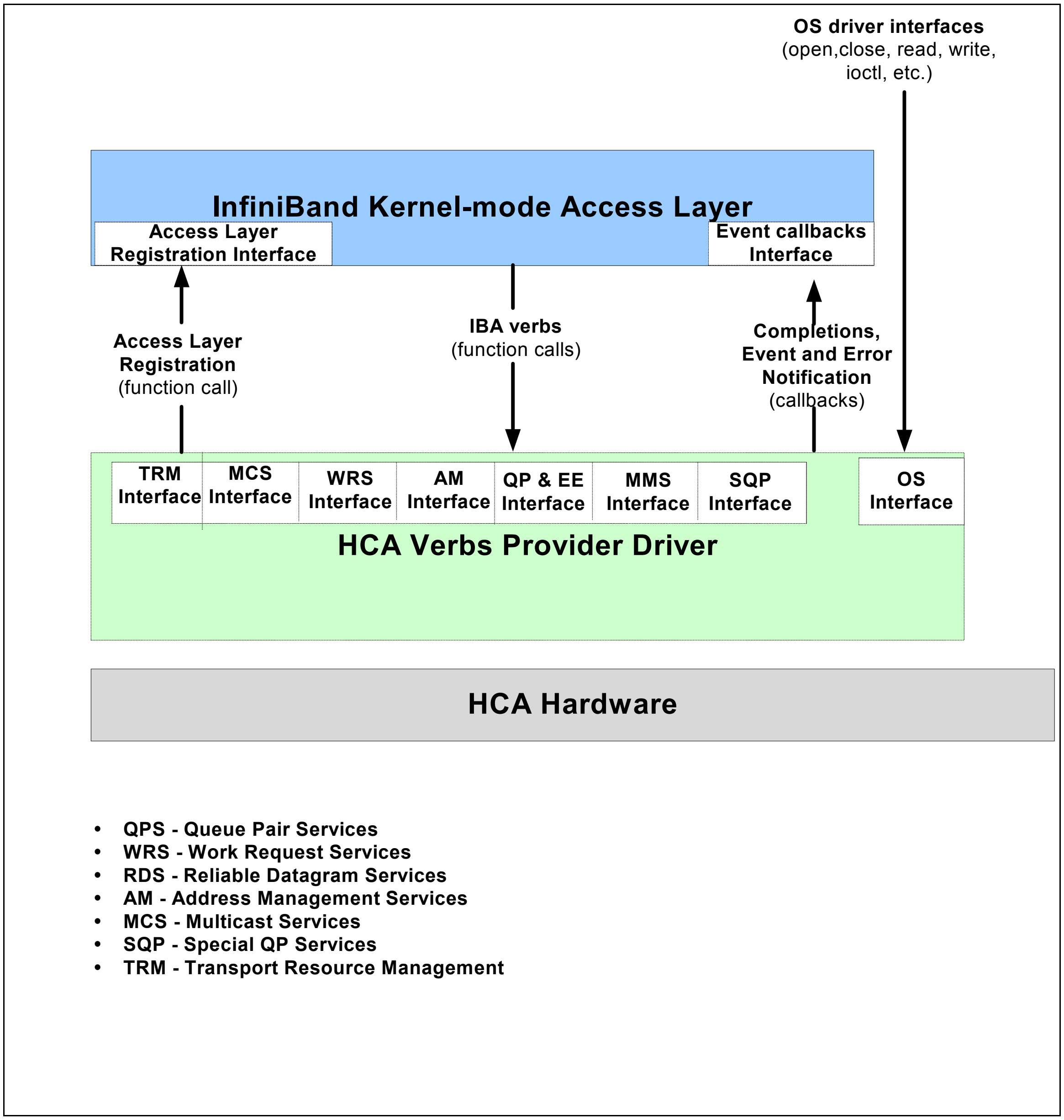

InfiniBand User-mode Access Layer
Services other than posting work requests and polling for completions
ioctl calls
Completion and Error Events
Loads shared library
(function calls)
HCA specific Pre-ioctl processing
(function calls)
HCA specific Post-ioctl processing
(function calls)
Speed Path Operations [post/poll]
(function calls)
User-mode Verbs Provider
Mapping Registers into user space
User Mode
Kernel Mode
(ioctls & mmap calls)
Polling CQs for Completions
(direct access to CQ buffers)
InfiniBand Kernel-mode Access Layer (User Level Proxy Agent)
Posting Work Requests
(direct access to WQE buffers and hardware registers)
Access Layer Registration
(function call)
Driver Interfaces
(function calls)
Completion and Error Events
(callbacks)
HCA Verbs Provider Driver
HCA Hardware

Figure 3-2: Interfaces between InfiniBand Access and HCA Software

The user mode library plugs into the InfiniBand User-mode Access Layer. The User-mode Access Layer discovers it by an entry in a system configuration file that associates a library with the HCA vendor. If a User-mode Verbs Provider is not provided for a HCA type, the User-mode Access Layer will use ioctl calls to the InfiniBand Kernel-mode Access Layer for all functions. However, if a user mode component is provided, it provides access to certain APIs for the major speed path operations. The architecture allows a vendor to provide side band access to aid direct communication with the hardware for speed path. An example would be posting work requests and polling completions.   In this case, the InfiniBand User-mode Access Layer calls the HCA User-mode Verbs Provider directly rather than making an ioctl call to the proxy agent in the kernel for those speed path operations.

If a User-mode Verbs Provider is available for the HCA type, the InfiniBand User-mode Access Layer will call the HCA User-mode Verbs Provider twice for each verbs related function that needs a kernel transition. The first call will be made before the ioctl to the kernel and the second one after the ioctl returns from the kernel. The HCA User-mode Verbs Provider can do any resource initialization and clean up during these calls. For e.g. it can map the H/W doorbell registers into the user process's virtual address space and allocate buffers for WQE construction and completion queues. The InfiniBand User-mode Access Layer will also provide a mechanism during these calls to pass private data between the HCA User-mode Verbs Provider and kernel mode HCA Verbs Provider .   Once the necessary resources are set up and accessible from the user space, the User-mode Verbs Provider then communicates directly with the hardware for posting work requests. HCA hardware then fetches and processes the work request. After the HCA processes the work request it posts the completion in the CQ buffer in the process address space. The notification is then generated by the HCA if the CQ was enabled for notifications by writing an entry to the Event Queue. The InfiniBand User-mode Access Layer then notifies the user process about the completion. The user process can then poll the completion queue to obtain status of the work request submitted.


OS driver interfaces
(open,close, read, write, ioctl, etc.)
InfiniBand Kernel-mode Access Layer
Access Layer Registration Interface
Event callbacks Interface
Access Layer Registration (function call)
IBA verbs (function calls)
Completions, Event and Error Notification (callbacks)
TRM Interface
MCS Interface
WRS Interface
AM Interface
QP & EE Interface
MMS Interface
SQP Interface
OS Interface
HCA Verbs Provider Driver
HCA Hardware

- QPS - Queue Pair Services
- WRS - Work Request Services
- RDS - Reliable Datagram Services
- AM - Address Management Services
- MCS - Multicast Services
- SQP - Special QP Services
- TRM - Transport Resource Management

Figure 3-3: HCA driver software components

The figure above provides more detail on the interfaces provided by the kernel mode HCA Verbs Provider driver. The HCA driver interfaces with the Linux kernel as a regular Linux PCI device driver.

The kernel mode driver interfaces directly with the HCA hardware on its bottom end. It discovers the presence of the HCA hardware as a regular Linux PCI driver. The PCI kernel mode component provides the HCA driver with necessary information about location and size of registers, interrupt vector etc, using the existing mechanisms provided by the Linux OS for any device driver.

The InfiniBand Kernel-mode Access Layer provides an API to allow HCA drivers to register with the Access Layer during HCA driver initialization. The Kernel-mode Access Layer then uses the direct function calls exported by the HCA driver to get access to the Verbs functionality.

Chapters 10 and 11 of the InfiniBand Specifications Volume 1.0a provide a list of the functionality that the driver is supposed to provide. The major functional groups providing access to the hardware are shown below

API Group	**Functionality**
Transport Resource Management	HCA access, Obtain HCA resources such as protection domains, reliable datagram domains
Address Management	Create, modify, query and destroy address handles
Queue Pair and EE Context Management	Create, modify, query and destroy queue pairs and EE contexts.
Special Queue Pair Operations	Allocate Special Queue Pair for QP0, QP1, Raw, Raw Ether queue pair types.
Memory Management	Register virtual, physical address, managing memory windows.
Multicast services	Attaching and detaching a QP from a multicast group
Work Request Processing	Posting work requests to send and receive queues. Polling for completion queue entries, requesting for completion notification.
Event Handling	Ability to notify consumers for completion and asynchronous events.

Table 3–1: Collection of Verbs Groups

3.3 Theory of Operation

3.3.1 Driver Interfaces

This section provides details on the interfaces provided by the and their capabilities. A later section presents a typical usage model of how the Access Layer uses the interfaces. The majority of the interfaces to the driver are derived from the Transport Verbs chapter of the InfiniBand architecture specification Vol1a. It is assumed that the driver will have nearly a 1 to 1 mapping of API calls to the verbs defined by InfiniBand.

3.3.1.1 Driver initialization Interfaces

The HCA Verbs Provider driver provides interfaces that allow the InfiniBand Kernel-mode Access Layer to open and initialize the driver, and determine the capabilities of the HCA.

Refer to the InfiniBand specification for details on the specific capabilities provided by these interfaces.

In Linux, the HCA Verbs Provider driver is a loadable kernel module that typically gets loaded during system start-up. The InfiniBand Kernel-mode Access Layer driver is loaded prior to the HCA Verbs Provider drivers. This is traditionally controlled via module dependencies and load order specified in the */etc/modules.conf* file.

If the HCA presents itself as a PCI device to the hardware, the driver registers with the PCI device driver. It provides the PCI device-id/vendor-id to the PCI driver. When the PCI driver finds an HCA that matches the PCI vendor-id/device-id, it calls the driver back providing pointers to the HCA resources. These are the typical interfaces used in Linux for PCI devices.

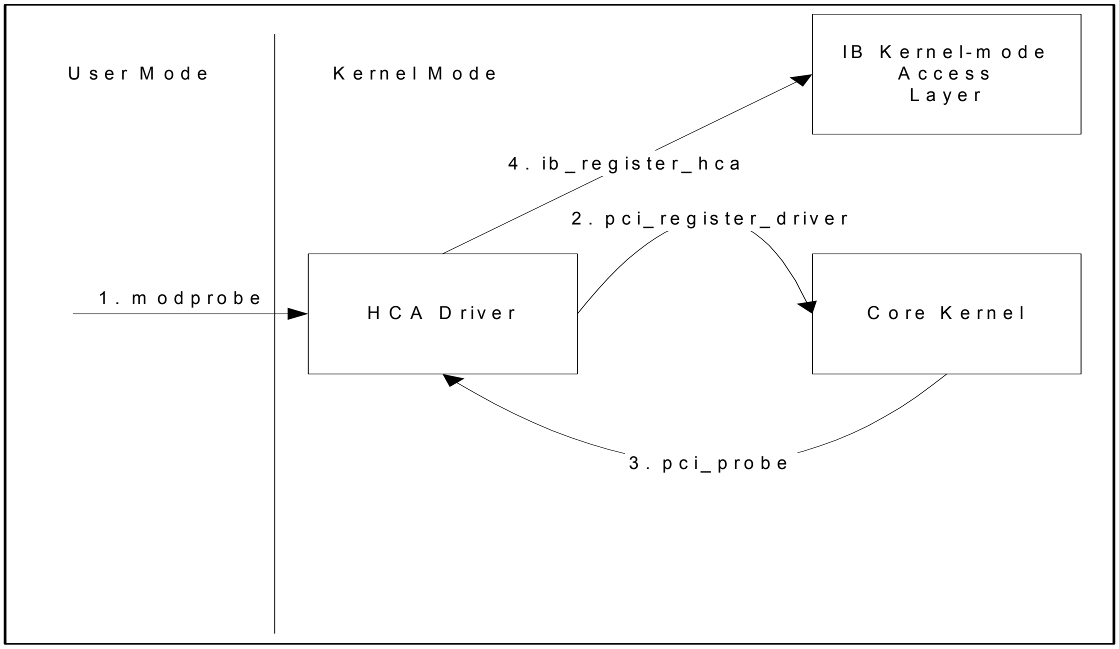


Figure 3-4: Loading the HCA driver

3.3.1.2 Driver close, unload, and cleanup

The HCA Verbs-provider driver notifies the InfiniBand Kernel-mode Access Layer when the driver is shutting down. This is triggered when a HCA hot-remove happens. The HCA Verbs Provider driver then calls the InfiniBand Kernel-mode Access Layer to de-register the HCA leaving the system. The Kernel-mode Access layer must remove all references to the specified HCA before the de-register API call completes.

Dynamically loaded drivers are protected from unloading by maintaining reference counts. The driver *cleanup_module ()* call must never fail. Hence it is important to maintain appropriate reference counts on each *open_hca ()* call from the InfiniBand Kernel-mode Access Layer.

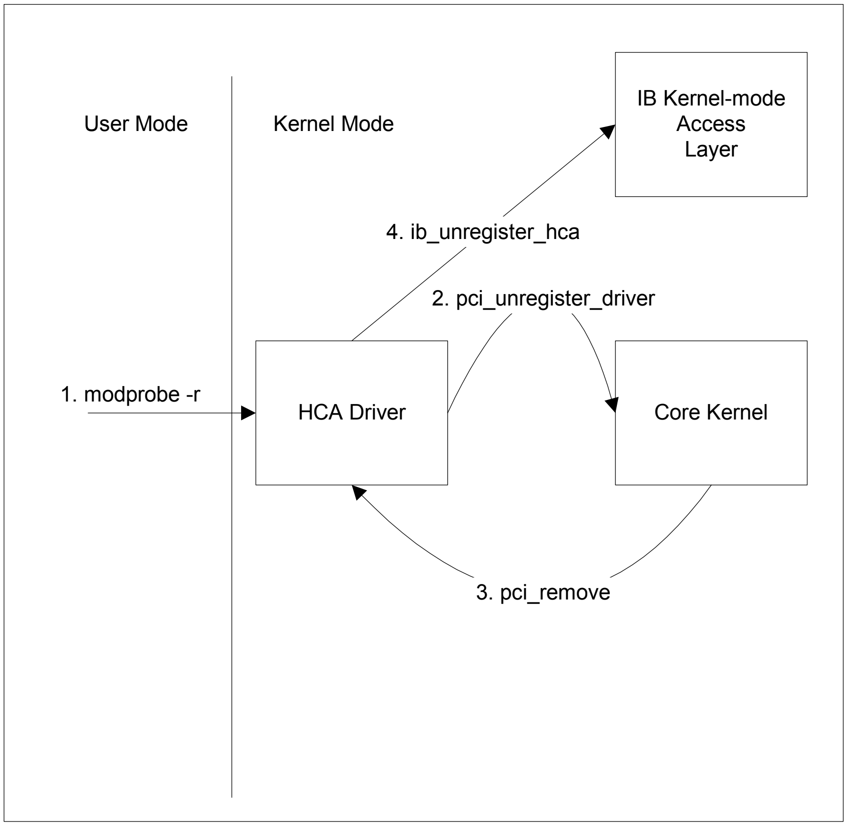


Figure 3-5: HCA Driver Unload Sequence

3.3.2 Typical Driver Usage Model

This section describes a typical usage model of the driver interfaces by the InfiniBand Kernel-mode Access Layer. It does not define every possible usage model, but outlines the way the driver interfaces could be used in a normal operation.

Initialization

The HCA driver is a loadable kernel module that typically gets loaded during system start-up. It is loaded sometime after the InfiniBand Kernel-mode Access Layer.

If the HCA presents itself as a PCI device to the hardware, the driver is instantiated by the Linux PCI driver. Upon start-up, the driver performs any required HCA initialization and sets up and initializes the driver data structures, e.g., things like the event queue, TPT, etc.

Next, the driver registers with the InfiniBand Kernel-mode Access Layer providing the GID of the HCA. The Kernel-mode Access Layer provides an interface to allow the driver to perform this registration.

The Kernel-mode Access Layer can then open and query the driver using the GID to get the resource attributes of the HCA, such as the number of QPs, CQs, TPT, etc. The Kernel-mode Access Layer can then also register a callback for asynchronous event notifications. This allows the Kernel-mode Access Layer the ability to process events that are not associated with a given work request. E.g., port state events.

Queue Pair Setup in preparation for sending data

Before the InfiniBand Access Layer can send data across the fabric, it must allocate protection domains, set up queue pairs and completion queues, and establish connections with remote nodes.

A protection domain must be allocated using the Get Protection Domain interface so that it can be associated with the queue pair when it is created. The Access Layer or upper layers of software can allocate a separate protection domain for each of its clients or client processes to provide memory protection between clients and/or processes. If the connection that the access layer is intending to create is a reliable datagram, the access layer must allocate a reliable datagram protection domain.

After obtaining a PD, the Access Layer creates a completion queue to be associated with the work queue pair. This is done using the Create Completion Queue Interface. Next the Access Layer allocates the queue pair using the Create Queue Pair interface passing the CQ, PD, and other desired attributes of the QP. The connection manager component of the Access Layer can then use the Modify QP interface to modify the state of the queue pair during connection establishment.

Memory Registration in preparation for sending data

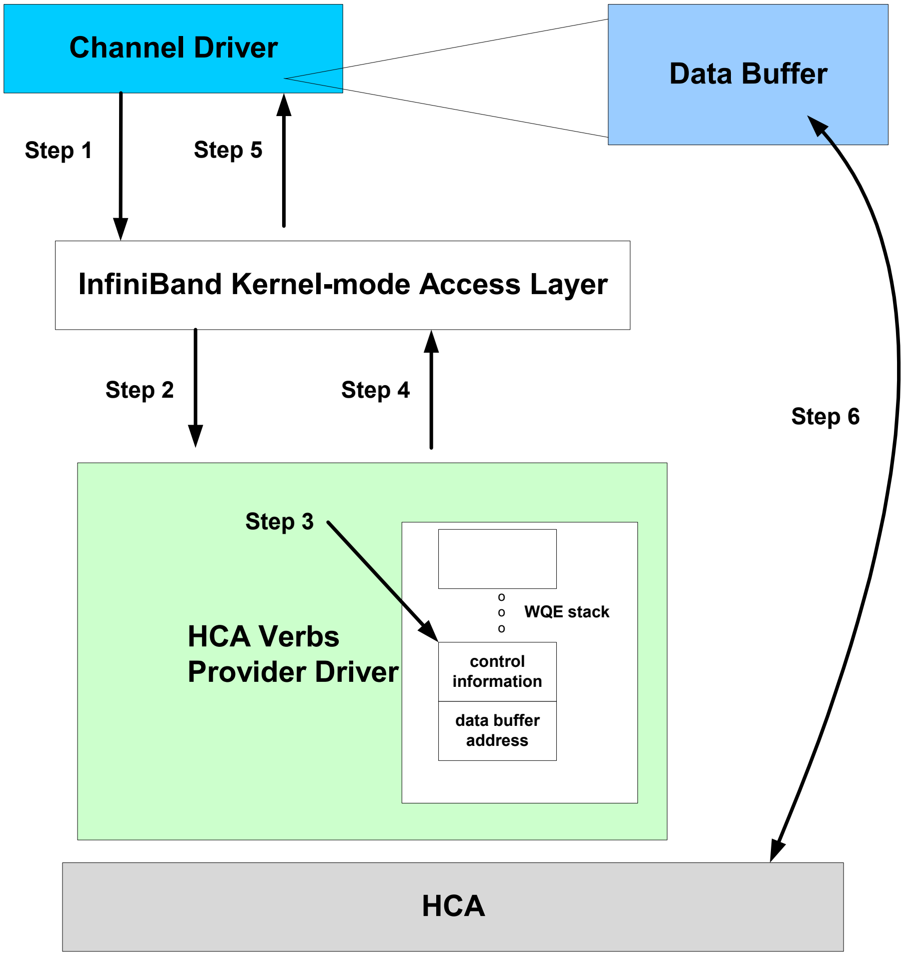
After the queue pairs have been allocated and set up for communication, the Access Layer must register all memory buffers that contain the WQEs or the user data buffers to/from which data will be transferred. To accomplish this, the Access Layer uses the memory registration interfaces of the Verbs Provider driver.

The Register Memory Region interface is used to register virtual addresses and the Register Physical Region interface is used to register physical memory regions. The Access Layer can use the Register Shared Memory Region interface to register memory that is shared between protection domains. It can use the memory window routines to allocate a memory window and then later bind the memory window to a memory region using a work request, but no mater which registration mechanism is used, all memory buffers that are described in work requests must be registered.

Posting Work Requests

Once the Access Layer has allocated and initialized queue pairs and completion queues and registered the memory buffers associated with a data transfer, the Access Layer uses the Post Send and Post Receive interfaces to post work requests.

The diagram below illustrates the typical flow of posting a send or receive work request.

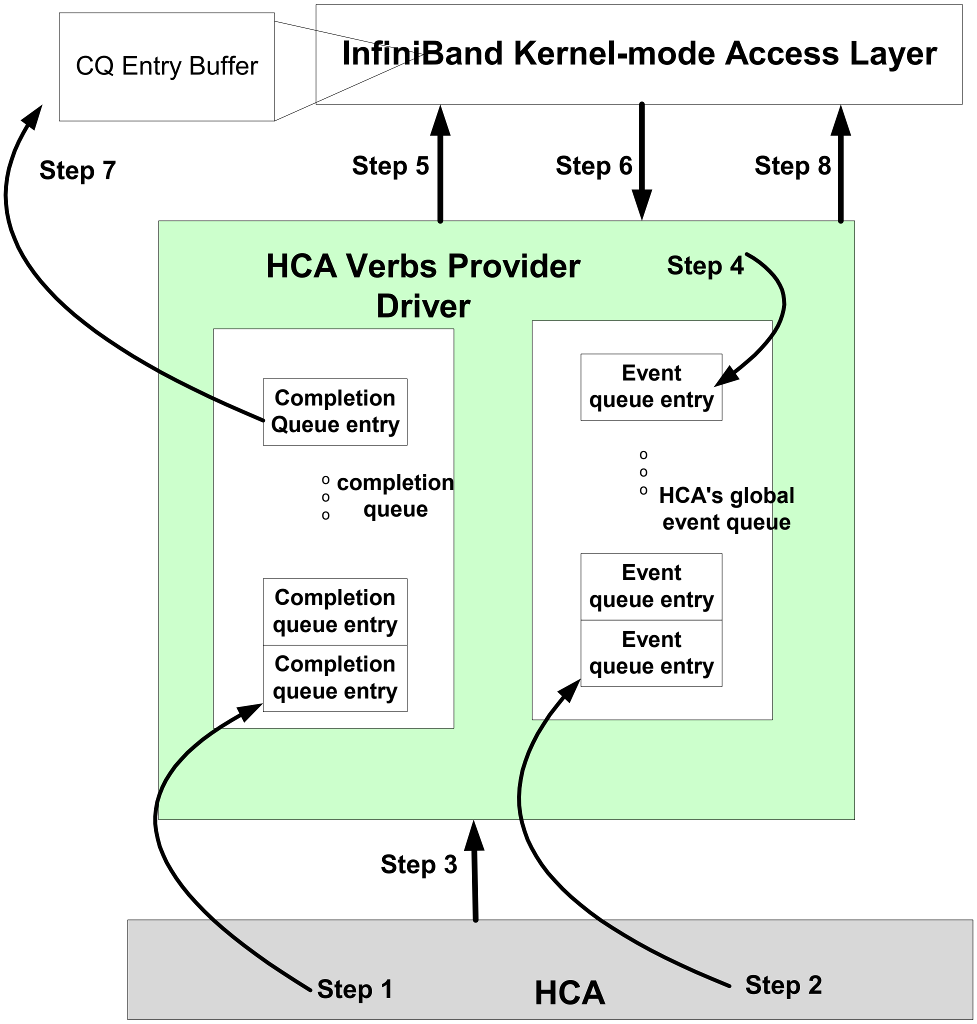


1. Channel Driver posts Work Request to Kernel-mode Access Layer.
2. Kernel-mode Access Layer Posts the Work Request to the appropriate driver.
3. HCA driver builds WQE, posts it to the H/W queue, and rings the doorbell.
4. HCA Verbs-Provider Driver returns to the Kernel-mode Access Layer.
5. Kernel-mode Access Layer returns to the Channel Driver.
6. HCA transfers data to/from the data buffer asynchronously while processing the WQE.

Processing Completions

Once work requests have been posted to the Send or Receive queues and the H/W has completed the work requests, the Access Layer can process the associated completions.

The diagram below shows the typical flow of the completion processing.



1. HCA fills in completion queue entry upon completion of a work request.
2. HCA fills in event queue entry.
3. HCA sends an interrupt to the HCA Verbs Provider driver.
4. The HCA driver polls the event queue to determine the type of event.
5. The HCA driver calls the Kernel-mode Access Layer's completion callback function
6. The Kernel-mode Access Layer Calls the Poll CQ routine.
7. The driver copies the CQ entry to the Access Layer buffer
8. The HCA Verbs Provider driver r returns from the Poll CQ call.

Memory Deregistration

Once the work requests have completed, the Access Layer can use the Deregister memory interfaces to deregister the memory buffers. This frees up the TPT entries for use in subsequent memory registration operations. It is not uncommon for memory buffers to be registered and deregistered for each I/O operation, but it is possible for the Access Layer or upper layers of software to reuse the same I/O buffers and thus only need to register the memory once. This can lead to increased performance in some applications.

Destroying Queue Pairs and Cleaning up after Disconnections

Typically when a connection with a remote node is terminated, the Kernel-mode Access Layer will release the resources associated with that connection for use in subsequent connections. The Access Layer uses the Destroy Queue Pair interface to free the queue pair for subsequent use. It uses the Destroy Completion Queue interface to release the CQ resources. It uses Deallocate Protection Domain and Deallocate Reliable Datagram Domain to free the PD resources that were associated with the QP.

4. Architectural Specification – Access Layer

4.1 Introduction

The access layer provides transport level access to an InfiniBand fabric. It supplies a foundation upon which a channel driver may be built. The access layer exposes the capabilities of the InfiniBand architecture, and adds support for higher-level functionality required by most users of an InfiniBand fabric. Users define the protocols and policies used by the access layer, and the access layer implements them under the direction of a user.

4.2 Design Methodology

The access layer architecture is constructed using bottom-up, object-oriented design techniques with top-down functional requirements. Based on user requirements, specific functionality exposed by the access layer is captured. Commonality between the different functional areas is then identified, and generic components are constructed to support the common areas. Generic components are created using standard library modules where possible. To support user-specific API requirements, detailed modules are layered over the generic components. This is shown in the Figure 4-1 below.

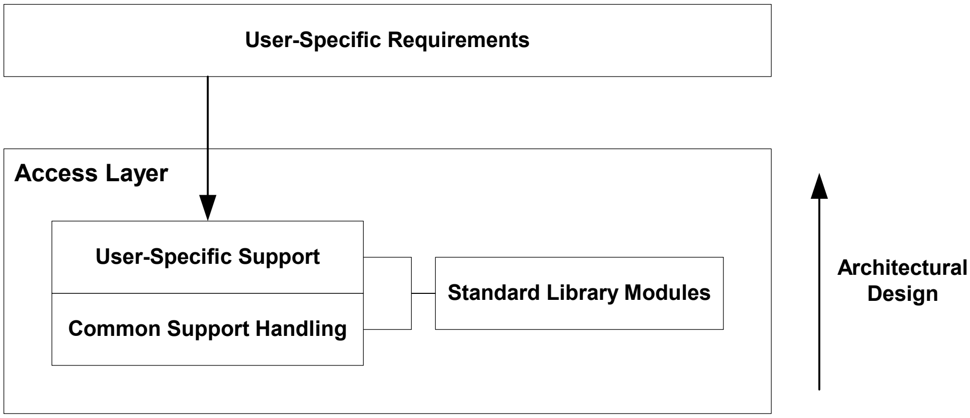


Figure 4-1: Access Layer Architectural Design

4.2.1 Functional Services

The Figure 4-2 highlights the services provided by the access layer.

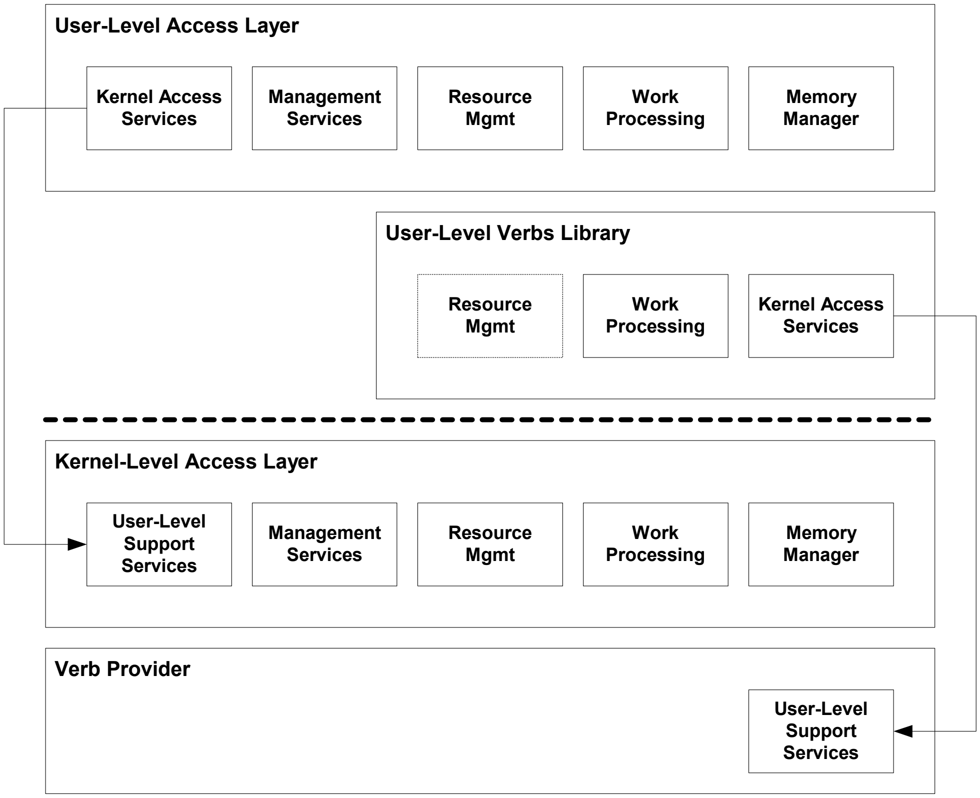


Figure 4-2: Access Layer Functional Services

As seen in the figure, the access layer is located above the verbs provider drivers and operates in both kernel and user-level environments. Kernel access and user-level support services provide proxy agents and implement IO control codes to communicate between the user-level and kernel-level modules. The user-level verbs library is an optional component that provides a speed path for some operations. If a user-level verbs library is not provided, the user-mode access layer will be used for all user-level operations. The main features of the access layer are grouped into the following areas: management services, resource management, work processing, and memory management. These areas are discussed in more detail in the following sections.

4.3 Theory of Operation

4.3.1 Management Services

The access layer provides extensive support for sending, receiving, and processing management datagrams (MADs). All management classes are supported. Figure 4-3 illustrates the MAD architecture exposed by the access layer.

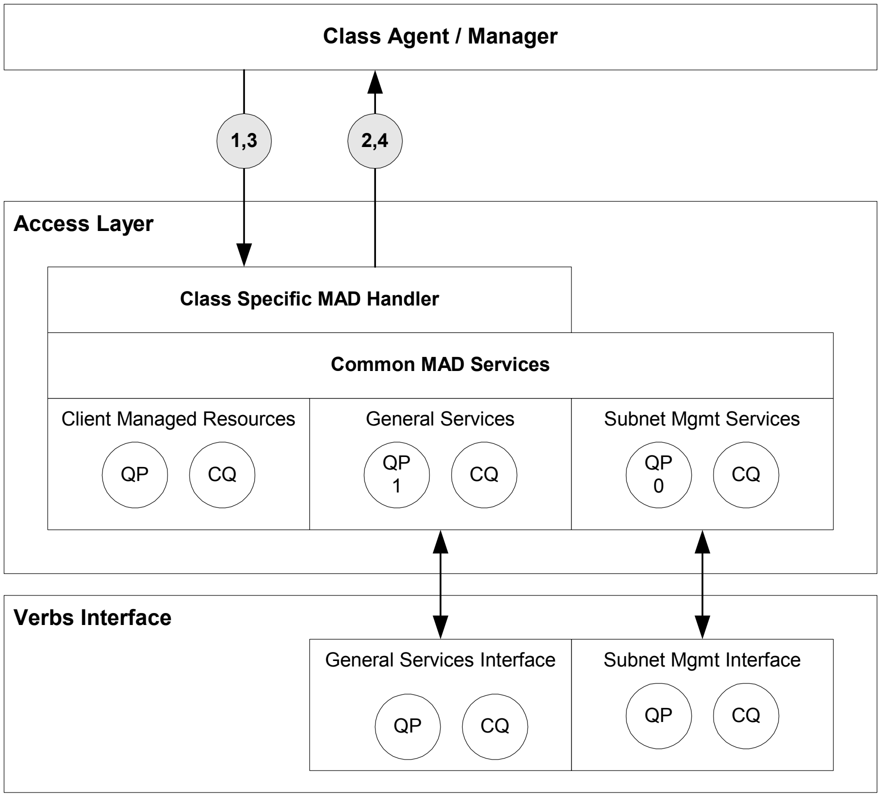


Figure 4-3: Access Layer MAD Architecture

Operation:

1. Clients register as an agent or manager of a specific class. All management classes are supported, but only one class manager may be registered for a specific class.
2. The access layer returns an alias to a QP that may be used to send management datagrams.
3. The class agent or manager posts MADs to the QP alias or, in the case of QP redirection, their own QP.
4. Clients receive incoming MADs through a callback. Unsolicited MADs are routed to the registered class manager. Responses are directed to the agent who initiated the request.

As shown by Figure 4-3, the access layer supports both subnet management (QP0) and general service (QP1) datagrams. Work requests posted to QP0 or QP1 are always processed by the access layer; direct access to these QPs is not allowed. MAD services are divided between common and class specific services. Class specific services deal directly with MADs for a given management class, such as communication management. For convenience, class specific handlers support high-level helper functions that allow users to avoid dealing directly with MADs as shown in Figure 4-4. Class specific helper functions are described in section 4.3.1.1.

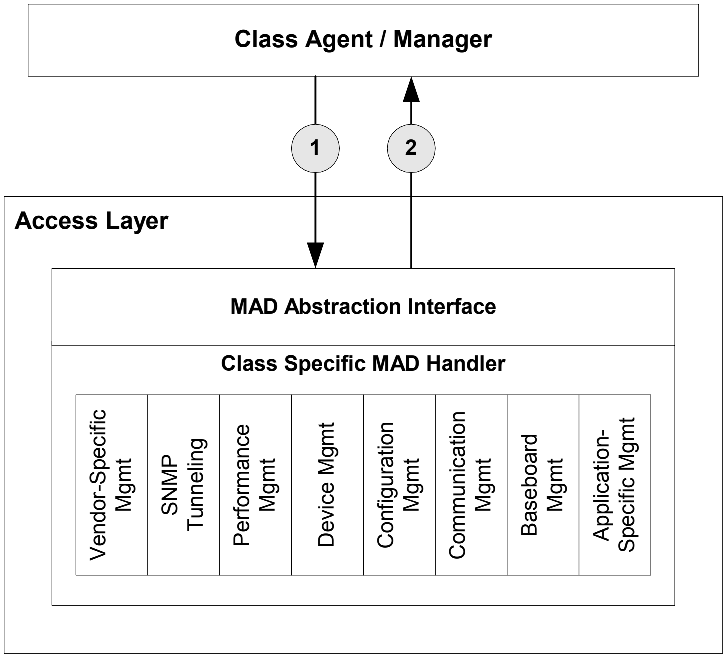


Figure 4-4: MAD Class Specific Handlers

Operation:

1. A user invokes a helper function supported by one of the class specific handlers. MAD helper functions format and send MADs on behalf of the user.
2. The response to a user's request is returned asynchronously to the user.

Regardless of a MAD's class, the access layer provides the following features where appropriate.

- Transaction ID management
- Segmentation and reassembly (SAR) for MAD classes that require SAR
- Support for queue pair redirection
- Queuing to ensure that multiple clients do not overrun an underlying queue pair
- Retransmission of MADs for which a response is expected
- Disabling of MAD support on a per-port basis

4.3.1.1 Management Helper Functions

The access layer provides a set of helper functions that allow users to perform MAD transactions without dealing directly with MADs themselves. These functions include Service Record Registration, Service ID Registration and Resolution, Information Queries, Registration For Local Events, and Subscription For Reports.

4.3.1.1.1 Service Record Registration

Service records allow clients to advertise the availability of basic services to the subnet. The access layer provides support for clients to register and de-register service records with subnet administration. The client provides the service record with the registration request. The access layer returns a handle to the client and performs the MAD transactions with subnet administration. Using the returned handle, the client can unregister the service. The result of the service registration or de-registration is returned to the client through a callback function.

4.3.1.1.2 Service ID Registration and Resolution

Registration of a service ID provides a method for remote users of unreliable datagram services to determine the queue pair number and queue key of a port that supports a given service ID. The access layer provides an interface to allow clients to register and deregister a service ID with the communication manager and to resolve service IDs using the service ID resolution protocol.

Registration:

A client registers a service ID by providing the service ID, the channel adapter and port GUIDs, a callback function, and an optional buffer and length for comparing the private data exchanged during the service ID resolution process. The access layer returns a handle that the client may use to deregister the service.

Resolution:

A client requesting service ID resolution provides the desired service ID, a path record to the remote port, a callback function, and an optional buffer containing private data. A high-level view of the service ID resolution process is shown in Figure 4-5.

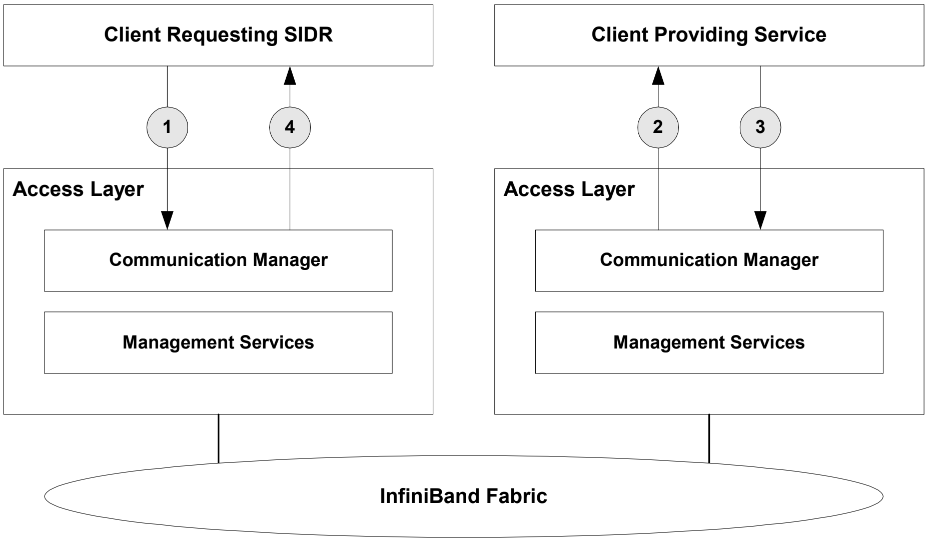


Figure 4-5: Service ID Resolution

Operation:

1. A client makes a service ID resolution request. The communication manager sends a service ID resolution request MAD to a remote node.

2. The communication manager invokes a callback after receiving a service ID resolution request. The client is given information about the request including the private data.
3. The client accepts or rejects the service ID resolution request. A client accepts by providing the QP and QKey. A rejection may optionally indicate redirection to another port.
4. The communication manager returns the result of the service ID resolution process.

4.3.1.1.3 Information Queries

The access layer provides support for client to query for commonly used data. Based on the query type and input data, the access layer performs required the MAD transactions with the appropriate class management entities to obtain the requested data. The result of the query operation is provided to the client through a callback function.

The access layer provides the following queries:

- Query for service records by service name
- Query for service records by well-known service ID
- Query for node record by node GUID
- Query for port record by port GUID
- Query for path records by a port GUID pair
- Query for path records by a GID pair

The access layer supports other queries by sending and receiving MADs through the standard management services.

4.3.1.1.4 Registration For Local Events

Clients of the access layer can register for notification of local events. Local events include the insertion and removal of channel adapters, ports becoming active and inactive, LID changes, partition changes, and IO controllers being assigned or unassigned to a port. Figure 4-6 illustrates the operation of local events.

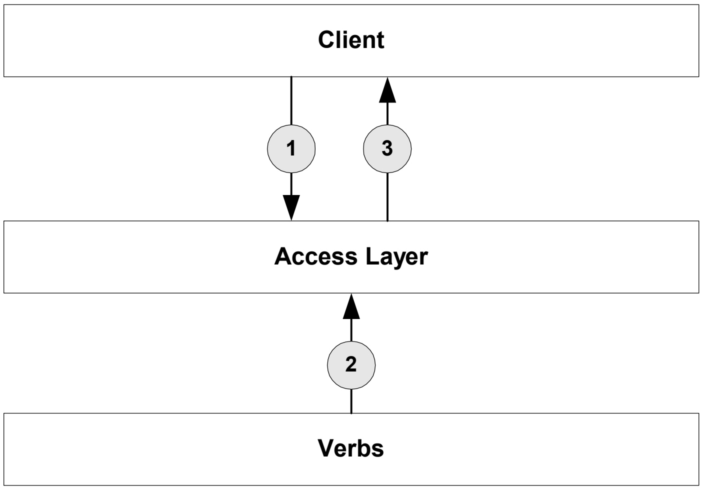


Figure 4-6: Local Event Operation

Operation:

1. The client registers for notification of a local event.
2. The access layer receives notification of a locally occurring event.
3. The access layer notifies clients registered for the local event via a callback.

4.3.1.1.5 Subscription For Reports

Clients of the access layer can register for reports delivered through the event forwarding mechanism.

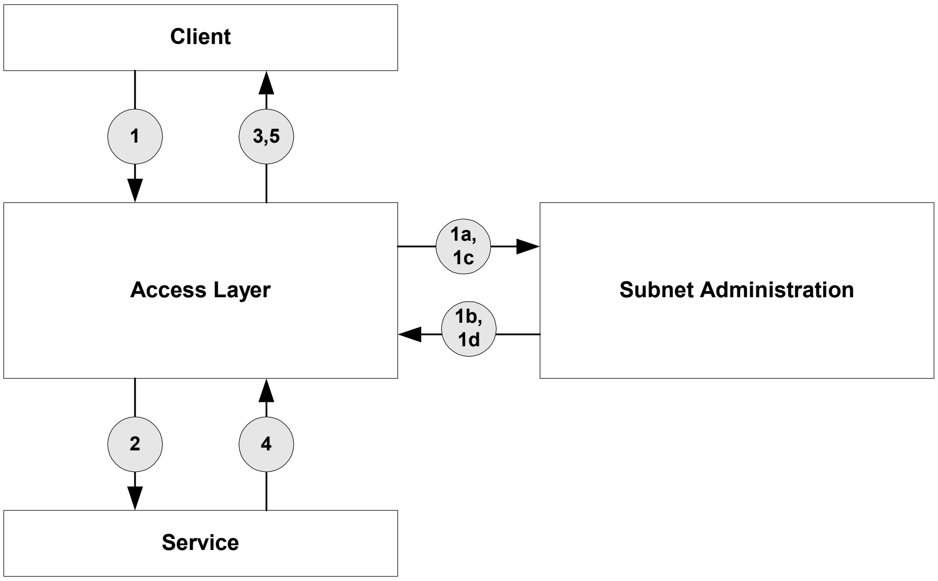


Figure 4-7: Report Subscription

Operation:

1. The client subscribes for a report providing the InformInfo record and the name of the service to subscribe with. The access layer performs the following tasks:
   a. The access layer issues a query to the subnet administration for the named service record. Refer to section 4.3.1.1.3
   b. The access layer receives the requested service record. The service record contains the GID where the specified service may be found.
   c. The access layer issues a query to the subnet administration to obtain path records associated with the specified service using the source and destination GIDs. Refer to section 4.3.1.1.3.
   d. The access layer receives the requested path records.
2. The access layer initiates a MAD transaction to subscribe for the report.
3. The access layer returns the result of the subscription process to the client through a callback.
4. At some point in the future, the access layer receives a report from the service for the subscribed event.
5. The access layer notifies the client of the subscribed event through a report callback function.

4.3.1.1.6 Rejecting An IO Controller Assignment

The InfiniBand Annex E: Configuration Management specification allows a host to reject the assignment of an IO controller. The access layer can perform this operation on behalf of a client. The client requests the IO controller rejection providing the reported IOC information. The access layer performs the MAD transactions with the configuration manager.

4.3.1.2 Communication Manager

In addition to providing support for class managers and agents to send and receive MADs, the access layer provides a class manager for communication. The communication manager abstracts connecting a queue pair with a remote queue pair and transitions the queue pair through the proper states. A high-level view of the communication manager is shown in Figure 4-8.

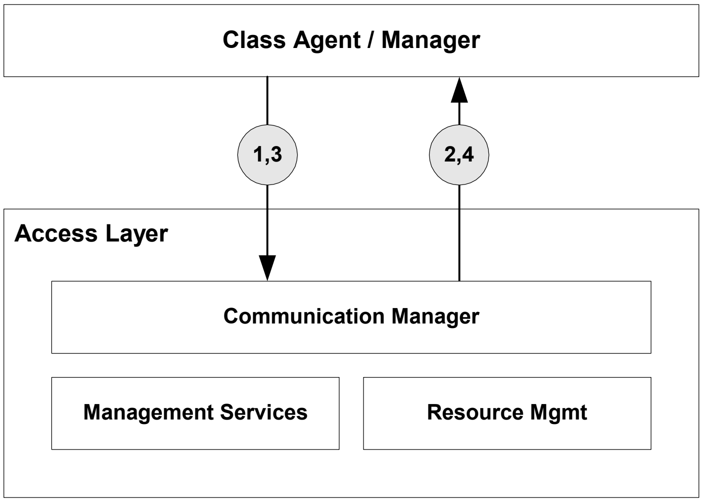


Figure 4-8: Communication Manager

Operation:

1. Clients make a request to connect a QP. For active connection requests, this results in the communication manager sending a connection request MAD to a remote node.
2. The communication manager invokes a callback after receiving a reply to the connection request. The user is given information about the connection attempt, along with the queue pair that will be used.
3. The client accepts or rejects the connection.
4. The communication manager returns the result of the connection process. If the connection was successful, the client may perform data transfers on the given queue pair.

The communication manager performs the following functions.

- Transitions the queue pair state throughout the connection process
- Manages the state of the connection throughout its lifetime
- Supports automatic path migration on the connection

4.3.1.3 Device Management Agent

Some clients of the access layer may require the ability to export an IO controller to the InfiniBand fabric as an IO unit. To support this requirement, the access layer provides a device management agent. The device management agent provides a central point of operation that allows multiple clients to register IO controllers. The device management agent maintains the data necessary to respond to device management MADs related to IO unit operations. A high-level view of the device management agent is shown in Figure 4-9.

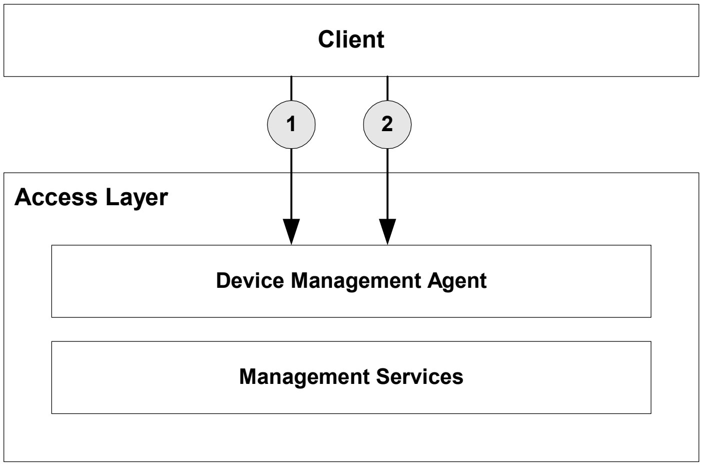


Figure 4-9: Device Management Agent

Operation:

1. The client registers an IO controller profile with the device management agent for a channel adapter. The first IO controller profile registered causes the access layer to set the PortInfo CapabilityMask IsDeviceManagementSupported flag for all ports of the given channel adapter. The access layer returns a handle to the registered IO controller.
1. The client adds service entries to the registered IO controller. The access layer creates a new service entry for the IO controller and returns a service entry handle.

The device management agent performs the following functions.

- Maintains the IO unit information, IO controller profiles, and service entries on behalf of all registered clients
- Automatically assigns IO controllers to slots.
- Indicates the presence of device management support in the PortInfo CapabilityMask flags
- Responds to the device management class MADs related to IO unit operations

4.3.1.4 Plug and Play Manager

The purpose of the plug and play manager is to notify kernel-mode clients of changes in device status. For example, a channel driver may request notification when an IO controller becomes available or unavailable, or perhaps when a channel adapter port becomes active or goes offline.

The plug and play manager reads a configuration file that contains a mapping of notification events to channel drivers. If a channel driver is not present at the time the plug and play manager needs to notify it, the plug and play manager will load the driver. The plug and play manager delivers notifications to channel drivers asynchronously through a callback registered. Figure 4-10 illustrates the channel driver load and notification process.

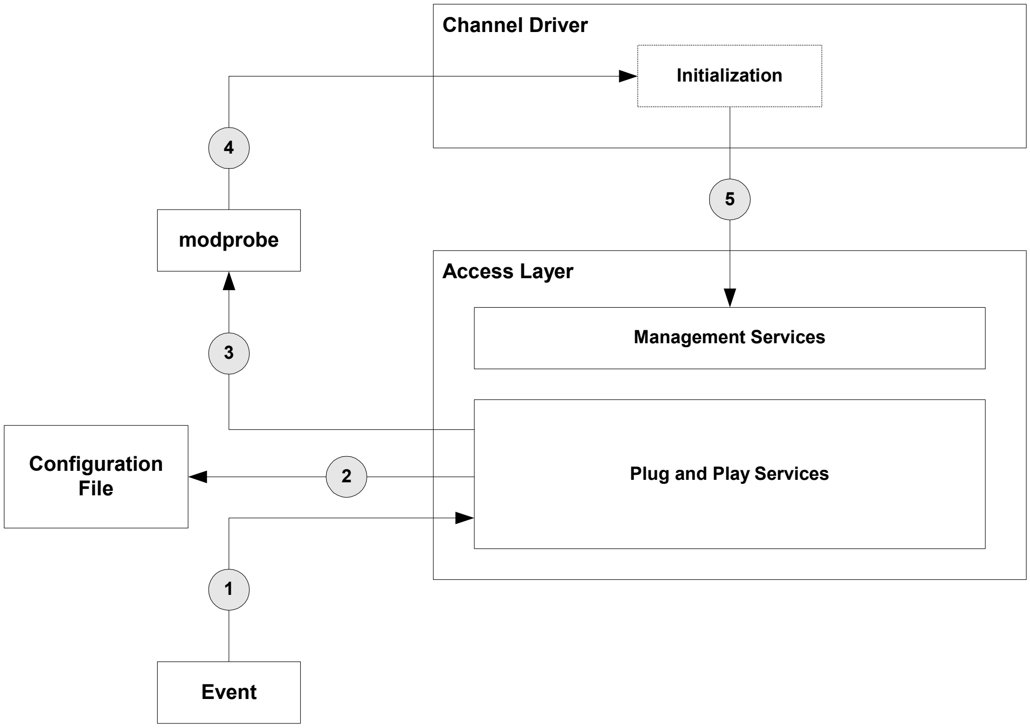


Figure 4-10: Channel Driver Load Sequence

1. The plug and play manager receives notification of an event that has occurred, such as the addition of a new IO controller.
2. The plug and play manager reads a configuration file to determine the correct channel driver to load.
3. The plug and play manager invokes a user-level application to load the appropriate channel driver.
4. The channel driver module is loaded and the driver initialization function is called.
5. The channel driver opens the access layer.

The plug and play manager is a built upon the local event and report subscription mechanisms described in 4.3.1.1.4 and 4.3.1.1.5. The notification callback uses the same parameter format as the local event and report subscription callbacks. Preserving this interface allows the configuration file to be thought of as a “manual” method for a channel driver to subscribe for notifications before the channel driver is actually loaded. The configuration file format supports a subset of the notification capabilities provided by the programmatic interface.

For IO controller channel drivers, the plug and play manager uses the compatibility string matching method described in InfiniBand Annex A: IO Infrastructure specification. This technique allows the configuration file to list a set of compatibility strings for each driver. The matching algorithm uses a search order that selects more specific channel drivers over more generic channel drivers.

4.3.2 Resource Management

The resource management support provided by the access layer is responsible for the allocation and management of exposed channel adapter resources, such as queue pairs, completion queues, address vectors, and so forth. The resource manager is responsible for managing resource domains for resources that are allocated within a hierarchy. For example a CA resource contains CQ and PD resources, and PD's contain QPs. Destruction of a resource domain will automatically result in the release of all associated resources. Figure 4-11 shows the general operation of the resource manager.

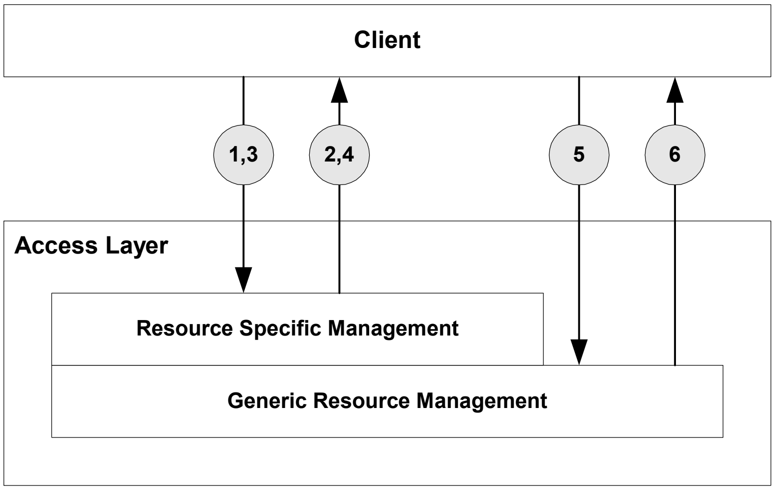


Figure 4-11: Resource Manager

Operation:

1. Clients allocate a new a resource by invoking a resource specific function.
2. The access layer synchronously returns a handle to the allocated resource.
3. Clients can query the access layer for additional properties of the resource. The type of information returned is dependent on the resource being accessed.
4. The access layer synchronously returns the properties of the resource.
5. Once a resource is no longer needed, it may be destroyed through a destruction call.

6. Once the resource has been destroyed, the user is notified. To avoid race conditions, the destruction process operates asynchronously.

Resource management performs the following functions.

- Expose all resources accessible through verbs.
- Manage resource domains to permit destruction of an entire domain. For example, destroying an instance of a CA will automatically release all related resources, such as CQs, QPs, etc.
- Resource management is not considered a critical speed-path operation.

4.3.3 Work Processing

The work processing module handles issuing work requests and completion processing. In general, work processing deals with operations that occur on queue pairs and completion queues.

4.3.3.1 General Work Processing

A general overview of work processing operation is shown in Figure 4-12.

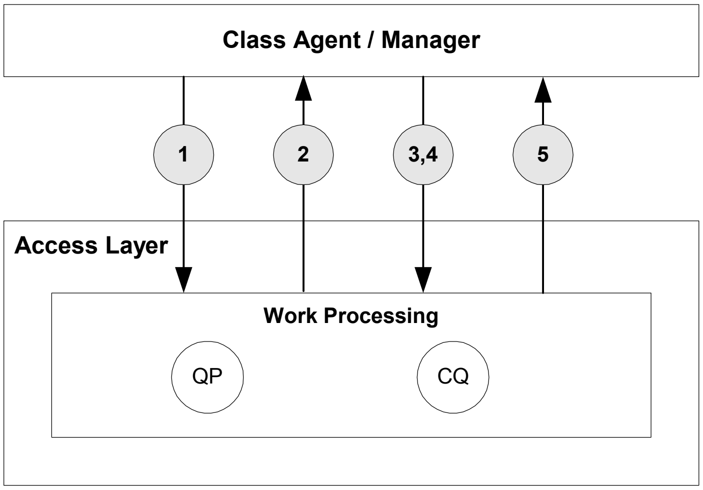


Figure 4-12: Work Processing Overview

Operation:

1. Clients post a work request to a specified queue pair.
2. Upon completion of a work request, the completion queue signals an event to notify the user of the completion. Clients may either wait on a completion event or may be notified through a callback mechanism.
3. Clients re-arm the completion queue to indicate that the completion event should be signaled on the next completed request.
4. Clients poll the completion queue to retrieve information on the completed work request.

5. The completion information is returned to the user synchronously as a result of polling.

Specific requirements of the work-processing module are listed below:

- Completion signaling is controlled by the user
- Users specify whether completion notification is provided via callbacks or event signaling

4.3.3.2 Unreliable Datagram Services

A more detailed operation of a general unreliable datagram QP is outlined below.

Operation:

- The client creates a QP.
- The access layer returns a handle to the newly created QP.
- The client initializes the QP for datagram transfers.
- The client creates an address vector that references the remote destination.
- The access layer returns a handle to the requested address vector.
- The client submits a work request, indicating the QP, QKey, and address vector of the destination.
- The access layer posts the work request for the client.
- The access layer notifies the client of the send completion via the CQ specified when the QP was created.
- The client retrieves the send completion information from the associated CQ.

4.3.3.3 MAD Work Processing

Work processing is handled differently than that shown in Figure 4-12 for clients using the MAD services provided by the access layer. With the use of MAD services, the access layer processes work requests before passing them on to the queue pair. Likewise, the access layer performs post-processing on completion requests. Figure 4-13 highlights the differences.

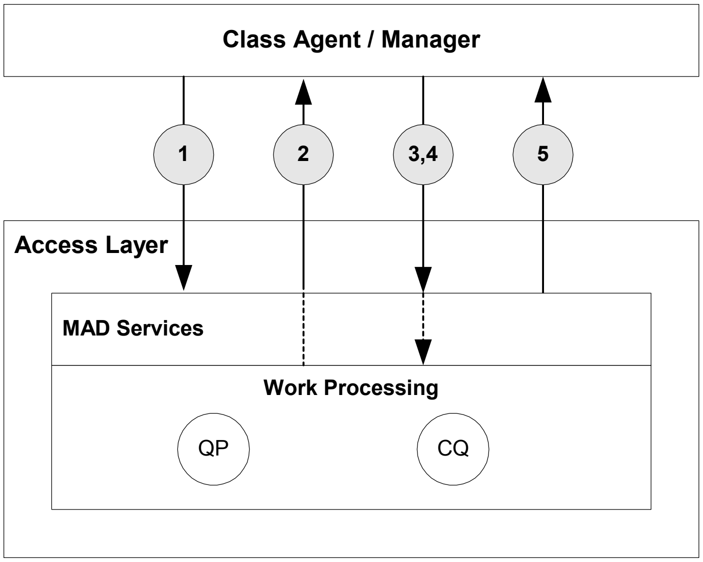


Figure 4-13: Work Processing with MAD Services

Operation:

1. Users indicate that a service should perform a specified task. The indication may come from the posting of a work request, the issuing of a MAD, or the invocation of a function call.

2-4.Steps 2-4 are similar to steps 2-4 in Figure 4-12. These steps only occur for users who request MAD services, but are using queue pair redirection. In such cases, when a user polls a completed work requests from a completion queue, the request is first given to the access layer for additional processing. The completed request is only returned to the user after all post-processing has completed.

5. The completion information is returned to the user asynchronously through a callback.

In general, most users of client specified services would only interface to steps 1 and 5 listed above.

4.3.4 Memory Manager

The memory manager provides users a way to register and deregister memory with a specified channel adapter. Memory must be registered before being used in a data transfer operation. Memory registration is considered separately from resource management for two main reasons. It is considered a speed path operation for kernel-mode users and uses synchronous deregistration. The memory registration process is shown in Figure 4-14.

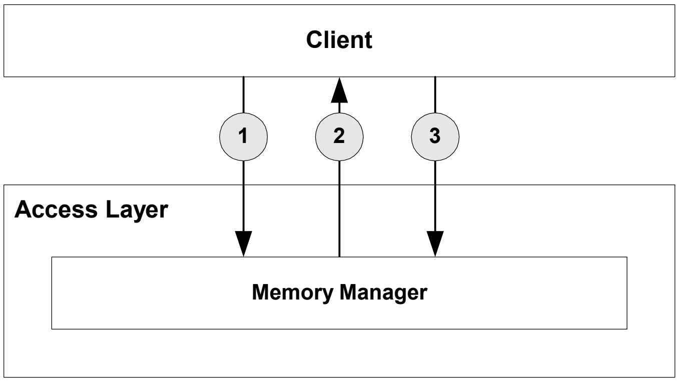


Figure 4-14: Memory Registration

Operation:

1. Clients register memory with the memory manager. Kernel-level users may register either physical or virtual memory. User-level clients cannot access to physical memory and may therefore only register virtual memory.
2. The memory manager returns a handle to the registered memory region, along with a key usable by a remote system to access the memory through an RDMA operation.
3. Once the memory is no longer being used to support data transfer operations, the client de-registers the memory region.

Requirements:

- Virtual and physical memory registration support
- Memory registration in the kernel is considered a speed-path operation

4.3.5 Operation Examples

4.3.5.1 Establishing a Connection to an IOC

The steps below provide a detailed example a client may follow to establish a connection to an IO controller.

Operation:

1. A client registers for notification of an IO controller with the Plug and Play manager. Refer to section 4.3.1.4 for information on the Plug and Play manager.
2. The access layer invokes a callback function to notify the client of the IO controller.
3. The client creates a protection domain.
4. The access layer returns a handle to the protection domain the client.
5. The client creates a CQ.
6. The access layer returns a handle to the CQ the client.
7. The client creates a QP.
8. The access layer returns a handle to the QP the client.

9. The client obtains path records to the IO unit by querying the subnet administrator using information provided to the notification callback function. Refer to section 4.3.1.1.3.
10. The access layer returns the requested path records to the client.
11. The client formats a connection request structure providing the QP handle and selected path record and initiates the connection request process through the connection manager. Refer to section 4.3.1.2.
12. The access layer performs the connection request and invokes the client callback with the connection reply.
13. The client accepts the connection through the connection manager.
14. The access layer transfers a ready to use datagram to the remote node and returns to the client.
15. The client QP is now connected and ready for use.

4.3.5.2 Client Generated MAD Transaction

The steps below provide a detailed example a client may follow to perform a MAD transaction with a class management entity.

Operation:

1. The client obtains the service record information of the remote node by querying the subnet administrator via the access layer. Refer to section 4.3.1.1.3
2. The access layer returns the requested service record to the user.
3. The client obtains path records to the remote node by querying the subnet administrator using the information returned in the service record. Refer to section 4.3.1.1.3.
4. The access layer returns the requested path records to the client.
5. The client creates a QP. They may create their own QP or create an alias to one of the special QPs.
6. The access layer returns a handle to the newly created QP or the QP alias to the client.
7. The client initializes the QP for datagram transfers, specifying callback information for handling completed MAD work requests.
8. The client registers the QP with a management class to request MAD support from the access layer. A client can register a single QP with multiple management classes.
9. The client creates an address vector that references the remote destination.
10. The access layer returns a handle to the requested address vector.
11. The client formats a work request, indicating the QP, QKey, and address vector of the destination.
12. The access layer posts the work request for the user. If a response is expected, the access layer continues to resend the request until a response is received or the request times out.
13. The client is notified of the send completion through a send MAD callback.

4.3.5.3 Joining a Multicast Group

The steps below provide a detailed example a client may follow to join and transfer datagrams with a multicast group.

Operation:

1. The client creates a QP.
2. The access layer returns a handle to the newly created QP.
3. The client initializes the QP for datagram transfers.
4. The client may post receive work requests to the QP.
5. The client obtains the PKey of the multicast group. The method used to obtain this PKey is implementation specific.
6. The client obtains the MGID of the multicast group. The method used to obtain the MGID is implementation specific.
7. The client joins the multicast group.
8. The access layer adds the QP as a group member, attaches the QP to multicast group, and returns the member record to the client. The client may receive datagrams from the multicast group before this call returns.
9. The client obtains path records to the group MGID by querying the subnet administrator using the information returned in the service record. Refer to section 4.3.1.1.3.
10. The access layer returns the requested path records to the client.
11. The client creates an address vector that references the MLID.
12. The access layer returns a handle to the requested address vector.
13. The client formats a work request, indicating the well-known QP number, and the QKey and address vector of the multicast group.
14. The access layer posts the work request for the client.
15. The client is notified of the send completion through a send callback.

5. Architectural Specification – Subnet Management

5.1 Introduction

The OpenSM Subnet Manager infrastructure provides a facility for subnet discovery and activation. OpenSM also provides Subnet Administration capabilities to support clustering and IPC.

5.2 System Overview

The Subnet Management system resides in User Mode above the Access Layer interface, as depicted in the following diagram.

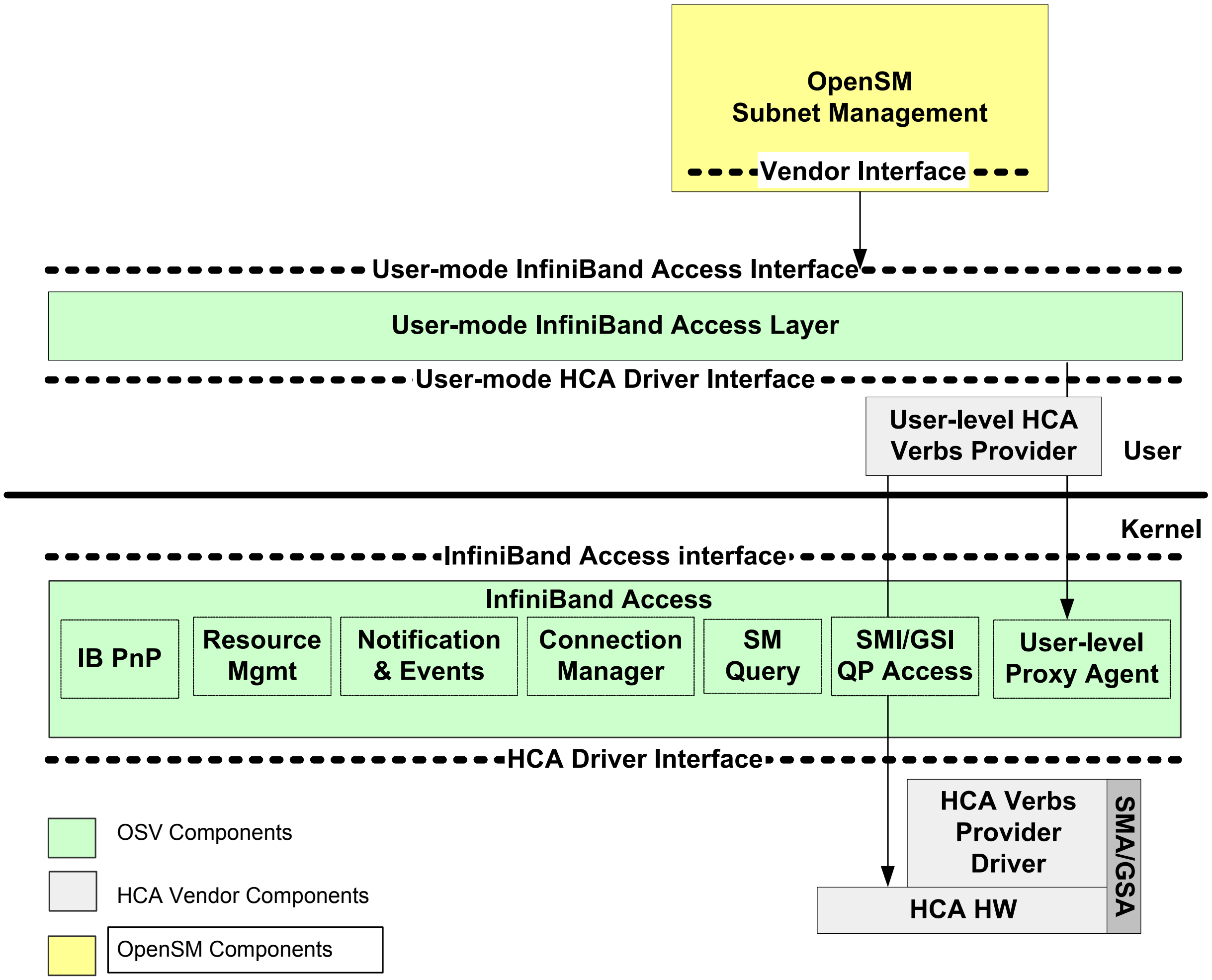


Figure 5-1: Subnet Management on the IB stack

As shown in Figure 5-1, OpenSM uses its internal Vendor Interface API's to send and receive QP0 and QP1 MAD packets. The Vendor Interface layer encapsulates and abstracts the underlying user mode InfiniBand Access API. This encapsulation allows developers to easily port OpenSM to other standard or proprietary InfiniBand access API's.

Pictorial of Component Structure

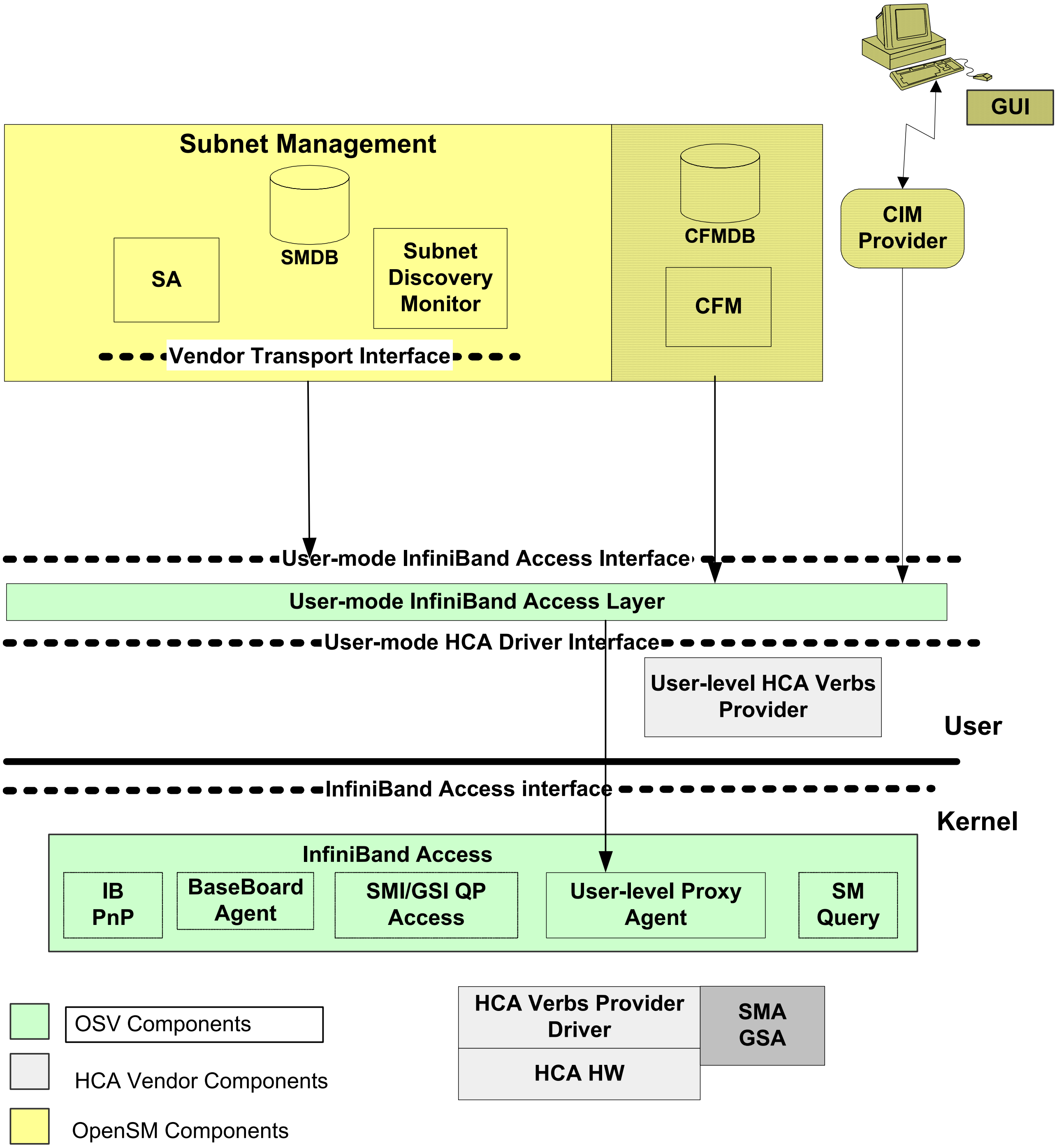


Figure 5-2: Subnet Management Components

As shown in Figure 5-2, the Subnet Management is composed of several components: Subnet Manager for discovery and configuration of the fabric, Subnet Manager Database (SMDB) for storing fabric data and Subnet Administration (SA) for handling IB queries against the SMDB, Three of the modules shown in Figure 5-2 are shaded to indicate these will not be available in the initial release of OpenSM: CFM, CIM Provider and a GUI (Graphical User Interface).

5.2.1 Relevant Specifications

The Subnet Manager conforms to the IBTA 1.0.a Specification as described in Chapters 13 and 14. The SA is described in Chapter 15.

5.2.2 List of External Interfaces

5.2.2.1 APIs

API Name and version	Brief Description
User mode InfiniBand Access Layer	See Chapter 4

5.2.3 List of Product Deliverables

5.2.3.1 User Interfaces

SM CLI	Normally the SM will start as a system daemon, but there is an optional interface to launch the SM from the command line.

5.2.3.2 Product Documentation

SM User's Guide	Documents the use of the SM
SM man page	Linux style man page for the SM CLI.
Installation Guide	Describes the installation and un-installation of the SM components

5.3 Theory of Operation

5.3.1 Initialization

The SM normally starts as a Linux daemon, started from the `/etc/rc` scripts. It may be optionally started by the system administrator from a shell.

5.3.2 SM Functionality and Interfaces

The Subnet Manager is responsible for the configuration of the subnet. On startup, the SM discovers the nodes of the fabric using directed-route SMP packets sent through the User mode InfiniBand Access Layer (uAL). The SM assigns LIDs to the switches and channel adapters, sets up the switch forwarding tables, and activates the ports on the subnet. It then continues to periodically sweep the fabric to identify new or newly disconnected links. If the SM discovers another SM on the same subnet, it will use the IBTA specified negotiation for the SM Master role. It is expected that only in the case of losing the negotiation to an SM from the same vendor that the fabric state (LID assignments, port state, forwarding table state) will be seamlessly maintained.

The SM maintains fabric data in the Subnet Management Database (SMDB), allowing the Subnet Administration (SA) to respond to SA record queries for subnet information. Generally speaking, the SM writes to SMDB and the SA reads from the SMDB.

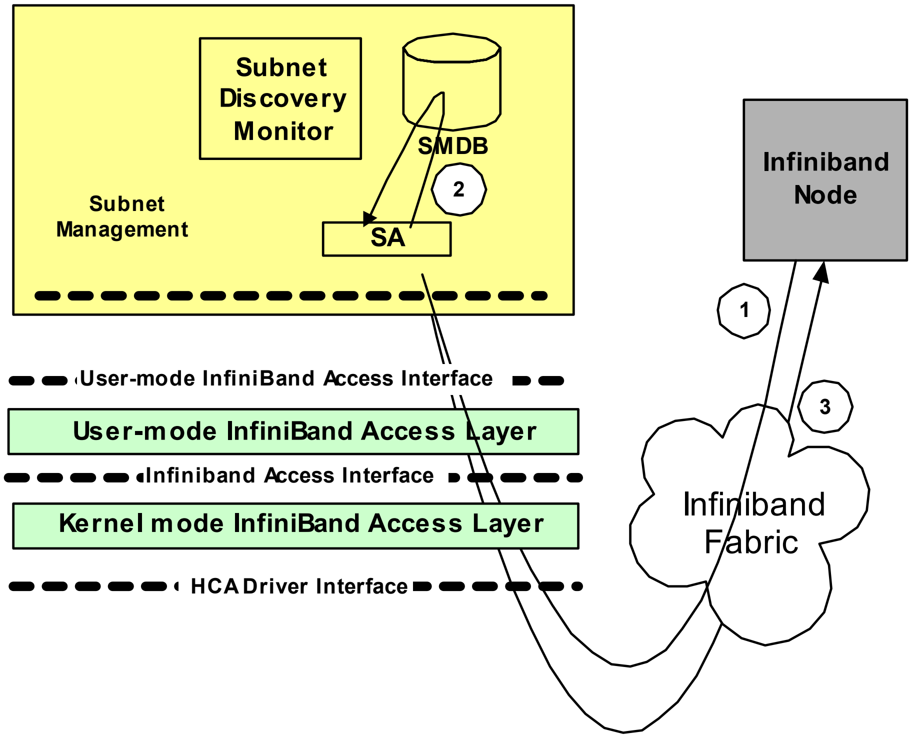
The SM plays a relatively small role in automatic path migration. When connections are established, the connecting parties may request multiple Path Records from the SA, and select one path (DGID/SGID pair) to be primary and the other to be the alternate path. For this to work properly, the SM must configure ports with LMC greater than zero, such that a port may have multiple LIDs. The switch forwarding tables must be configured so that the various paths have as little overlap as possible, given the configuration of the links on the fabric. The SA must be able to return multiple path records when these conditions exist. The SM is not involved in the actual migration of the path at the time of link failure.

The SM is responsible for managing the Multicast Forwarding Tables of the switches in the subnet. The SM is informed of changes to the multicast configuration via local communication with the SA, which in turn handles the SA MCMemberRecord messages from end nodes requesting to join or leave a multicast group.

5.3.3 SA Functionality

Through the use of Subnet Administration class MADs, SA provides access to and storage of information of several types as defined in IBTA Spec1.1 Chapter 15. For the initial release of OpenSM, SA will support only the required information (SA Attributes) that is needed to support SRP (chapter 6), IPoIB (chapter 7) and SDP (chapter 8) channel drivers. SA attributes supported are: NodeRecord, PortInfoRecord, PathRecord, MCMemberRecord and ServiceRecord.

5.3.3.1 SA Query Data Path


Subnet Discovery Monitor
SMDB
2
Subnet Management
SA
Infiniband Node
1
3
User-mode InfiniBand Access Interface
User-mode InfiniBand Access Layer
Infiniband Access Interface
Kernel mode InfiniBand Access Layer
HCA Driver Interface
Infiniband Fabric

All nodes on the fabric obtain all the fabric information, such as Node, Port or Path Records, by sending standard SA Record queries (1) to the SA, using the User Access Interface, which retrieves the appropriate data (2) from the SMDB, and returns the Query record (3) through the SM Vendor Transport Interface via the OSV Access Layer.

5.3.4 OpenSM Internals in Brief

OpenSM is based on a dispatcher and controller model. The dispatcher is a pool of threads that handle dispatching of messages between the controllers. Each controller handles a narrowly scoped portion of the workload. For example, one such controller is the MAD controller that communicates with the vendor transport layer to receive MADs. The MAD controller parses the MAD to determine which other controller should perform the needed processing. The MAD controller then posts this MAD to the dispatcher for processing by the recipient controller as soon as a worker thread becomes available.

This modular approach allows developers to quickly add modules (controllers) that plug into the framework to support new features. Typically the number of worker threads in the dispatcher equals the number of CPUs in the system. All controllers run in the context of the dispatcher thread that invokes them. External components such as the SMDB are shared by all the controllers. Refer to OpenSM HLD/LLD for details on this design.

5.3.5 OpenSM Multi-threaded Discovery

OpenSM performs fully multi-threaded subnet sweeps (initial subnet discovery is simply the first sweep). One implication is that during any subnet sweep, OpenSM may discover switches and nodes in a different order than during a previous sweep. OpenSM is designed such that the order in which it discovers subnet objects is irrelevant to both the internal representation of the subnet in the SMDB and the final configuration assigned to the subnet itself.

5.3.6 OpenSM Non-disruptive Subnet Configuration

To minimize traffic disruption, OpenSM attempts to preserve the existing configuration of a subnet. This situation can occur when OpenSM inherits a subnet due to failure of an established SM, when the OpenSM node itself is rebooted in an active subnet, or when two or more active subnets merge. Where possible OpenSM preserves existing LID assignments. LID conflicts are resolved by reassigning LID values to one of the conflicting nodes.

5.3.7 LMC > 0

OpenSM supports Lid Mask Count (LMC) values greater than 0, which allows multiple paths between endpoints on a subnet. Specifically, the number paths permitted is 2^{LMC}. Given equal hop count paths, OpenSM attempts to minimize path overlap when configuring switch forwarding tables. OpenSM will not configure a paths with less overlap if those paths contain more hops than an overlapping path. Future implementations may consider other factors in redundant path selection such as link width and the number of VLs supported by alternative paths.

The LMC value selected is a user configuration parameter.

6. Architectural Specification – SRP Driver

6.1 Introduction

Host systems require access to remote storage devices across an InfiniBand fabric. The method used to access these devices is defined by the IO protocol. The SCSI RDMA Protocol (SRP) developed by the ANSI INCITS T10 Technical Committee is designed to take full advantage of the features provided by the InfiniBand Architecture. In addition, the SCSI command set is widely used throughout the industry and is applicable to a wide variety of device types. SRP allows a large body of SCSI software to be readily used on InfiniBand Architecture and is rapidly emerging as the protocol of choice for block-based storage. This section describes the architecture of the Linux SRP device driver.

6.2 Overview

User level processes typically access storage devices through a file system. The Linux operating system supports a number of file systems. The range of file systems supported is made possible through a unified interface to the Linux kernel known as the Virtual File System (VFS). The VFS interface provides a clearly defined interface between the kernel and the different file systems. VFS maintains internal structures, performs standard actions, and forwards tasks to the appropriate file system driver.

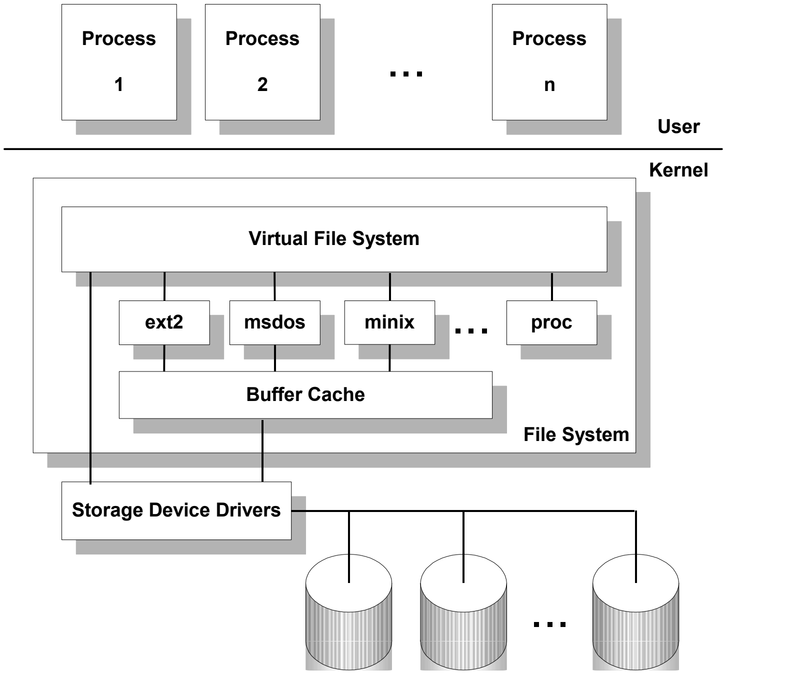


Figure 6-1: Linux File System Components

The central demand made of a file system is the purposeful structuring of data to allow high performance and randomized access. Linux employs a dynamic buffer cache to increase the performance of block devices accessed through the file system. In addition to storage device access through a file system, a process may also access a raw or character storage device directly, bypassing the file system and buffer cache components.

The storage device drivers provide access to physical devices by abstracting the details of the underlying hardware interface, for example IDE or SCSI. The SCSI device driver stack provides access to devices supporting SCSI protocols. The Linux SCSI framework contains an abstraction layer called the SCSI mid-layer. This layer provides upper-level drivers and applications with a device-independent set of interfaces to access devices. The SCSI mid-layer routes IO requests to low-level SCSI Host Bus Adapter (HBA) specific drivers. Low-level SCSI drivers provide access to devices through particular HBA hardware interfaces. Multiple low-level SCSI drivers may be loaded at the same time as required by the underlying hardware. This model simplifies the implementation of both the hardware-specific HBA drivers (the low-level SCSI drivers) and applications. The SRP device driver is implemented as a Linux low-level SCSI device driver.

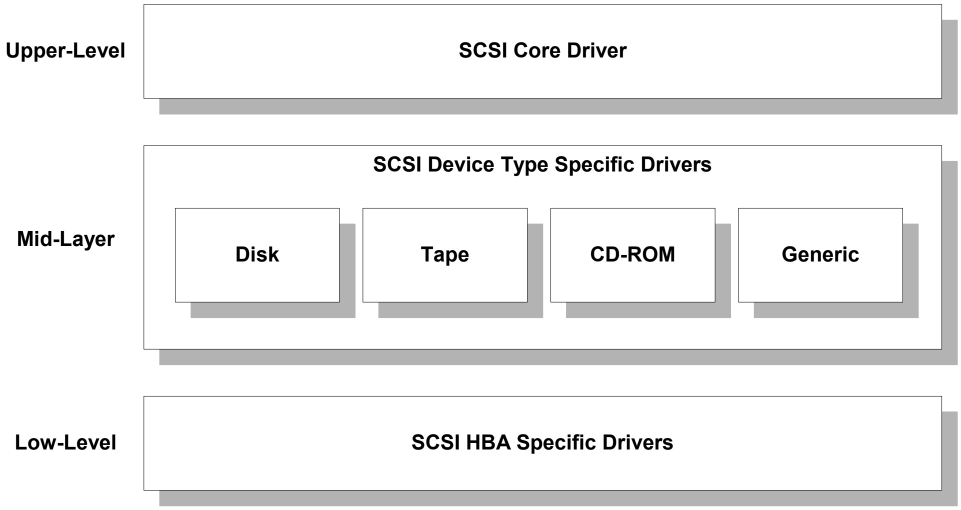


Figure 6-2: Linux SCSI Driver Stack

The SRP device driver differs from traditional low-level SCSI drivers in Linux in that the SRP driver does not control a local HBA. Instead, it controls a connection to an IO controller to provide access to remote storage devices across an InfiniBand fabric. The IO controller resides in an IO unit and provides storage services.

The SRP device driver uses the interface provided by the InfiniBand Access Layer to communicate across an InfiniBand fabric to an IO controller. Thus, the SRP device driver is not dependent on any particular channel adapter implementation.

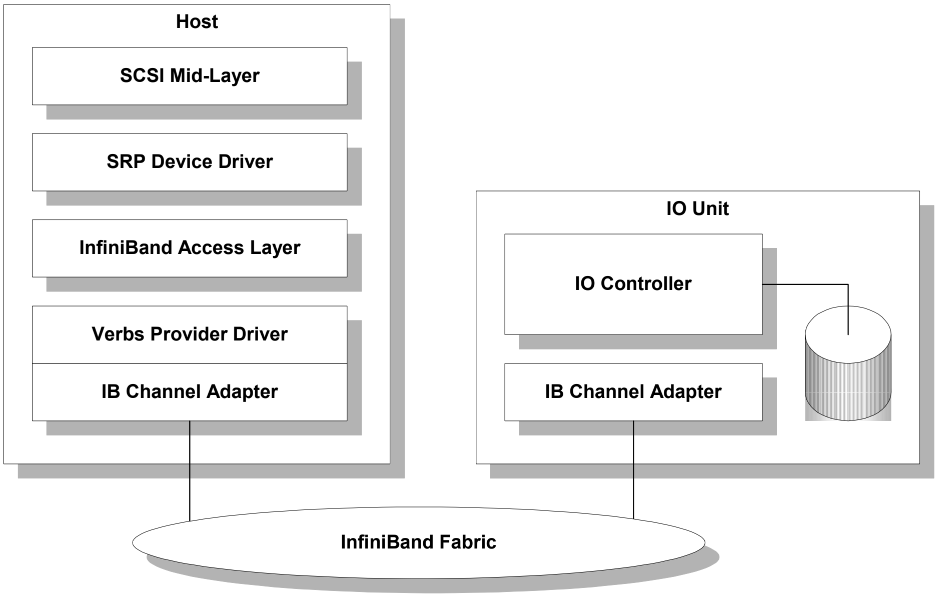


Figure 6-3: SRP Device Driver / IO Controller Relationship

The SRP device driver is known as the SRP initiator whereas the service on the IO controller is known as the SRP target. The SRP Specification defines the communication protocol used between the initiator and the target.

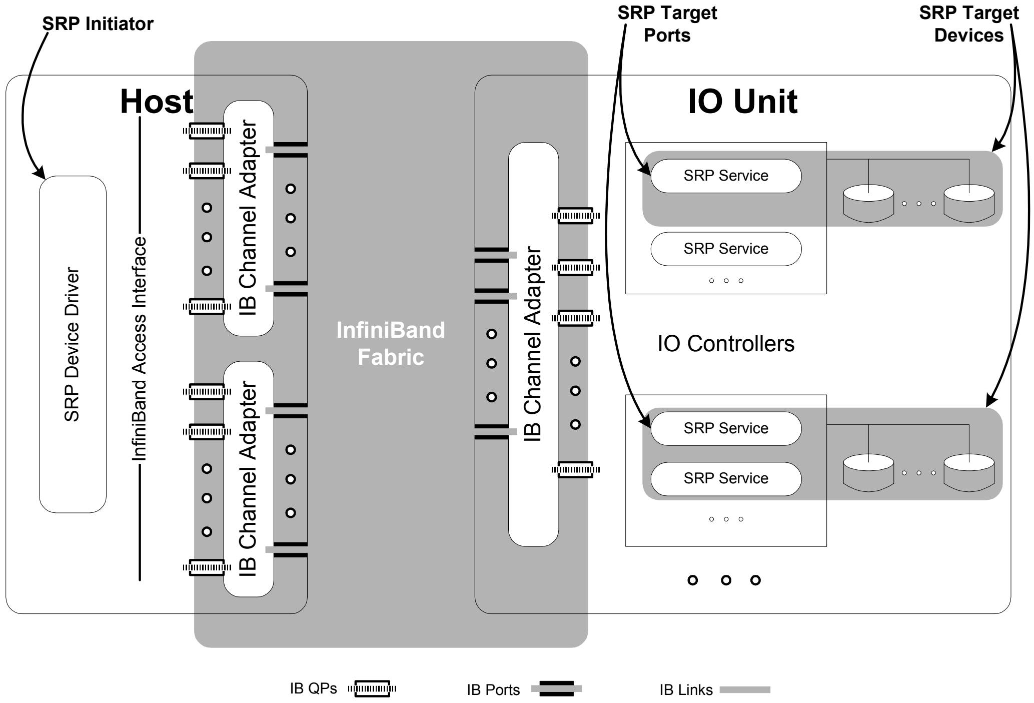


Figure 6-4: SRP Architecture Mapping

The SRP protocol provides transport services to enable a basic client-server model where an initiator presents SCSI tasks to a target for execution. All operations in this model use reliable service connections across the InfiniBand fabric. SRP defines the message format and behavior required to transfer commands and data between an initiator and a target. SCSI commands and completion status are exchanged asynchronously using message send operations.

SRP defines a buffer management mechanism that allows a target to limit the number of requests that may be queued on the target for execution. A target may use this mechanism to manage internal resources, for example, to dynamically allocate message buffers among multiple initiators. This allows the target to provide optimal use of limited resources and improve overall system performance.

In SRP, the target performs all device data transfers to or from initiator memory using RDMA operations. An initiator allows RDMA access from the target by registering its data buffer memory. Information describing the registered data buffer memory is included in the SRP command.

A typical SRP IO transaction is as follows:

1. The initiator builds an SRP request message that contains a SCSI command, a device logical unit number, and a data buffer memory descriptor, and sends the request to the target.
2. The target receives the SRP request and performs an RDMA operation to transfer the initiator data buffer memory contents to or from the device.
3. The target builds an SRP response message indicating the completion status of the request and sends the response to the initiator.

The initiator can also perform SRP task management operations, for example, to abort a task on the target. In addition, the target can send messages to the initiator describing asynchronous events, such as the insertion of new media into a removable device.

6.3 Theory of Operation

The following subsections describe the operation of the Linux SRP device driver.

6.3.1 Load / Unload

The InfiniBand Access Layer provides a Plug and Play manager. The Plug and Play manager maintains a mapping of IO controller profile class, subclass, and protocol identifier values to channel drivers. If a channel driver is not present at the time the bus driver needs to notify it of a new IO controller, then the Plug and Play manager will load the channel driver. The Plug and Play manager delivers add and remove notifications to a channel driver asynchronously through a function call interface registered with the bus driver when the channel driver is loaded.

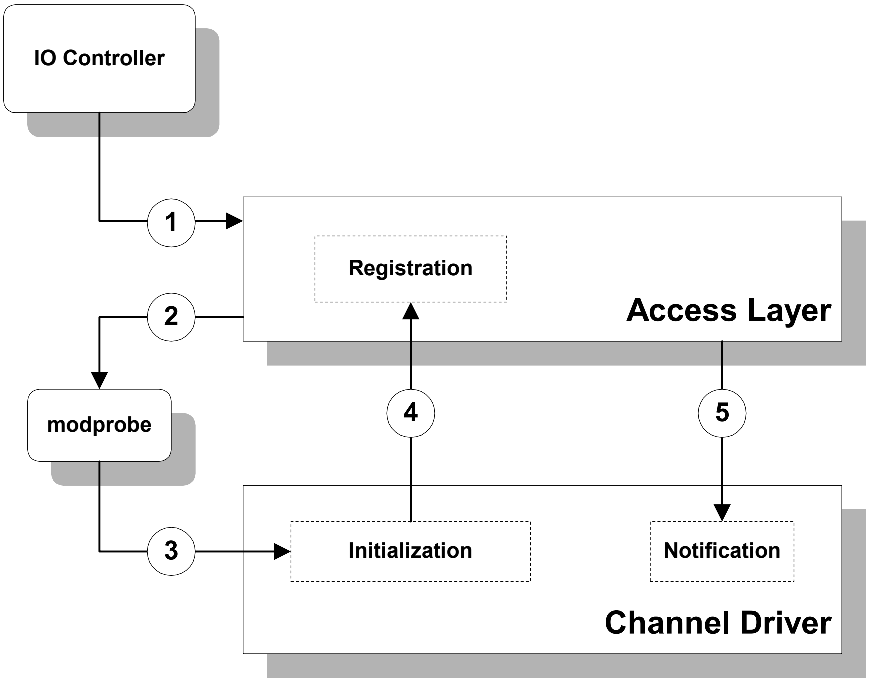


Figure 6-5: Channel Driver Load Sequence

1. The Access Layer receives notification of a new IO controller.
2. The Access Layer loads the appropriate channel driver.
3. The channel initialization function is called.
4. The channel driver registers with the Access Layer.
5. The Access Layer notifies the channel driver of a new IO controller instance.

6.3.2 Initialization

The Linux SCSI mid-layer provides a generic module initialization routine for low-level drivers to use. This routine is the first entry point, called only once, to initialize the low-level driver. During initialization, the low-level driver is expected to register with the Linux SCSI mid-layer. However, the Linux SCSI mid-layer will not properly handle the addition of subsequent SCSI controllers after this registration has been performed. For this reason, the SRP driver initialization routine is different from most low-level SCSI driver initializations. The SRP registers with the SCSI mid-layer only when the driver is certain that it has been informed of all SRP target ports assigned to the host. To achieve this certainty, the SRP driver queues a timer call to perform the registration a small amount of time in the future, which sets a grace period after the SRP driver is notified of an target port. If another target port is added to the SRP driver during the grace period, the grace period starts over. If the grace period ends without a new call to add a target port, the registration function is called allowing the SCSI mid-layer to detect devices, and complete the initialization process.

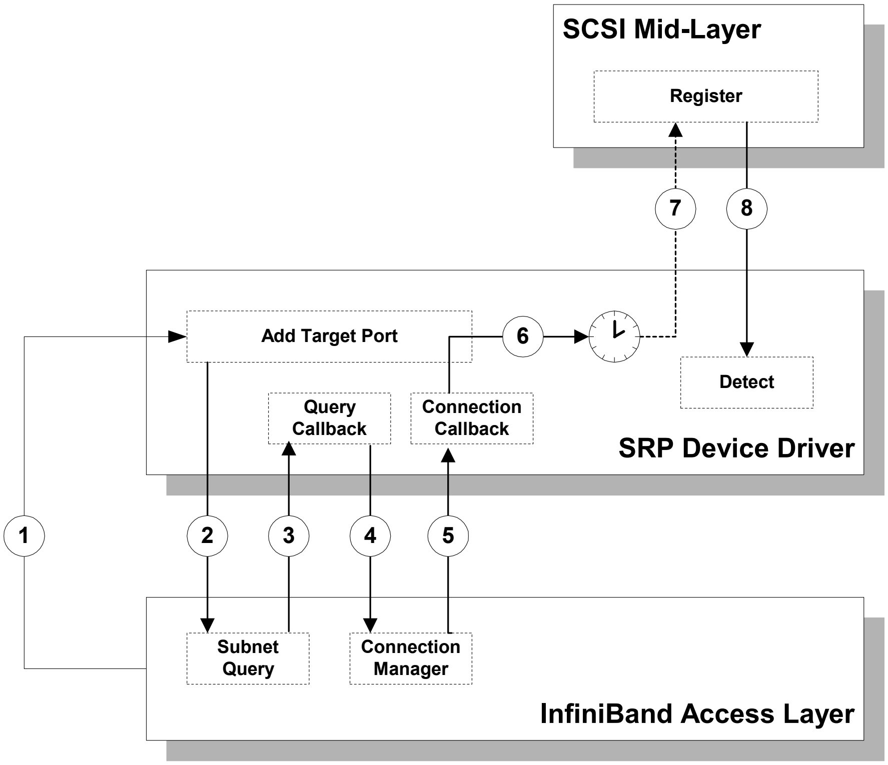


Figure 6-6: SRP Device Driver Initialization Sequence

1. The Access Layer notifies the SRP device driver of a new target port.
2. The SRP driver issues a subnet query for connection path records.
3. The SRP driver query callback function is invoked.
4. The query callback function issues a connect request to the connection manager.
5. The SRP driver connection callback is invoked.
6. The SRP driver add target port function is signaled that the connection is established which starts the grace period timer.
7. The grace period timer expires without a restart and registers with the SCSI mid-layer.
8. The SCSI mid-layer registration routine calls the SRP driver to detect the number of HBAs.

6.3.3 Connection Management

6.3.3.1 Connection Establishment

A switched fabric may allow several connection paths between a host and an IO controller. The SRP device driver obtains a list of one or more path records from the InfiniBand Access Layer and maintains an index into the path record list to establish a connection to the IO controller. If a path record query times out, the driver may retry the query. If a connection request times out, the driver may retry the request on the current path before advancing to the next path record in the list. If none of the paths is successfully connected, the driver will attempt to acquire a new list of path records from the access layer from which to restart the connection process. Once established, a connection will allow normal IO transaction processing.

6.3.3.2 Connection Failover

If the underlying channel adapter supports automatic path migration (APM) and multiple paths exist between the host and the IO controller, the SRP device driver will establish a connection with primary and alternate paths. Otherwise, the SRP device driver will perform connection error handling and failover in software. If an unexpected error occurs on an established connection, the SRP device driver will advance to the next path record in the list if possible and re-initialize the connection from that point, wrapping back to the beginning of the path list if necessary. Re-initializing a connection may include loading and enabling a new alternate path for APM or establishing a new connection through the connection manager.

6.3.4 Command Processing

SCSI commands are delivered to the SRP device driver through entry points provided when the driver registers with the Linux SCSI mid-layer. When the mid-layer has a SCSI command for a device controlled by the SRP driver, the queue command entry point is called with a pointer to a SCSI request structure as the argument. The SRP driver registers the host buffer memory, formats an SRP request, and sends it in a message to the target. Included in the request is the SCSI command and any necessary information describing the host buffer memory to or from which data will be transferred. The target interprets the request and executes the command using its local storage resources. Data movement between host memory and the target is facilitated by RDMA. The RDMA must be complete before the host receives the target SRP response message. The SRP response notifies the host that the IO transaction has completed and the completion status. The host buffer memory is deregistered when the IO transaction completes.

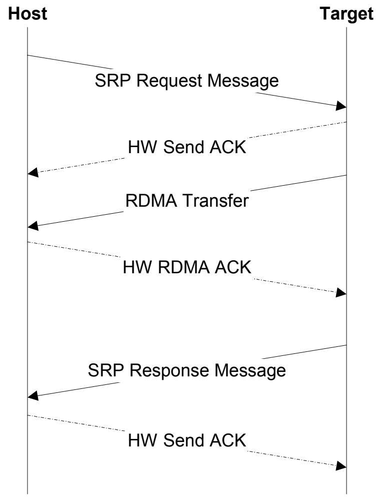


Figure 6-7: SRP IO Transaction Diagram

The SRP device driver is event driven. These events include the following:

- Arrival of a SCSI command from the mid-layer
- Completion of a message send operation from the InfiniBand Access Layer (send messages are SRP requests or responses destined for the target)

- Completion of a message receive operation from the InfiniBand Access Layer (received messages are SRP responses or requests from target)
- Detection of error conditions

Note that because send and receive operations are independent, involve other software layers, and may utilize different completion queues, it is possible to receive an SRP response message from the target before the corresponding SRP send message request has completed. The SRP driver design must allow for this scenario.

6.3.5 Asynchronous IO

The SRP device driver has the ability to process multiple IO requests simultaneously. For each IO request from the SCSI mid-layer, the SRP driver maintains a data structure for the duration of the request's life – that is, from the time the request arrives until the time the completion response message arrives from the target. The data structure contains details about the request and its state. The address of this structure serves as the SRP driver context value. Asynchronous completion events use this context to associate the completion with one of perhaps many outstanding tasks.

6.3.6 Transaction Ordering

The SRP device driver provides an interface to random and sequential access devices. The SRP device driver does not maintain a different transaction ordering policy based on the class of device, it simply passes IO requests and responses through between the host client and device server. Therefore, to provide a uniform internal design and mode of operation, the SRP driver maintains strict ordering for all requests and responses. Multiple IO requests may be in process simultaneously, allowing the target device, adapter, or SCSI device class driver to reorder requests based on the policy of that layer to maintain data consistency.

6.3.7 Multithreading

The SRP device driver has the ability to process multiple IO requests simultaneously; therefore, critical data structures must be accessed using thread-safe mechanisms. The design of the SRP driver will minimize the extent of any locks to increase parallelism on SMP systems and to achieve maximum CPU effectiveness (IOps / % CPU utilization).

6.3.8 Interrupt Processing

The SRP device driver differs from traditional low-level SCSI drivers in Linux in that the SRP driver does not control a local HBA. Instead, the SRP driver controls a connection through an InfiniBand host channel adapter. Hardware interrupts from the channel adapter are handled by the HCA specific driver and delivered to the InfiniBand Access Layer. The access layer notifies the SRP driver of these events through a registered set of callbacks. The execution priority level of the callback is determined by the specific event and notification mechanism used by the InfiniBand Access Layer.

6.3.9 Error Handling

The SRP device driver can encounter errors from several sources. For example, errors may originate from the InfiniBand access layer, the IO protocol, the operating and IO subsystem interfaces, and the underlying IO device.

6.3.9.1 Transport Errors

The SRP device driver establishes connection-oriented, acknowledged channels that provide a reliable interface to remote IO controllers. Any transport errors occurring on the channels will result in the migration or destruction of that connection. When a transport error occurs, the SRP driver will attempt to failover to a new connection and resume IO processing.

6.3.9.2 Invalid Requests

It is assumed that the target is properly designed and implemented such that invalid messages are not sent to the host. If the host receives an invalid request or response message, the protocol engine will respond with the corresponding protocol error notification message to the Linux SCSI mid-layer.

6.3.9.3 Device Errors

Errors from the underlying devices are considered a natural part of IO processing. Such errors are converted into the appropriate error condition status and delivered to the Linux SCSI mid-layer for further processing. The SRP device driver makes no assumptions about the underlying device or the intended use of that device. The SRP device driver does not retry failed IO transactions. The underlying device, or the SCSI mid-layer may perform retries.

6.3.10 OS Interfaces

The SRP device driver is implemented as a Linux low-level SCSI driver. As such, the SRP device driver conforms to the standard OS low-level SCSI driver interfaces.

6.3.11 InfiniBand Access Interfaces

The SRP device driver resides above the InfiniBand Access layer and requires capabilities:

- Dynamically loaded on demand.
- Notification of IO controller addition or removal.
- Obtain an IO controller profile.
- Obtain the service entries for an IO controller.
- Obtain a list of path records to an IO controller.
- Create a reliable connected channel to an SRP target port.
- Send and receive SRP messages asynchronously.
- Separate or combined send and receive completion queues.
- Completion processing using callbacks or polling.
- Register / Deregister physical memory for remote access.
- Notification of channel errors.
- Notification of port state changes.

7. Architectural Specification – IPoIB Driver

7.1 Introduction

IPoIB provides standardized IP encapsulation over IBA fabrics as defined by the IETF. An IPoIB link driver interfacing to the lower edge of the IP network stack performs this encapsulation. All data transfers use the unreliable datagram service of IBA. In addition to unicast transfers, the IPoIB RFC requires the use of the IBA UD multicast service.

The primary responsibilities of the IPoIB driver are performing address resolution to map IPv4 and IPv6 addresses to UD address vectors and the management of multicast membership. The following sections provide an overview of how these responsibilities are performed in a Linux environment.

7.2 System Structural Overview

The IPoIB driver integrates into the Linux network stack as a layer-2 network driver. The network driver is responsible for constructing/deconstructing its own headers, transmitting frames, as well as receiving frames and routing them to the appropriate layer-2 network driver user.

Physical link addressing is unique to each type of interconnect (i.e. Ethernet, ATM, FDDI, etc.). Therefore, the IP protocol suite defines a physical address resolution process that is responsible for mapping IP addresses to physical address. For IPv4, this mapping is performed through the Address Resolution Protocol (ARP). The IPv6 protocol performs this mapping through a Neighbor Discovery (ND) protocol using ICMPv6. All of these resolution protocols require broadcast and multicast services. For IBA, this means interacting with the fabric Subnet Manager (SM) to create, join, leave, and delete multicast groups.

Due to the way IBA routes packets through the network, it is difficult for one node to tell another the complete address vector required for end-to-end communication. For this reason, the IPoIB RFC defines a link address as a port GID + QPN. The sender is responsible for contacting the SM for complete addressing information to any particular end node based on the source and destination port GID and P_Key. Therefore, full address resolution requires an additional step not found in other links. The first step is the broadcast request followed by a unicast reply to exchange GID/QPN information. The next step is to contact the SM, through the Access Layer, to obtain a PathRecord to the destination node. The extra step means that the ARP and ND implementation of the Linux network stack cannot be used unmodified. Although there is a relatively high overhead to this process, address caching employed by the Linux network stack mitigates how often this process is used.

In addition to interactions with the SM, the IPoIB network driver requires the Unreliable Datagram (UD) services of the Access Layer. As stated above, this includes both unicast, broadcast, and multicast services.

Finally, the IPoIB driver interfaces to Linux IB Plug-and-Play events through the Access Layer. Each time an CA port becomes active for the first time, an IPoIB network adapter is created for each partition configured as an IPoIB subnet and registered with the Linux networking subsystem. Subsequent port state notifications will result in link-up and link-down notifications to the Linux network subsystem.

7.2.1 Pictorial of Components Structure

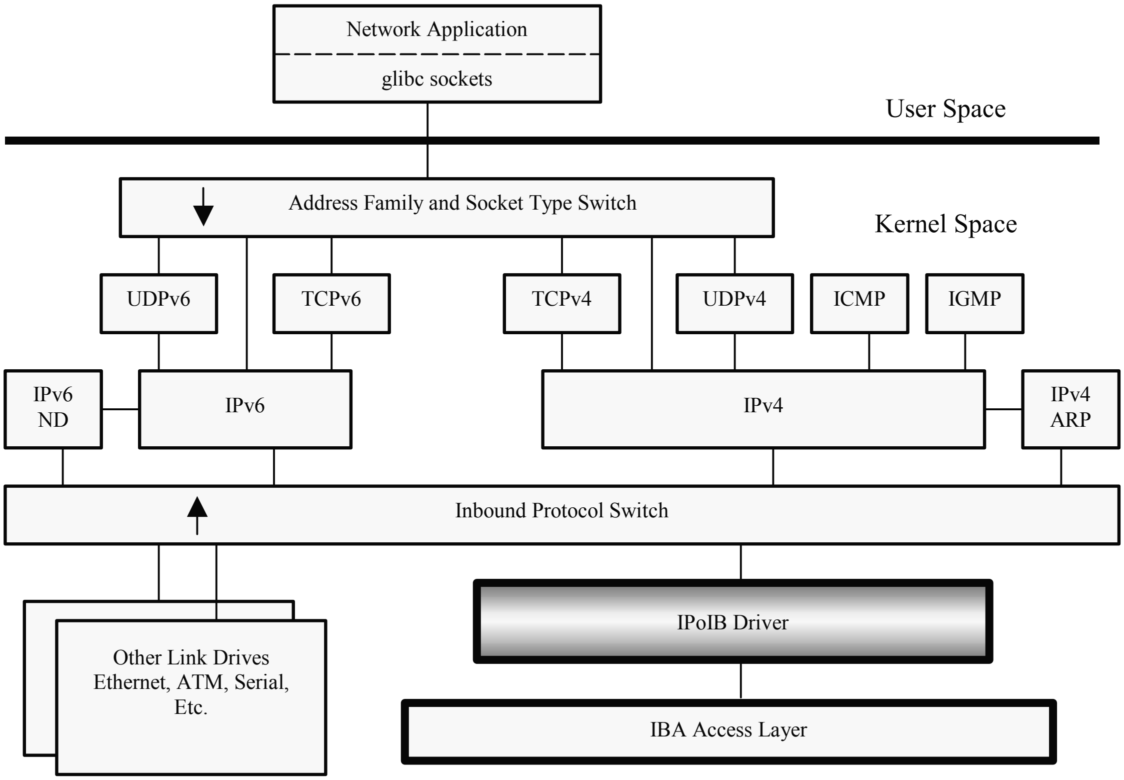


Figure 7-1: IPoIB components within Linux network stack

The diagram above illustrates the major networking related components in the Linux OS. The boxes with heavy borders identify the IBA specific elements. The darkly shaded boxes identify the components that are specific to the IPoIB driver implementation. Except for the socket switch at the top and the protocol switch at the bottom, flow through connecting components should be considered bi-directional.

When a socket is created by an application, it specifies an address family (i.e. AF_INET), and a type (i.e. SOCK_STREAM, SOCK_DGRAM). This information is used by the socket switch to direct requests to the appropriate network component for processing. Data flows only from user to kernel space through the switch. Any data returned to the user from the underlying protocols is done directly.

At the point an IP layer needs to place a frame on the wire, it will request the hardware address from an internal “neighbor” facility associated with the a link driver. If the hardware address for the target IP address is unknown, the neighbor facility will invoke the link-specific address resolution protocol to resolve it. The IPoIB RFC defines an address resolution protocol for mapping IPv4 and IPv6 addresses to IBA address vectors. These protocols are implemented by the standard Linux address resolution mechanisms with some additional “shadow” support from the IPoIB Driver.

With hardware address in hand, the IP layer calls the link specific entry point to have a frame placed on the wire. For IP addresses associated with the IP subnet serviced by IBA, this would be an entry point in the IPoIB driver. Note that the transmission of frames bypasses the Inbound Protocol Switch.

When the IPoIB driver receives frames, it must determine the intended destination lower edge protocol. The IPoIB working group has defined an encapsulation header that includes a four-byte protocol field for this purpose. Once the incoming protocol type is determined, it is stored in the receiving socket buffer and sent to the Inbound Protocol Switch. The switch uses the protocol value to direct the frame to the appropriate processing function.

7.2.2 Component Details

Component Type	**IPoIB Driver**: A loadable Linux layer-2 network link device driver.
Purpose	The IPoIB driver provides a standard interface between the Linux network stack and the IB fabric suitable for implementing the IPoIB RFC.
Functionality	The IPoIB driver implements the address resolution protocols for IPv4 and IPv6, as well as sends and receives encapsulated IP packets through the UD IBA facility.
Externally resources required	The IPoIB driver utilizes Linux network resources such as sk_buff's and control structures. It will also require CA resources such as UD queues and context structures.
Externally visible attributes	All visible attributes are defined by the Linux *net_device* structure for the link driver and the *neigh_table* structure for address resolution.
External interfaces	All external interfaces are defined by the Linux *net_device* and *neigh_table* structure.
Internal interfaces	The IPoIB driver is a thin veneer between the Access Layer and the Linux network stack. It does not make use of any significant internal subsystems.
Dependencies and Inter-relationships with other major components	The IPoIB driver is dependent on the HW Access Layer.
Requirements and constraints	The IPoIB implementation must comply and demonstrate interoperability with the IETF IPoIB RFC. Modifications to the Linux kernel and network stack must be kept to a minimum.
Development Verification/Validation/Analysis/Test strategies	TBD

7.2.3 List of Third Party Components

The IPoIB driver implementation does not use any third party components.

7.2.4 List of External Interfaces

7.2.4.1 OS Interfaces

OS Interface Name and version	Brief Description
Linux *struct net_device* (v2.4)	The *net_device* structure defines the interface for layer-2 network driver.
Linux *struct neigh_table* (v2.4)	The *neigh_table* structure defines the interface for general address resolution protocols used in ARP and ND.

7.2.4.2 APIs

API Name and version	Brief Description
ioctl(2) extensions for SIOCDIPIBARP, SIOCGIPIBARP, and SIOCSIPIBARP	These ioctl(2) types will allow user mode applications to set, get, and delete IPoIB ARP cache entries. This will be particularly useful for the Sockets Direct implementation.

7.2.5 List of Product Deliverables

7.2.5.1 User Interfaces

Component Name	Brief Description
ibarp(8)	An administrative interface to display, set, and delete IPoIB ARP cache entries.

7.2.5.2 Programs without Interfaces

Component Name	Brief Description
IPoIB Driver	Kernel loadable network device driver.

7.2.5.3 Programmatic Interfaces (APIs)

Component Name	Brief Description
ioctl(2) extensions for SIOCDIPIBARP, SIOCGIPIBARP, and SIOCSIPIBARP	These ioctl(2) types will allow user mode applications to set, get, and delete IPoIB ARP cache entries. This will be particularly useful for the Sockets Direct implementation.

7.2.5.4 Internal Libraries

Component Name	Brief Description
None	

7.2.5.5 Product Documentation

Component Name	Brief Description
IPoIB driver manual page	Unix style manual page that describes IPoIB driver capabilities and command line options when loaded through insmod(8)
ibarp(8) manual page	Unix style manual page that describes the usage of the ibarp(8) utility.

7.3 Theory of Operation

The goal of the IETF IPoIB architecture is to carry IP protocol frames over an IBA fabric. The easiest way to accomplish this is to present a network adapter interface to the existing IP protocol stack of the OS. The network adapter interface of the OS is well defined. The role of the IPoIB driver is to translate OS network adapter transactions to Hardware Access Layer transactions.

To succeed in this role, the IPoIB driver must perform the following tasks:

- Create an OS network adapter interface for each IPoIB partition of all active IBA ports
- Map IBA Unreliable Datagram address vectors to pseudo hardware address
- Map IP multicast address to IBA multicast groups
- Interact with the Subnet Manger to create, join, leave, and delete IBA multicast groups as well as register for fabric topology events.
- Supplement ARP and Neighbor Discovery protocols to perform PathRecord lookups
- Send and receive IP frames over the Unreliable Datagram service of Hardware Access Layer

7.3.1 Network Adapter Creation and Management

The Access Layer will call the IPoIB driver’s initialization entry point for each partition associated with an active port. IPoIB initialization will perform the following tasks:

- Create a kernel *net_device* structure to hold the state and context associated with this IPoIB subnet.
- Send a join/create multicast group request to the SM for the broadcast and all-nodes multicast group mapped to the P_Key associated with the notification.
- Register the network adapter with the kernel.

The Access Layer will also provide indications when a port leaves a partition or the port itself becomes inactive. If the port is removed from a partition or becomes inactive, the IPoIB driver will un-register and delete the network adapter instance and will leave any multicast groups that were joined on its behave.

7.3.2 Mapping IBA Address Vectors to Adapter Hardware Addresses

The networking infrastructure assumes all network adapters are associated with a linear hardware address. The Linux OS further assumes that the hardware address can be contained in a fixed array of eight bytes. The IBA address vector presents a unique challenge in this environment.

For purposes of ARP and ND protocol headers, the IPoIB RFC defines an IBA address as the concatenation of a GID, QPN, and eight reserved “Capabilities” bits. This results in an IPoIB virtual hardware address of 20 bytes.

Although we will work with Linux Fellow Travelers, changing the size of the Linux hardware address array would require recompile of the kernel as well as all network device drivers that run under it. Since this may not be practical, the IPoIB driver will provide an IPoIB hardware address mapping mechanism. This mechanism must be able to map between what is stored in Linux network structures and the IPoIB address format as well as the address vector handle needed to post a send to a UD QP.

7.3.3 ARP and Neighbor Discovery

As stated above, the IPoIB RFC defines a virtual IBA hardware address carried in ARP and ND protocol headers. Since only the GID and QPN are given, a PathRecord lookup must be performed to obtain the other required components of an address vector. This additional interaction with the SM means the IPoIB driver cannot use the standard Linux ARP protocol module without additional support.

This additional support will be provided as a “shadow” ARP/ND facility of the IPoIB Driver. The IPoIB Driver must already keep its own address structures, if nothing else, to hold the address vector handle for sending frames to a target node. Whenever the IPoIB Driver is asked to send a frame to a node that it doesn’t have a shadow entry for, one is created. At the same time, it will ask the Access Layer to return a PathRecord for the target address. While this request is pending, the transmit buffer will be placed on a queue for that shadow address entry. When the information for creating the target address vector handle has been obtained, all pending outbound frames will be sent at that time.

As with standard ARP caches, the IPoIB Driver will keep track for how often an entry has been referenced. If the entry becomes stale, it will be freed.

7.3.4 QP Usage

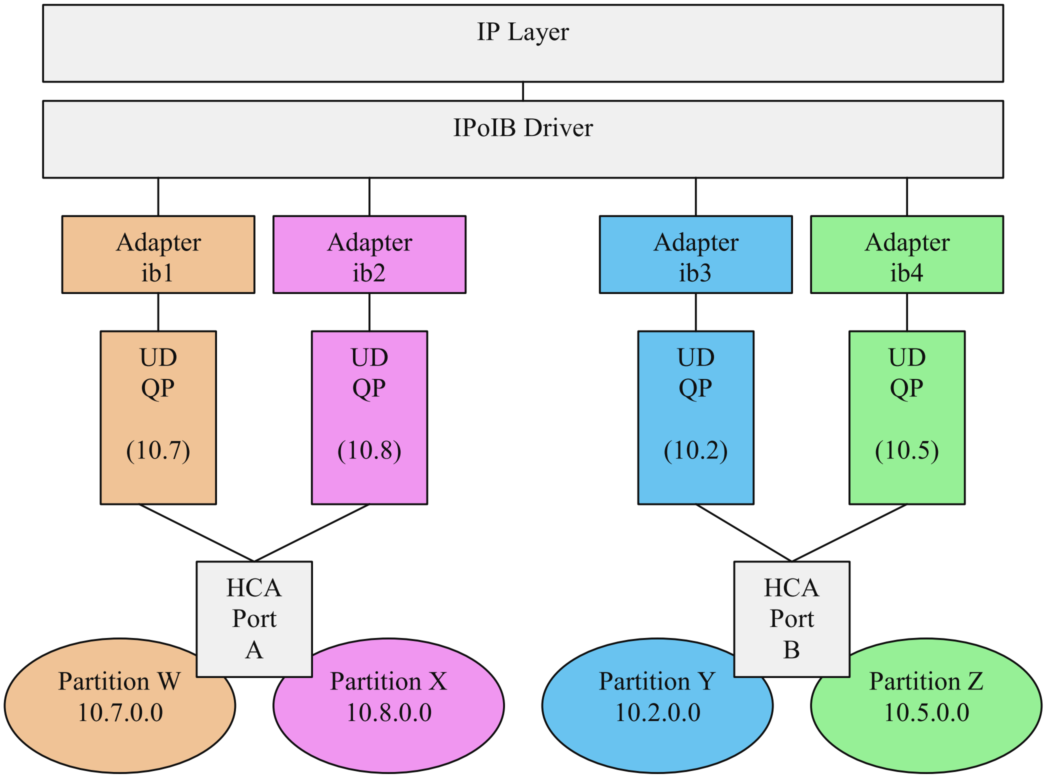


Figure 7-2: QP relationship to IBA partitions and network adapters

The link architecture of the IPoIB driver closely models an Ethernet interface. Where the MAC is the addressable entity for Ethernet, the QP is the addressable unit for IPoIB.

Using this model, the IPoIB driver will use a single UD QP per adapter instance. An adapter instance is created for each partition associated with a port. To meet IPoIB requirements, this equates an IB partition to a network subnet. All IPoIB frames for the subnet, both unicast and multicast, will be received and posted on this QP.

The driver also uses a single completion queue per adapter instance. Both send and receive completions are handled through this completion queue.

8. Architectural Specification – Offload Sockets Framework and Sockets Direct Protocol (SDP)

8.1 Introduction

The Offload Sockets Framework (OSF) enables network applications to utilize Linux sockets and File I/O APIs to communicate with remote endpoints across a system area network (SAN), while bypassing the kernel resident TCP/IP protocol stack. The offload sockets framework is completely transport and protocol independent and can be used to support multiple offload technologies. For the rest of this document, as an application of the Offload Sockets Framework, Sockets Direct Protocol (SDP) is used as the target protocol and InfiniBand as the target transport. However, other transport technologies (such as TCP Offload Engines - TOE) and protocols (such as iSCSI) can easily make use of the offload sockets framework.

The Sockets Direct Protocol (SDP) is an InfiniBand specific protocol defined by the Software Working Group (SWG) of the InfiniBand Trade Association (IBTA). It defines a standard wire protocol over IBA fabric to support stream sockets (SOCK_STREAM) networking over IBA. SDP utilizes various InfiniBand features (such as remote DMA (RDMA), memory windows, solicited events etc.) for high-performance zero-copy data transfers. SDP is a pure wire-protocol level specification and does not go into any socket API or implementation specifics.

While IP-Over-IB (IPoIB) specifies a mapping of IP (both v4 & v6) protocols over IBA fabric and treats the IBA fabric simply as the link layer, SDP facilitates the direct mapping of stream connections to InfiniBand reliable connections (or virtual circuits). The IPoIB specification is currently being created and published by an IETF working group. The IPoIB specification will define the packet level format of IP packets on the IBA fabric, and describe the InfiniBand address resolution protocol (IBARP). Conceptually, the IPoIB driver in Linux will look like a network driver and will plug-in underneath the IP stack as any standard Linux network device. The IPoIB driver exposes a network device per IBA port (and partition) on the host system and these devices are used to assign (statically or dynamically - using protocols such as DHCP) IP addresses. The SDP stack simply makes use of these IP assignments for endpoint identifications.

Sockets Direct Protocol only deals with stream sockets, and if installed in a system, allows bypassing the OS resident TCP stack for stream connections between any endpoints on the IBA fabric. All other socket types (such as datagram, raw, packet etc.) are supported by the Linux IP stack and operate over the IPoIB link drivers. The IPoIB stack has no dependency on the SDP stack; however, the SDP stack depends on IPoIB drivers for local IP assignments and for IP address resolution.

8.2 Requirements for Offload Sockets Framework in Linux

This section lists a set of requirements and goals for offload sockets framework support in Linux. The items listed in this section are by no means complete and may need to be further refined before finalizing on the best solution.

- All offload protocols/transports need to have a standard Linux network driver. This allows network administrators to use standard tools (like ipconfig) to configure and manage the network interfaces and assign IP addresses using static or dynamic methods.
- The offload sockets framework should work with and without kernel patches. To this effect, the offload protocols and transports will reside under a new offload address family (AF_INET_OFFLOAD) module. Applications will be able to create socket instances over this new address family directly. However, for complete application transparency, an optional minimal patch to the Linux kernel can be applied (socket.c) to allow re-direction of AF_INET sockets to the new AF_INET_OFFLOAD address family. The AF_INET_OFFLOAD module will work as a protocol switch and interact with the AF_INET address family. The patch also defines a new address family called AF_INET_DIRECT for applications that want to be strictly using the OS network stack. This kernel patch can be optional based on distributor and/or customer requirements.
- All standard socket APIs and File I/O APIs that are supported over the OS resident network stack should be supported over offload sockets.
- Support for Asynchronous I/O (AIO) being added to Linux. AIO support is being worked in Linux community. The offload framework should utilize this to support newer protocol and transports that are natively asynchronous. (For example, SDP stack could utilize the AIO support to support PIPELINED mode in SDP)
- Architecture should support a layered design so as to easily support multiple offload technologies, and not just SDP. Makes sure the added offload sockets framework is useful for multiple offload technologies.
- The proposed architecture should support implementations optimized for zero-copy data transfer modes between application buffers across the connection. High performance can be achieved by avoiding the data copies and using RDMA support in SANs to do zero copy transfers. This mode is typically useful for large data transfers where the overhead of setting up RDMA is negligible compared to the buffer copying costs.
- The proposed architecture should support implementations optimized for low latency small data transfer operations. Use of send/receive operations incurs lower latency than RDMA operations that needs explicit setup.
- Behavior with signals should be exactly same as with existing standard sockets.
- Listen() on sockets bound to multiple local interfaces (with IPADDR_ANY) on a AF_INET socket should listen for connections on all available IP network interfaces in the system (both offloaded, and non-offloaded). This requires the listen() call from

application with IPADDR_ANY to be replicated across all protocol providers including the in-kernel TCP stack.

- select() should work across AF_INET socket file descriptors (fd) supported by different protocol/transport providers including the in-kernel IP stack. This guarantees complete transparency at the socket layer irrespective of which protocol/transport provider is bound to a socket. .
- Operations over socket connections bound to the in-kernel protocol (TCP/IP) stack should be directed to the kernel TCP/IP stack with minimum overhead. Application bound to kernel network stack should see negligible performance impact because of offload sockets support.
- Ability to fallback to kernel TCP/IP stack dynamically in case of operation/connection failure in direct mapping of stream connections to offloaded protocols/transports. Connection requests for AF_INET sockets that fail over offload stack is automatically retried with the kernel TCP/IP stack. Once a direct mapped connection is established, it cannot be failed back to the TCP stack, and any subsequent failures are reported to application as typical socket operation failures.
- Offload Socket framework enables sockets of type STREAMS only. Other socket types will use only the OS network stack.
- Offload sockets framework will support offloading of stream sessions both within local subnet and outside local subnet that needs routing. Offload protocols/transports will have the ability to specify if they do self-routing or need routing assistance. Ability to offload stream sessions to remote subnet will be useful for TOE vendors in general and for IBA edge router vendors who map SDP sessions on IBA fabric to TCP sessions outside fabric. For protocols/transports that do self-routing, the offload sockets framework simply forwards the requests. For protocols/transports that need routing support (such as SDP), the framework utilizes the OS route tables and applies its configurable policies before forwarding requests to offload transports.
- Since the socket extensions defined by the Interconnect Software Consortium (ICSC) in the open group are work in progress at this time, the offload sockets framework will not attempt to address them in this phase. This could be attempted at a later phase.
- Offload sockets framework should not affect any existing Linux application designs that uses standard OS abstractions and features (such as fork(), exec(), dup(), clone(), etc.). Transparency to applications should be maintained.
- Offload sockets framework should support both user-mode and kernel-mode socket clients. Maintain the existing socket semantics for existing user mode or kernel mode clients.
- The offload sockets framework currently deals with only IPv4 address family. Even though the same offload concepts can be equally applied to offload IPv6 family, it is deferred for later stages of the project.

8.3 System Structural Overview

Sockets (BSD definition) are the most common API used by applications for accessing the network stack on most modern operating systems (including Linux). Most implementations of Sockets such as in Linux also support File I/O APIs (such as read, write) to operate over sockets. Offload Sockets Framework allow applications to use these same standard sockets and File I/O APIs to transparently communicate with remote endpoints/nodes across a system area network (SAN), bypassing the kernel resident TCP/IP protocol stack.

8.3.1 Existing Sockets Architecture in Linux

Currently, Linux implements the Sockets and associated networking protocol stack as a series of connected layers of software modules, all kernel resident. These layers are initialized and bound to each other during kernel start up. Figure shows Linux networking architecture.

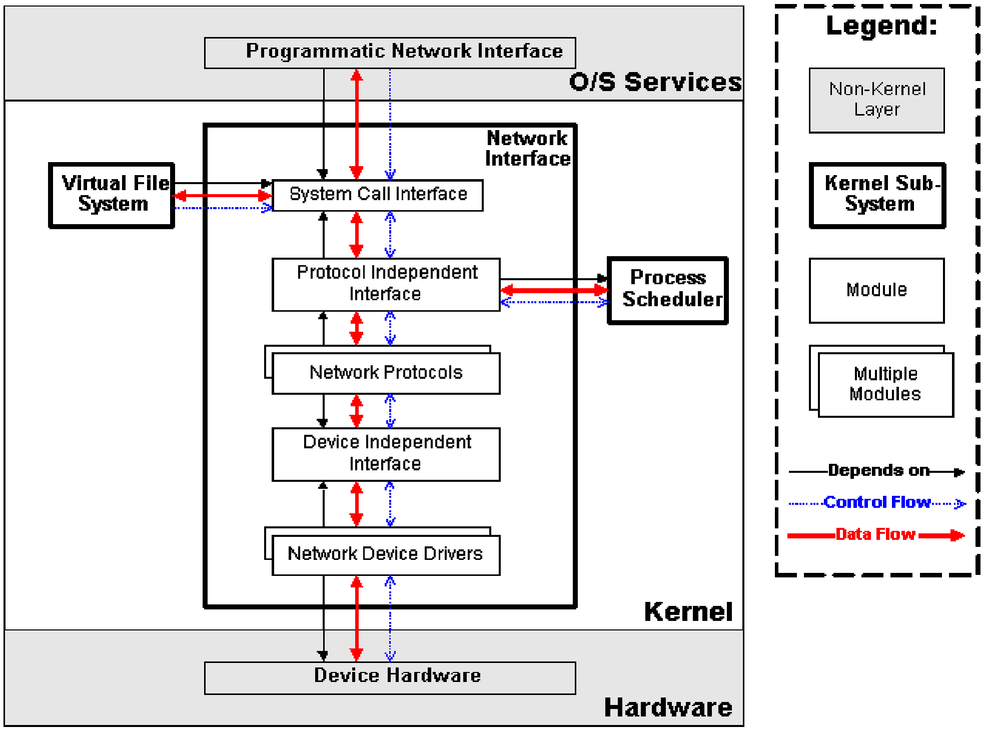


Figure 8-1: Linux Networking Architecture

Each network object is represented as a socket. Sockets are associated with processes in the same way that i-nodes are associated; sockets can be shared between processes by having its process data structures pointing to the same socket data structure.

Linux also associates a VFS inode and file descriptor, for each socket allocation, to facilitate the normal file operations over the socket handle. At kernel initialization time, the address families built into the kernel register themselves with the BSD socket interface. Later on, as applications create and use BSD sockets, an association is made between the BSD socket and its supporting address family.

The Key features of the existing Linux network architecture are:

1. Network device drivers communicate with the hardware devices. There is one device driver module for each possible hardware device.
2. The device independent interface module provides a consistent view of all of the hardware devices so that higher levels in the subsystem don't need specific knowledge of the hardware in use.
3. The network protocol modules are responsible for implementing each of the possible network transport protocols.
4. The protocol independent interface (BSD Socket Layer) module provides an interface that is independent of hardware devices and network protocol. This is the interface module that is used by other kernel subsystems to access the network without having a dependency on particular protocols or hardware.

8.3.2 Limitations in Existing Linux Sockets Architecture

The current AF_INET sockets architecture in Linux is hard-wired to the in-kernel TCP/IP network stack; making it impossible to hook into the socket API (and file I/O API) calls in user-space without standard C-library code modifications (or load time hooks).

In the current network architecture the protocol layers are tightly coupled to each other and pre-initialized at kernel initialization, making it impossible to load and select between protocol providers that possibly support the same address family (such as AF_INET).

Kernel coding shortcuts do not follow a strict protocol dispatch model within the AF_INET protocol family. Without kernel modifications, introducing new AF_INET transport protocols is impossible without providing a completely new address family.

8.3.3 Evaluations of Alternative Architectures

This section briefly surveys the related work done in this area and explores the solution space for application-transparent high performance I/O architecture (including user-mode I/O) in Linux.

Some of the previous work in this space has traded off transparency for performance. These solutions typically define a custom API that fits the underlying hardware architecture and requires applications to be recoded. An example for this approach is VI architecture that specifies its own VIPL API.

For complete application transparency, the user-mode I/O architecture needs to fit underneath the existing standard socket API. Survey of related work shows the following as some of the possible solutions for transparently supporting high-performance I/O underneath the socket and File I/O APIs:

a) User-mode implementation of high-performance sockets with direct mapping of stream connections to NIC hardware resources from user-mode. This would involve modifying the user-mode library that exports the socket and file I/O APIs (e.g. glibc), such that the socket and file I/O APIs are abstracted out into a separate offload user-mode I/O library that gets demand loaded by glibc. The I/O library will provide a bottom edge interface that can be used by specific protocol libraries (such as one for SDP) to register with it. This solution also requires changes to the socket driver in kernel for supporting operations such as select(), fork(), dup() etc. The user-mode I/O library provides two code paths: If there are no offload user-mode I/O libraries (socket provider for network I/O, file system provider for file I/O) installed, the code is same as it is currently in glibc (i.e. makes a syscall to kernel components). If user-mode providers exist, they are transparently loaded. Another option (mostly cosmetic) is to encapsulate (contain) the standard glibc library within another library. This new library exposes the same set of interfaces as the standard glibc library, and decides if a specific API call needs to be handled by it or forwarded to the standard glibc. While this solution might provide the most optimal performance for speed path operations, the major drawbacks with this approach is that it requires extensive changes to standard C-libraries and kernel socket driver to support direct user-mode mapping of stream sockets transparently to applications.

b) Define a new address family (AF_INET_OFFLOAD) for enabling any offload protocols/transports and modify the socket driver (socket.c) to transparently re-direct AF_INET sockets to AF_INET_OFFLOAD. The major difference between this option and option (a) is the additional overhead of traps to the kernel (syscalls) in speed path operations. On IA-32 architecture, the syscall overhead was measured at 805 nanoseconds on a 933 MHz processor; IA-64 performance is sub 947 nanoseconds, which implies the syscall overhead to trap to kernel is not the most significant component in today's TCP/IP stack code path. This solution restricts the kernel modifications to the socket driver to enable application transparency by re-directing sockets created under AF_INET address family to the AF_INET_OFFLOAD address family., and provides better modularity to fit into the Linux OS.

c) Changing the Linux loader to transparently hook into the socket and file I/O calls from user-mode. This requires modifying the standard Linux loader to dynamically load the user-mode I/O library into an application's address space and mangle the in-memory function tables (created when glibc was loaded) so that the socket and file I/O functions point to the equivalent functions in the new library (method often used by debuggers). The dynamic loading and function pointer mangling can be done at the time of loading the glibc library, or could be delayed until the first socket or file I/O API call makes the syscall and enters the kernel. Some of the related work shows this solution has been successfully applied for transparent performance analysis of libraries. However, the biggest drawback with this solution is that it requires changes to the loader, and makes it harder to debug.

d) Use the call intercept methods employed in strace(1) and ltrace(1) to modify the behavior of network and file I/O calls transparently to application. This approach is very similar to solution(c) and suffers the same drawbacks.

The current implementation of sockets stack in Linux does not leave room for adding high-performance network I/O support easily. Similar limitation also exists in the file system stack. Based on initial research, it appears that the kernel-mode solution (explained as option (b) above) offers almost the same performance benefits as the user-mode solution, with far less complexity and better architectural modularity to fit into the Linux OS.

8.3.4 Software Component View

The Offload Sockets Framework architecture implements high-performance socket support in the kernel by providing a new Offload Protocol Switch (OPS) via a new address family (AF_INET_OFFLOAD) module. This new address family module will support dynamic binding of offload protocols and transports under the offload family. The offload protocols will register with the offload family and the transport modules will register with the offload protocol modules. The offload sockets architecture is shown in Figure .

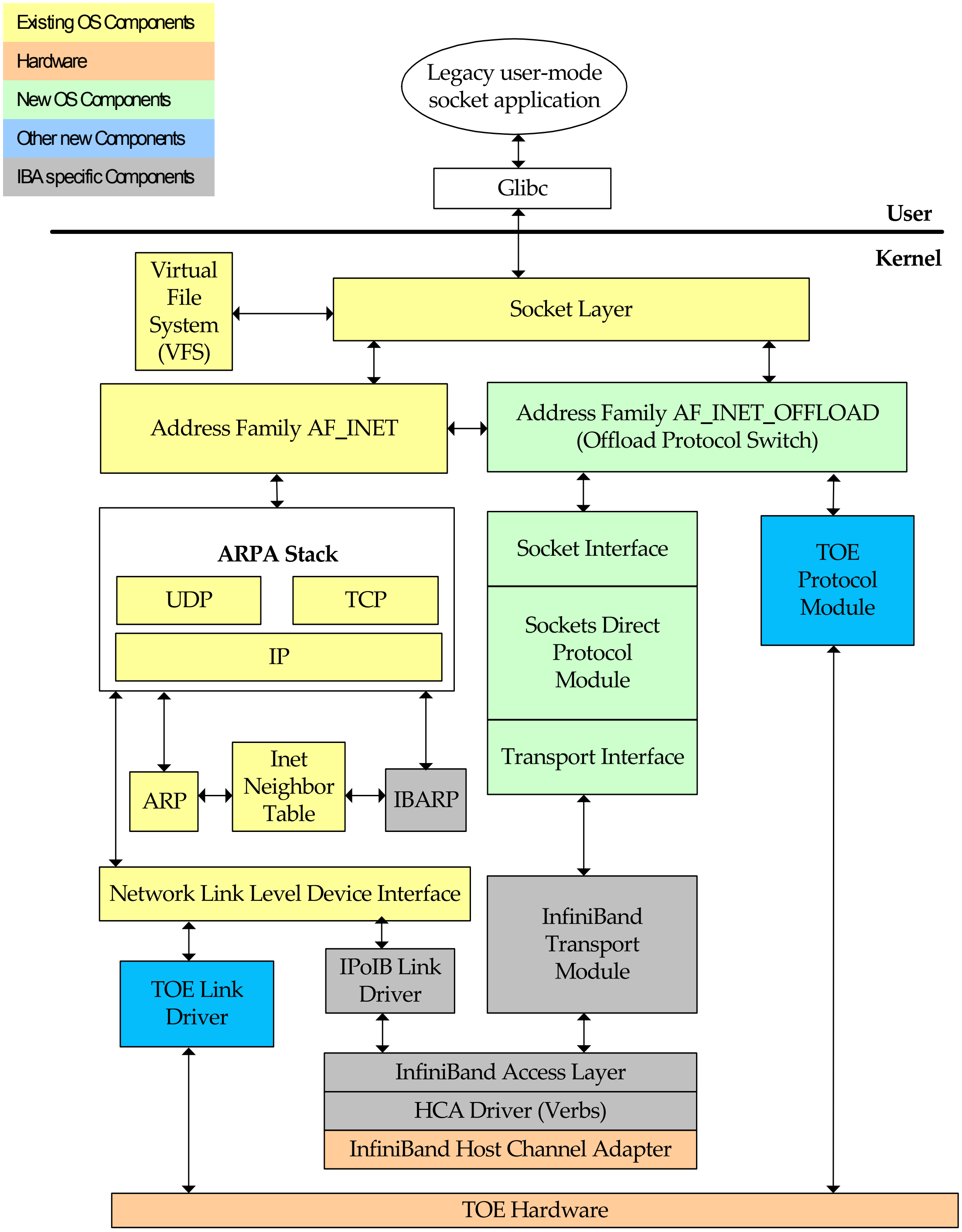


Figure 8-2: Offload Sockets framwork in the kernel

There are multiple major components to this, such as the Offload Protocol Switch Module, the SDP protocol module (which include the socket and transport Interface), and the InfiniBand Transport module.

8.3.5 Component Details:

8.3.5.1 Offload Protocol Switch

Component Type	**Offload Protocol Switch**: New kernel module supporting AF_INET_OFFLOAD.
Purpose	The OPS module exposes a new address family (AF_INET_OFFLOAD) and preserves the protocol operation semantics with the kernel socket driver at the top while providing a new offload sock structure, interface, and binding to offload protocols and transports underneath to support hardware devices capable of offloading all reliable transport features.
Functionality	The OPS module provides the standard socket call interface for AF_INET_OFFLOAD and switches socket traffic across multiple protocol providers underneath. Provides new binding interface for new offload protocol modules to register network interface information along with a new offload proto operations. Attempts to switch sock_stream connections to offload protocols and falls back to standard stack if failures occur. Switches to standard stack at the AF_INET interface level requiring no changes to the AF_INET stack.
Externally resources required	Offload protocol and transport modules.
External interfaces	All external interfaces to AF_INET are defined by the Linux socket calls in *include/linux/net.h and include/linux/sock.h.* There is a new offload protocol structure and interface used for offload protocols that register with the AF_INET_OFFLOAD address family.
Internal interfaces	The OPS module is a thin veneer between the socket driver and the offload and non-offload INET protocols. Internal interfaces will be defined to provide routing information for the offload protocol modules and transport interfaces.
Dependencies and Inter-relationships with other components	The OPS is dependent on standard socket driver interface at the top and a new offload protocol structure and interface at the bottom.
Requirements and constraints	The OPS must be built to support a completely different address family while at the same time be designed to support switching across the standard AF_INET address family. This will enable Linux distributors the freedom to decide whether or not to support legacy IP socket applications under AF_INET or simply support offloading only under the new AF_INET_OFFLOAD address family. Modifications to the Linux kernel for transparent AF_INET support must be kept to a minimum.
Development Verification/Validation/Analysis/Test strategies	Standard socket applications test suites.

8.3.5.2 Offload Protocol Module - Sockets Direct

Component Type	**Offload Protocol Module – Socket Direct Protocol**: A new loadable Linux kernel module.
Purpose	The OP module provides standard socket session management across reliable transports. SDP is a standard wire protocol that maps TCP_STREAMS to reliable networks like InfiniBand. Preserves the standard socket abstraction at the top and manages these sessions across a new offload transport interface at the bottom.
Functionality	The OP module provides standard socket semantics via protocol operations exported up to OPS. Manages sockets across multiple transport providers underneath. Supports new binding interface for new offload transport modules and the new binding to OPS. Attempts to switch sock_stream connections to offload transports beneath.
Externally resources required	OPS (AF_INET_OFFLOAD address family) module and the IB Transport Module.
External interfaces	All external interfaces are defined by the new offload protocol structure and interface at the top and the new offload transport operations for the bottom half.
Internal interfaces	The OP module includes a socket/session management at the top, a Sockets Direct wire protocol engine in the middle and a transport management and interface at the bottom that is capable of supporting multiple transport modules.
Dependencies and Inter-relationships with other major components	The OP is dependent on OPS definitions at the top, SDP specification at the middleware layer, and the OTI definitions at the bottom. All of these are new interfaces with the exception of the *proto_ops* that is already defined.
Requirements and constraints	The OP must preserve the socket semantics at the top and support SDP specification as defined in the IBTA. Required to support InfiniBand transports.
Development Verification/Validation/Analysis/Test strategies	Standard socket applications test suites.

8.3.5.3 Offload Transport Module - InfiniBand Transport

Component Type	**Offload Transport Module**: A new loadable Linux kernel module.
Purpose	Provides abstraction of the underlying IBA Access Interface.
Functionality	Maps IBA access interfaces to standard OTI operations for socket based offload protocol modules. Transport services exported include: transport registration, IP to IB name services, IB connection services, memory mapping, and RDMA and data transfer.
Externally resources required	The IB-OT requires OTI definitions and registration mechanisms and InfiniBand Access Layer.
Externally visible attributes	OTI defined operations.
External interfaces	OTI defined operations and interface mechanism.
Internal interfaces	Interface with IPoIB device driver for address resolution.
Dependencies and Inter-relationships with other major components	The OTI dependent on new definitions of OTI operations interface and structure
Requirements and constraints	Requirements of these transports include memory registration, memory window binding, message sends and receives, RDMA write and reads, connect request/accept/reject/listen, and more (TBD)
Development Verification/Validation/ Analysis/Test strategies	IB_AT developer unit tests. Standard socket applications test suites.

8.3.6 Product Deliverables

8.3.6.1 Programmatic Interfaces (APIs)

Component Name	Brief Description
Offload Transport Interface.	Kernel mode offload transport interface to register and bind offload devices.

8.3.6.2 Product Documentation

Component Name	Brief Description
Linux SDP and IB Transport HLD	A document that describes the high level design and defines the APIs for Intel's implementation of a Linux SDP driver.
Linux Offload Socket Framework HLD	A document that describes the sockets framework including OPS and OTI.
Linux SAS: Offload Sockets Framework and SDP section	A section of Software Architecture Specification describing the high-level block diagram. Includes requirements and dependencies on all other modules.

8.4 Theory of Operation

8.4.1 Socket Driver and Offload Protocol Switch Module Interaction

The Offload Protocol Switch (OPS) module provides a dynamic binding facility for offload protocols modules. It is a loadable module that registers dynamically with the Linux socket driver (socket.c). It exposes a new address family (AF_INET_OFFLOAD) but at the same time interacts with the AF_INET address family so that IPPROTO_ANY traffic can be directed to both the offload protocols under AF_INET_OFFLOAD and to the standard AF_INET protocols. Applications can also directly create sockets on AF_INET_OFFLOAD address family and bind to any offload protocols registered with this offload address family, without requiring any kernel patches.

In order to seamlessly support sockets created by applications with AF_INET address family a small patch must be applied to the socket driver (socket.c) sock_create() code to direct AF_INET address traffic to the offload address family module (AF_INET_OFFLOAD) exposed by the OPS module. The OPS module will switch sockets appropriately based on family, protocol type, and protocol. No modifications are needed in the AF_INET stack since the OPS module will interact with the standard AF_INET stack via the address family socket interface. Figure 3 shows the original Linux socket driver address family switching logic and the changes in the proposed kernel patch.

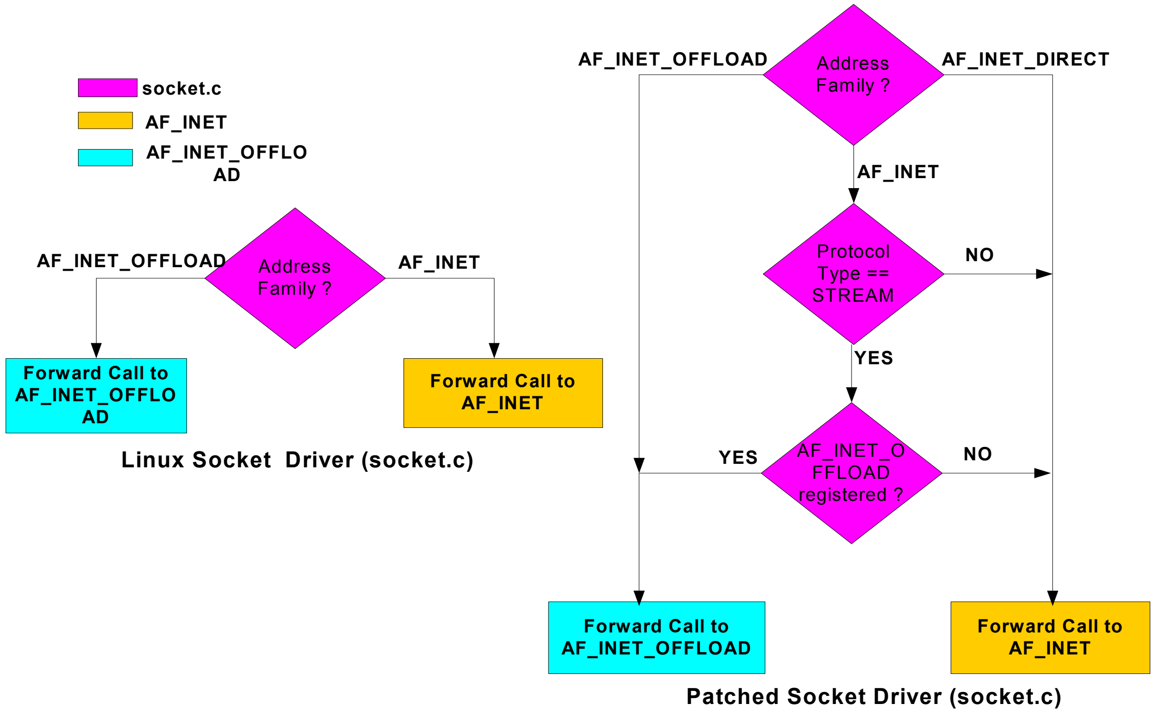


Figure 6-3: Proposed Patch to Linux Socket Driver

Figure 4 shows a high level overview of the protocol switching logic during socket creation by the offload protocol switch implemented under the AF_INET_OFFLOAD address family.

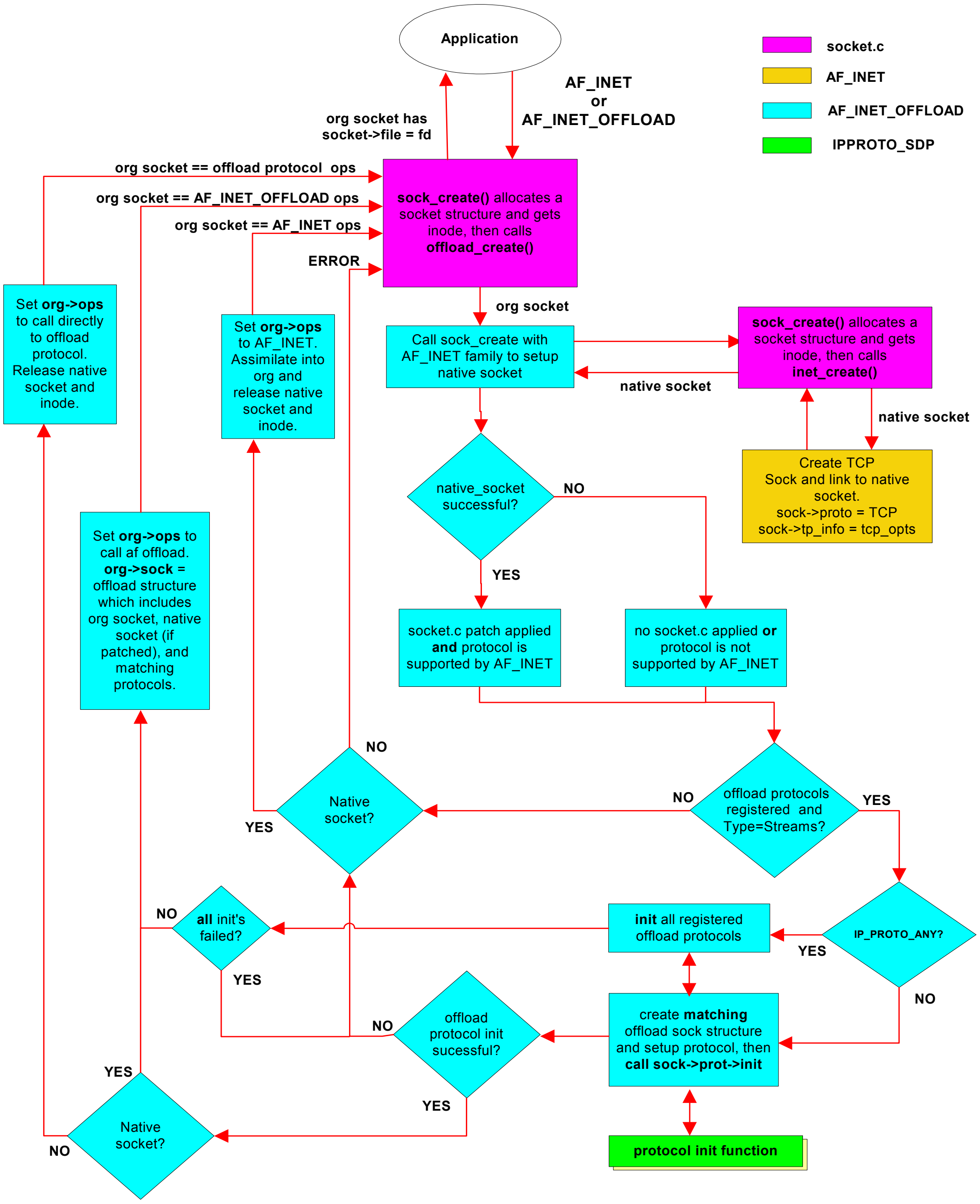

Application
AF_INET
or
AF_INET_OFFLOAD
org socket has
socket->file = fd
socket.c
AF_INET
AF_INET_OFFLOAD
IPPROTO_SDP
org socket == offload protocol ops
org socket == AF_INET_OFFLOAD ops
org socket == AF_INET ops
ERROR
sock_create() allocates a socket structure and gets inode, then calls offload_create()
org socket
Call sock_create with AF_INET family to setup native socket
sock_create() allocates a socket structure and gets inode, then calls inet_create()
native socket
native socket
Create TCP Sock and link to native socket.
sock->proto = TCP
sock->tp_info = tcp_opts
Set org->ops to call directly to offload protocol. Release native socket and inode.
Set org->ops to AF_INET. Assimilate into org and release native socket and inode.
native_socket successful?
NO
YES
socket.c patch applied and protocol is supported by AF_INET
no socket.c applied or protocol is not supported by AF_INET
Set org->ops to call af offload. org->sock = offload structure which includes org socket, native socket (if patched), and matching protocols.
NO
Native socket?
YES
NO
offload protocols registered and Type=Streams?
YES
NO
all init's failed?
YES
init all registered offload protocols
YES
IP_PROTO_ANY?
NO
NO
offload protocol init sucessful?
create matching offload sock structure and setup protocol, then call sock->prot->init
YES
YES
Native socket?
NO
protocol init function

Figure 8-4: Protocol Switching Logic in AF_INET_OFFLOAD for Socket Creation

8.4.2 Offload Protocol Switch and SDP Module Interaction

Figure shows the various steps involved in the OPS and offload protocol modules initialization and operation

1. The socket.c sock_create() code is modified to forward all AF_INET sock_create calls to AF_INET_OFFLOAD module. The AF_INET_OFFLOAD module will direct the create call based on the offload protocols that have registered. If the create is forwarded back to

the AF_INET, the OPS will use AF_INET_DIRECT so that the sock_create() code will know to switch the create directly to the AF_INET path thus sending all subsequent socket calls directly to the correct family.

2. The AF_INET_OFFLOAD module, as part of its initialization, registers its address family *AF_INET_OFFLOAD* to the socket layer by calling *sock_register ()* call back to the socket layer. All future socket calls, with address family *AF_INET_OFFLOAD* will be directed by the socket layer to the OPS layer.
3. When Offload Protocol modules get loaded, the *ops_proto_register ()* is called to register their entry points and capabilities like self-routing, rdma etc to the AF_INET_OFFLOAD module.
4. When a network interface configuration changes (address assignment, address change, route changes, etc.), the Offload protocol module which is bound to this network interface notifies the APS module by calling o*ps_inet_notification()*.
5. Depending on the incoming request, the OPS module will then switch to proper protocol module. The switching policy depends on the protocol capabilities, binding priority set by the user. The AF_INET stack is the default stack, and if none of the offload protocol modules are loaded or if none of the module capabilities matches the incoming request, the OPS will forward the request to the AF_INET stack. For protocol modules that do not support self-routing, the AF_INET_OFFLOAD driver will handle the routing issues.

Once an appropriate protocol module is chosen, it is up to the protocol stack to handle the request. For example if SDP protocol module is chosen to service an request, then it is up to the SDP module to establish a session with its peer, pre register buffer element and decide on mode of data transfer (Send vs. RDMA Write for example).

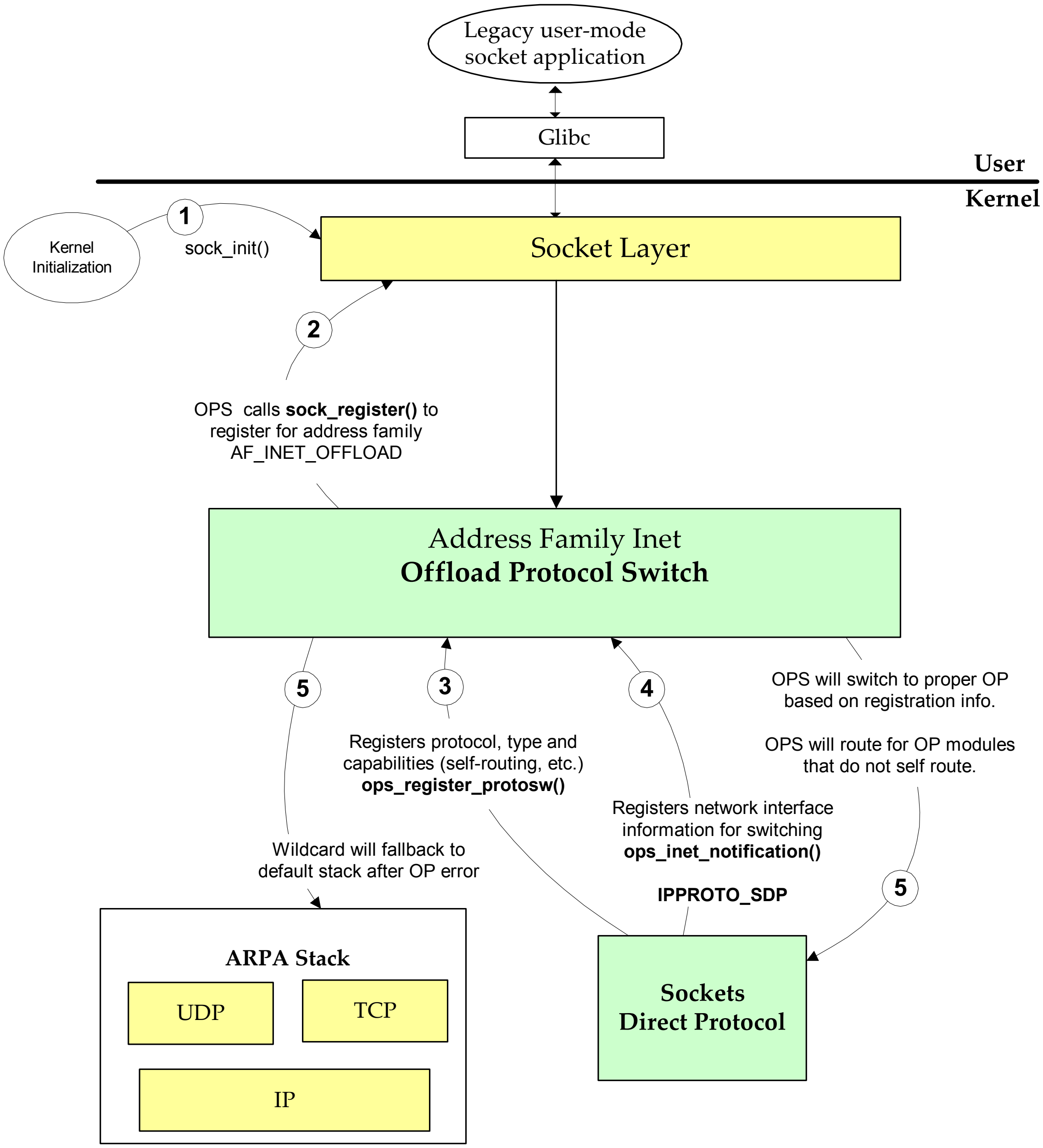

Legacy user-mode socket application
Glibc
User
Kernel
Kernel Initialization
1
sock_init()
Socket Layer
2
OPS calls sock_register() to register for address family AF_INET_OFFLOAD
Address Family Inet
Offload Protocol Switch
5
3
4
OPS will switch to proper OP based on registration info.
OPS will route for OP modules that do not self route.
Registers protocol, type and capabilities (self-routing, etc.) ops_register_protosw()
Registers network interface information for switching ops_inet_notification()
Wildcard will fallback to default stack after OP error
IPPROTO_SDP
5
ARPA Stack
UDP
TCP
IP
Sockets Direct Protocol

Figure 8-5: OPS and SDP interaction

8.4.3 SDP and InfiniBand Transport Interaction

Offload Transports (OTs) implement the offload transport interface abstractions. The InfiniBand transport is designed to be a kernel loadable module, which provides an InfiniBand hardware transport specific implementation of the offload transport abstraction defined by the offload transport interface.

Figure shows the various steps involved in the OT module initialization and operation.

1. The InfiniBand (IB) Transport module will register with the SDP module and provide the transport operations interface and capabilities.
2. The InfiniBand Transport module obtains kernel notification services for network devices, IP configuration, and IP routing information using the device name provided by the link driver. When an IP configuration notification occurs for this net device the transport module will forward the via the event notification upcall to SDP, if SDP has registered for events.
3. The SDP module, using the transport register event call, will register for address, net device, and route changes.
4. The SDP module maintains a list of transports, with address information, and will switch to appropriate transport based on address information during a socket bind.

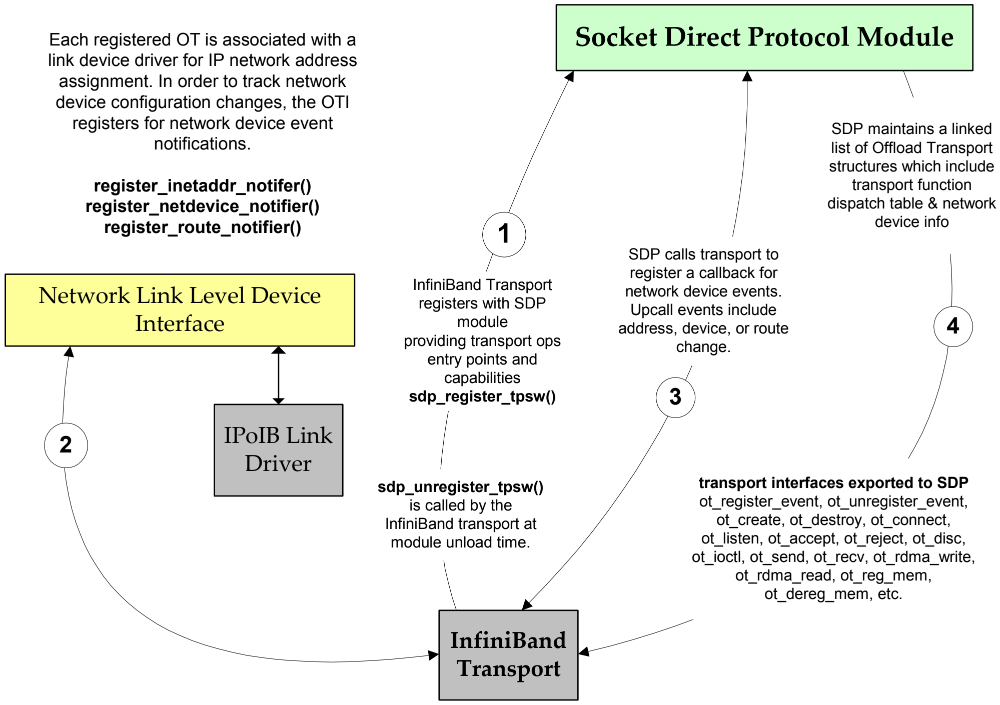

Socket Direct Protocol Module
Each registered OT is associated with a link device driver for IP network address assignment. In order to track network device configuration changes, the OTI registers for network device event notifications.
register_inetaddr_notifer()
register_netdevice_notifier()
register_route_notifier()
SDP maintains a linked list of Offload Transport structures which include transport function dispatch table & network device info
1
InfiniBand Transport registers with SDP module providing transport ops entry points and capabilities
sdp_register_tpsw()
SDP calls transport to register a callback for network device events. Upcall events include address, device, or route change.
Network Link Level Device Interface
4
3
2
IPoIB Link Driver
sdp_unregister_tpsw()
is called by the InfiniBand transport at module unload time.
transport interfaces exported to SDP
ot_register_event, ot_unregister_event, ot_create, ot_destroy, ot_connect, ot_listen, ot_accept, ot_reject, ot_disc, ot_ioctl, ot_send, ot_recv, ot_rdma_write, ot_rdma_read, ot_reg_mem, ot_dereg_mem, etc.
InfiniBand Transport

Figure 8-6: SDP and InfiniBand Transport

9. Architectural Specification – uDAPL

9.1 Introduction

The direct access transport (DAT) over an IBA software stack defines a new I/O communication mechanism. This mechanism moves away from the current local I/O model based on attached transactions across buses to a remote, attached, message-passing model across channels.

uDAPL is a System Area Network (SAN) provider that enables an application to bypass the standard TCP/IP provider and use the native transport to communicate between hosts in a cluster of servers and workstations on the fabric. This also enables the applications to take advantage of the underlying transport service provided by InfiniBand Architecture to permit direct I/O between the user mode processes.

The primary responsibility of the uDAPL are transport independent connection management, transport independent low latency data transfer and completion

UDAPL is intended to support following type of application:

- Heterogeneous Clusters/Databases
- Homogeneous Clusters/Databases
- Message Passing Interface (MPI)

9.2 Requirement for uDAPL

The detailed requirement for uDAPL is available at DAT consortium. However here are the key requirements

- Provide transport API mechanism to work with IB , iWARP etc
- Provide/use transport independent Name Service
- Provide transport independent Client/Server and Peer-to-Peer connection management
- Provide mechanism for zero copy model

9.3 System Structural Overview

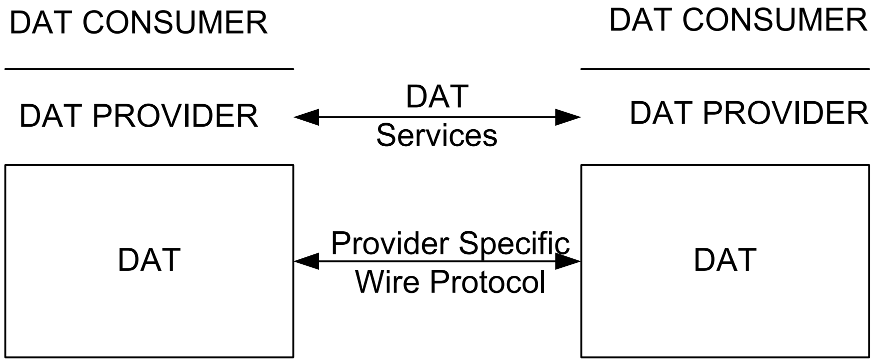


Figure 9-1

The Direct Access Transport (DAT) Model is shown above. There are two significant external interfaces to a Direct Access Transport service provider. One interface defines the boundary between the consumer of a set of local transport services and the provider of these services. In the DAT model, this would be the interface between the DAT Consumer and the uDAPL Provider. The other interface defines the set of interactions between local and remote transport providers that enables the local and remote providers to offer a set of transport services between the local and remote transport consumers. In the DAT model, this would be the set of interactions between a local uDAPL Provider and a remote uDAPL Provider that are visible to the local DAT Consumer and/or remote DAT Consumer.

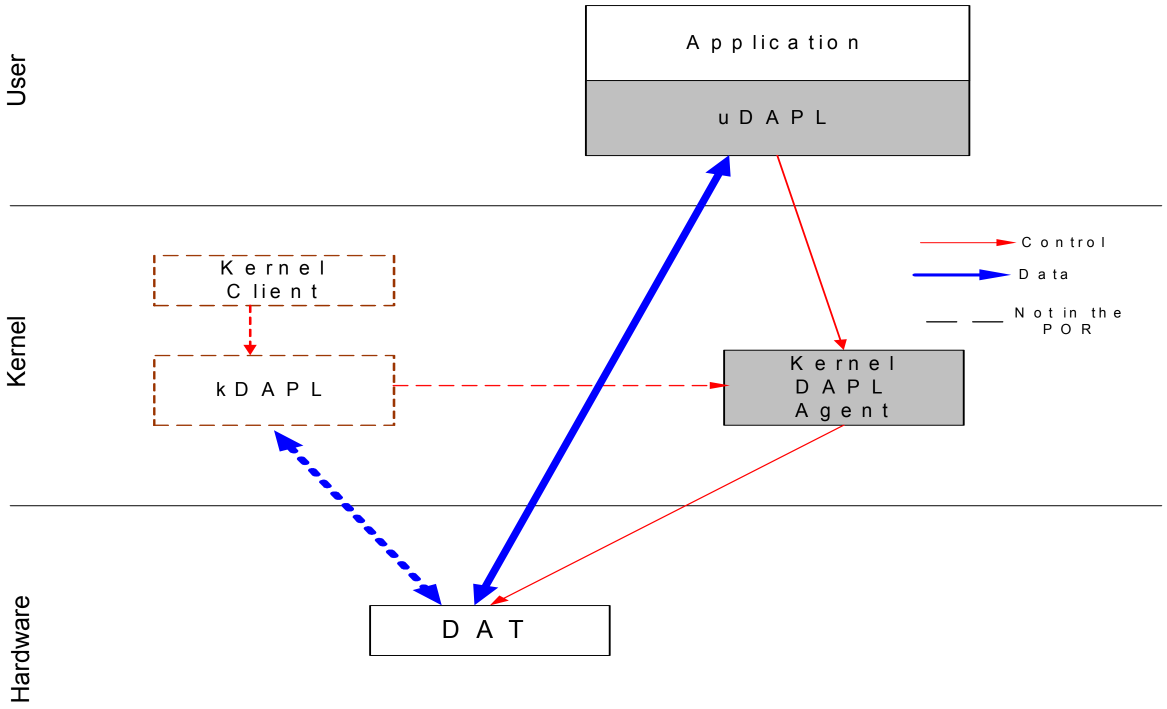
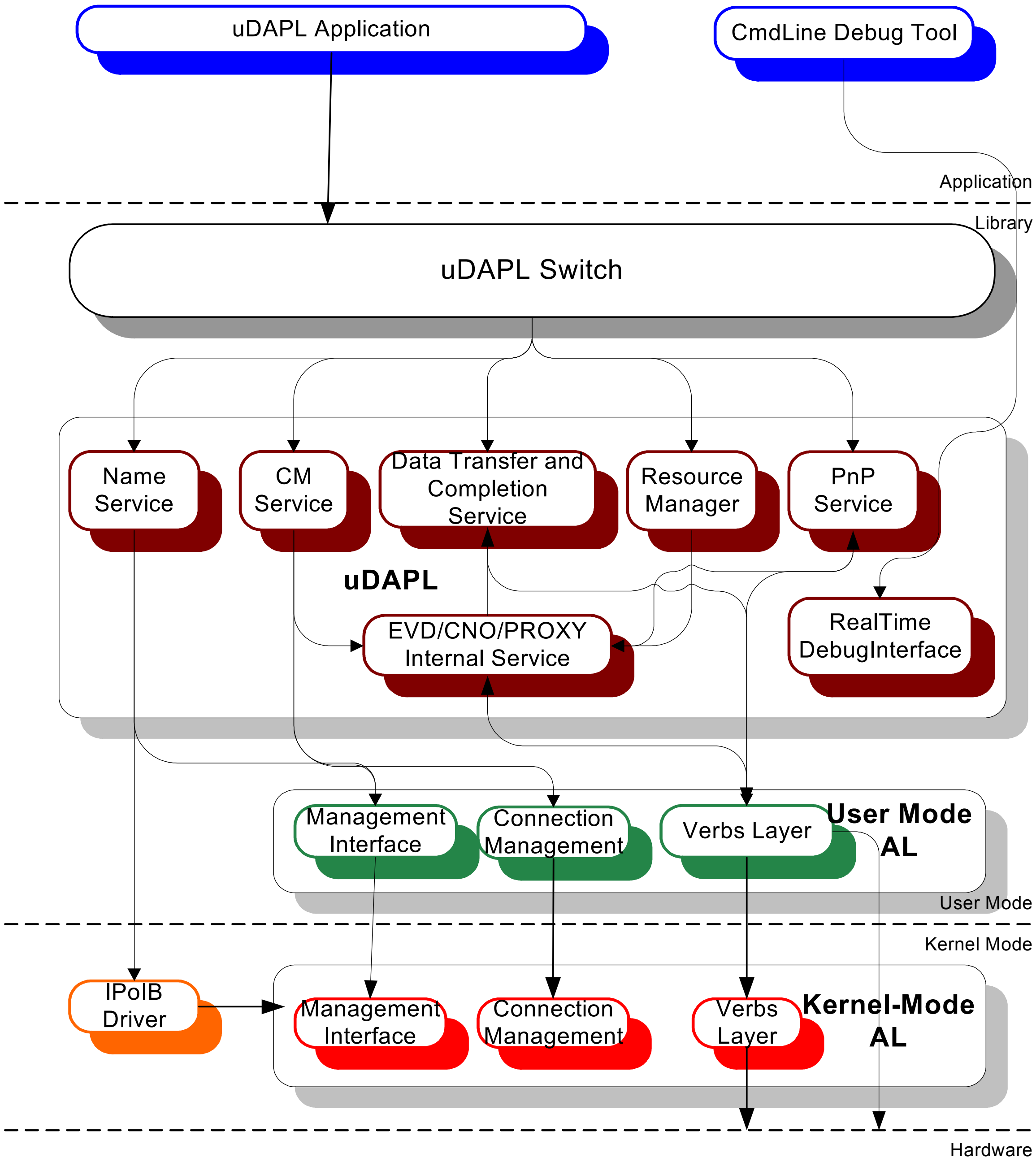


Figure 9-2

Above figure illustrates generic uDAPL implementation and interaction. uDAPL is a dynamic shared library and provide user mode APIs. This Library interacts with uDAPL kernel Agent for any resource management but does data transfer and transfer completion in user mode without taking a kernel transition to provide very low latency. DAT can use any physical hardware that can provide the required characteristics. DAT is specified to support features that are subset IB Channel Adapters and hence DAT can be directly mapped to it. Pictorial of Component Structure



The diagram above illustrates the major uDAPL components and various interfaces to the components such as the uDAPL switch, Access layer and IPoIB driver.

uDAPL switch enables Interface Adapter enumeration , name service and PnP capabilities in provider independent manner. Exact details of this component is yet to be decided

uDAPL has well defined API s that applications can use to create Event Dispatcher, Endpoint, connections etc. Also uDAPL has private protocol interface through PIPE and acts as command line interface debug tool.

uDAPL interfaces with the Access Layer for all InfiniBand specific operation and maps the InfiniBand operation to DAT operation.

Also uDAPL uses the IP addressing scheme to establish IB connection. It uses the IPoIB driver and Management features in the Access Layer for converting IP address to IB address and IB path records

Also uDAPL provides PnP functionality based on addition or removal of IP address. It receives IP address change notification from IPoIB driver. Asynchronous notification from Access layer is also used by the PnP mechanism

9.3.1 Component Details

Component Type	**uDAPL**: A loadable Linux user mode component that provides API as defined by DAT collaborative.
Purpose	The uDAPL provides a standard interface between the uDAPL consumer and Direct Access Transport (IB) in transport independent manner.
Functionality	The uDAPL implements the Name service, Connection Manager, Data transfer/completions and maps them to InfiniBand counterpart.
Externally resources required	The uDAPL utilizes HCA resources such as RC queues, CQs, Protection Domains, TPT etc.
Externally visible attributes	All visible attributes are defined by uDAPL specifications.
External interfaces	All external interfaces are defined by the Intel's Access Layer specification & IPoIB specification.
Internal interfaces	The uDAPL is a thin layer between the Access Layer and the uDAPL consumer. It does not make use of any significant internal subsystems.
Dependencies and Inter-relationships with other major components	The uDAPL is dependent on the HW Access Layer & IPoIB implementation.
Requirements and constraints	The uDAPL implementation must comply with the DAT Consortium's uDAPL specification and interoperate with other uDAPL (over IB) implementations.
Development Verification/Validation/Analysis/Test strategies	TBD

9.3.2 List of External Interfaces

9.3.2.1 OS Interfaces

There are a number of OS services that uDAPL is dependent upon to export the proper semantics. For example, the ability to pin/unpin memory is an OS function. Other examples are the OS wait proxy for waiting on heterogeneous events and OS shared memory support to allow shared mappings in the hardware. However some of these features are abstracted either by Access Layer or Component Library. At the time of writing this document, uDAPL by itself does not have any special OS requirement.

9.3.2.2 Access Layer

uDAPL heavily depends on Access layer for accessing all HCA resources. uDAPL also depends on Access Layer to cleanup the resource when uDAPL consumer abruptly terminates.

9.3.3 List of Product Deliverables

9.3.3.1 Consumer Interface

uDAPL APIs are under development at DAT consortium. Refer to the DAT consortium for latest specification.

9.3.3.2 Debug Interface

uDAPL provides debug interface through PIPE/shared memory (yet to be decided). Intended debug Commands are

- Change debug Level
- Reset API usage Counter
- Log API usage counter
- Reset API time stamp histogram
- Log API time stamp histogram
- Reset SEND/RDMA/RECV time stamp histogram
- Log SEND/RDMA/RECV time stamp histogram

9.3.3.3 Dynamo with uDAPL Support

Dynamo/Iometer with uDAPL support is needed for uDAPL performance analysis

9.3.3.4 Product Documentation

High Level Design document for uDAPL will be written to provide information about implementation, configuration and optimization

9.4 Theory of Operation

uDAPL maps IB resources, features and usage model into DAT resources, features and usage model. uDAPL is expected to be a thin layer doing this IB mapping and providing the required feature.

The Following uDAPL features needs to be mapped to IB features

- Interface Adapter
- End Point
- Name Service
- Connection Qualifier
- Connections
- Local Memory Region & Shared Memory regions
- Event Dispatchers
- PnP Mechanism

9.4.1 Interface Adapter Creation and Management

uDAPL instance exposes each mapped to IB partition on Active IB Port (i.e., IP address) as an DAT Interface Adapter. uDAPL relies on IPoIB notification mechanism for arrival/change/leaving of IP address. Based on this notification uDAPL maintains the Interface Adapter (IA) information that can be created/closed by the consumer.

9.4.2 Endpoint

uDAPL Endpoints are dynamically mapped to IB QP. Since it is mapped dynamically, End point creation and Endpoint close may not be synchronous with QP creation and destructions. Since DAT doesn't define TIME-WAIT state for Endpoint, uDAPL must hide this QP state internally.

9.4.3 DAT Connection and Handoff

uDAPL exposes IB RC service through DAT. Also it uses IP address for IB connection. It uses IPoIB driver ioctl interface and Access Layer's Management APIs to convert IP address into an IB address and path records that are needed for establishing IB connection.

uDAPL supports Connection Handoff. The details of which are left to the HLD.

9.4.4 Mapping DAPL Connection Qualifier

uDAPL maps Connection Qualifier into IB service ID. Exact mapping scheme is yet to be decided..

9.4.5 Memory Management

Both memory regions and memory window of IB architecture are exposed as Local Memory region (LMR) and Remote memory region (RMR) in uDAPL. In addition uDAPL also supports shared memory registration across process. To support this uDAPL uses Access Layer's special API that provides Key based registration. The intention of this feature is to reduce the TPT usage. These shared memory registration are supported only with in the native host channel adapter. Proposed Intel's NS provides API that can group IA that are belong to the same host channel adapter.

9.4.6 Event Dispatchers

uDAPL Event Dispatchers (EVD) are implemented over IB CQs and software dispatcher. The goal of the EVD implementation is not to proxy the CQ but to map it directly for minimum latency.

10. Debugging, Error Logging, and Performance Monitoring

10.1 Instrumentation

10.1.1 Performance Counters

The component library provides performance counter functions to allow timing the execution of functions. Use of the performance counters does not disable thread scheduling, and time stamps can include time spent while pre-empted. Never the less, the performance counters are useful in determining the effect of performance of various changes. The performance counters can be used to time execution within as well as between functions. The performance counter provider serializes accesses to each counter on a per-counter basis, which can impact system performance. Performance counters should be enabled only when gathering performance statistics. Because the performance counters are compiled in, there is no way of enabling or disabling them at runtime.

10.1.2 Trace Buffers

The component library will provide trace buffer capabilities. The trace buffers are intended to provide a history of execution rather than performance information. This history can be crucial in determining the cause of deadlocks or other errant behavior. Multiple threads can access the trace buffers simultaneously and contention for trace buffer entries is controlled via interlocked variables. This makes the trace buffers light weight enough to leave enabled in externally released builds and allows traces to be captured in the field for difficult to reproduce situations. A trace buffer entry includes a user-defined keyword and timestamp, allowing the control flow of software to be reconstructed based on analysis of the trace buffer, even in multi-processor systems.

10.2 Debug

The component library supports debugging by providing functions to generate debug output, assert that parameters are correct, halt the system in cases of fatal errors, and halt execution for debugging purposes.

10.2.1 Debug Output

Debug output generation negatively impacts performance, and should be disabled to the in non-debug builds. In debug builds, software should allow control to minimize output to only that in which the people debugging are interested.

10.2.1.1 cl_dbg_out and cl_msg_out

The component library provides two functions to generate debug output: cl_dbg_out and cl_msg_out. These two functions are identical except that cl_dbg_out only generates output in debug builds. Both functions follow the same syntax as the standard C library printf function.

10.2.1.2 Debug Macros

The component library provides macros for commonly needed functionality relating to debug output generation in debug builds. These macros allow controlling the amount of debug output (CL_PRINT), generating function entry (CL_ENTER) and exit (CL_EXIT) messages, and generating function trace messages (CL_TRACE and CL_TRACE_EXIT). Function entry, exit, and trace functions include in the output the name of the module and function generating the output. For details regarding the use of these macros, see the component library documentation relating to CL_PRINT, CL_ENTER, CL_EXIT, CL_TRACE, and CL_TRACE_EXIT.

All macros provided by the component library to generate debug output are automatically disabled in non-debug builds. The debug macros only generate output if all bits specified by the message's debug level are set in the module debug level, or if the message is an error message. Modules can have any number of debug levels, allowing runtime debug level adjustments relating to specific areas to be enabled independently from others. Users are encouraged to allow flexibility in controlling debug output such that the output can be tailored to the situation being debugged.

10.2.2 ASSERT

In a kernel build, the ASSERT macro will halt the system. In user mode, the application aborts execution following the assertion. Never the less, developers should assert that parameters are valid unless they provide runtime parameter checking. The cost of a system panic should pale in light of the savings in debug investigations necessary to find bugs. The ASSERT macro has no effect in non-debug builds.

10.2.3 cl_panic

The component library provides the cl_panic function to allow kernel mode developers to cause the system to halt when a fatal error is detected. In user mode, cl_panic is equivalent to the standard C library function abort(). In the kernel, calling cl_panic allows preserving the memory and system state once an error is detected but before the system crashes. Unlike the ASSERT macro, cl_panic works in both debug and non-debug builds.

10.2.4 cl_break

The component library provides the cl_break function to allow execution to stop when a debugger is attached. Unlike cl_panic, execution can be resumed after a call to cl_break.

10.3 Error Logging

The component library provides event-logging capabilities, which should be used by all software wishing to log events. The log provider allows three different log message types, informational, warning, and error. Log messages are directed to the system log file, and event logging is available in both the kernel and user mode. Refer to the documentation for the cl_log function for detail about the use of the function and its parameters.