OpenRDMA Software Architecture

The OpenRDMA Project, Editor: Bernard Metzler

Version1.0

December 21, 2004

Abstract

This document provides an architectural overview on the OpenRDMA Linux stack. It currently gives no implementation details. The OpenRDMA Project will update this architecture as the group reaches new levels of agreement.

Contents

1 Goals

The OpenRDMA Group agreed upon the following goals, which must be met by the OpenRDMA software acchitecture:

1. OpenSource
   (a) All IHV independent code is completely open source.
   (b) The project is open to all contributors and all contributors (not companies) have equal rights.
   (c) The OpenRDMA code will be licenced under a dual GPL and MIT license.
2. Programming Interfaces and offered Services
   (a) As far as applicable, the architecture follows open standards for its interfaces
   (b) The RNIC-PI as currently defined by the ICSC of the Open Group will be used to interface with IHV private RNIC components.
   (c) RDMA services will be available via the IT-API and DAPL API's.
   (d) The architecture does not preclude the development of SDP components to provide RDMA services for socket applications. Such components would preferable be designed as consumers of native RDMA services provided via IT-API or DAPL.
   (e) As far as interface semantic between generic components and IHV specific code is not part of any open interface specification, the OpenRDMA Group will establish appropriate interface descriptions each IHV coding for OpenRDMA must follow. This may include, for example, a generic interface to TCP context migration, IP routing and MAC address resolution, if not defined by the RNIC-PI.
3. Supported RDMA Transports
   (a) The achitecture is open to support RDMA services over both iWARP based and InfiniBand based transports.
4. RNIC Vendor Independence
   (a) Using standard and open interfaces, the architecture integrates RNIC's from all IHV's that implement drivers for these interfaces.
   (b) RNIC HW/SW of different IHV's can be used concurrently.
5. Linux OS Integration
   (a) Acceptance by Linux Kernel maintainers
   (b) Modularization: The architecture consists of components strictly seperating between RNIC vendor specific and generic code.
   (c) All modules of the OpenRDMA architecture are demand loadable.
   (d) The OpenRDMA software runs as a loadable extension to the vanilla Linux kernel and without changes to the original source code.
   (e) IHV's not intending to contribute their hardware dependent source code to the project can integrate by contributing object code linkable with the defined OpenRDMA interfaces.

2 Main Building Blocks

Fig.1 summarizes the high level design of the OpenRDMA software architecture. It provides a generic infrastructure (green in Fig.1) for the integration of IHV private code (blue in Fig.1) into the Linux OS. This infrastructure offers RDMA services for both user level and kernel level RDMA applications also known as RDMA *consumers* (grey in Fig.1).

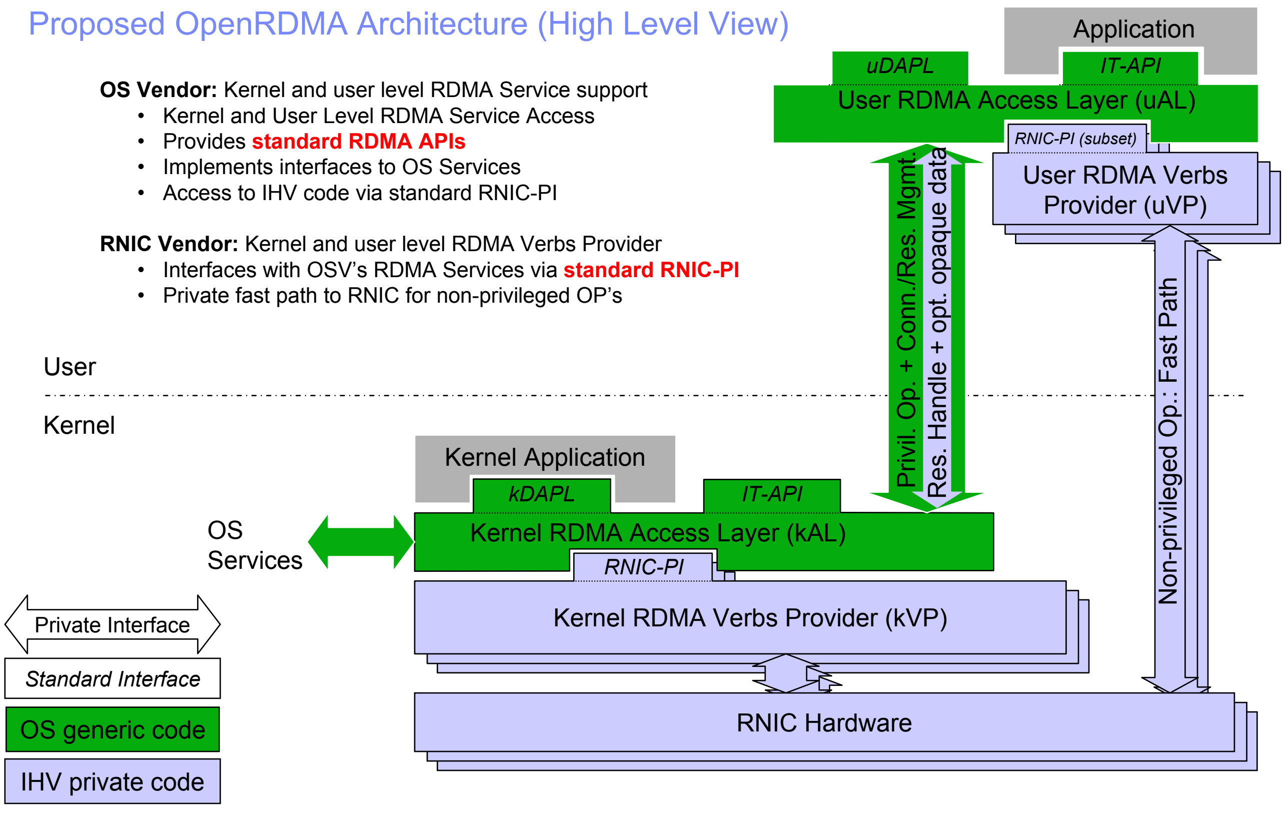

Proposed OpenRDMA Architecture (High Level View)
OS Vendor: Kernel and user level RDMA Service support
Kernel and User Level RDMA Service Access
Provides standard RDMA APIs
Implements interfaces to OS Services
Access to IHV code via standard RNIC-PI
RNIC Vendor: Kernel and user level RDMA Verbs Provider
Interfaces with OSV's RDMA Services via standard RNIC-PI
Private fast path to RNIC for non-privileged OP's
Application
uDAPL
IT-API
User RDMA Access Layer (uAL)
RNIC-PI (subset)
User RDMA Verbs Provider (uVP)
Privil. Op. + Conn./Res. Mgmt.
Res. Handle + opt. opaque data
Non-privileged Op.: Fast Path
User
Kernel
Kernel Application
kDAPL
IT-API
Kernel RDMA Access Layer (kAL)
RNIC-PI
OS Services
Kernel RDMA Verbs Provider (kVP)
RNIC Hardware
Private Interface
Standard Interface
OS generic code
IHV private code

Figure 1: General OpenRDMA Architecture

2.1 OS Generic OpenRDMA Components

The IHV independent infrastructure for RDMA support consists of a *kernel level Access Layer* (kAL) and a *user level Access Layer* (uAL). On a host system, only one instance of the kAL may exist. The uAL is a loadable library module instantiated by each user level application which is accessing OpenRDMA services. kAL and uAL's are interacting via a generic, but OS private interface. This interface is further used to support interaction between user level and kernel level software components of an IHV.

2.1.1 Kernel Access Layer

The kAL provides IHV-independent OS support for RDMA operations. It directly supports kernel-level consumers and also privileged operations of user-level consumers. The kAL supports the multiplexing of several RNICs from different vendors.

The kAL employs OS-specific interfaces with the kernel network stack to manage LLP connections and with the OS resource management. It offers generic interfaces to the uAL, to the Kernel RDMA Verbs Provider(s) and to kernel clients such as NFSeR, iSER.

While the kAL will be implemented as a single kernel module, it is expected that it will consist of the following functional blocks (see also Fig. 2):

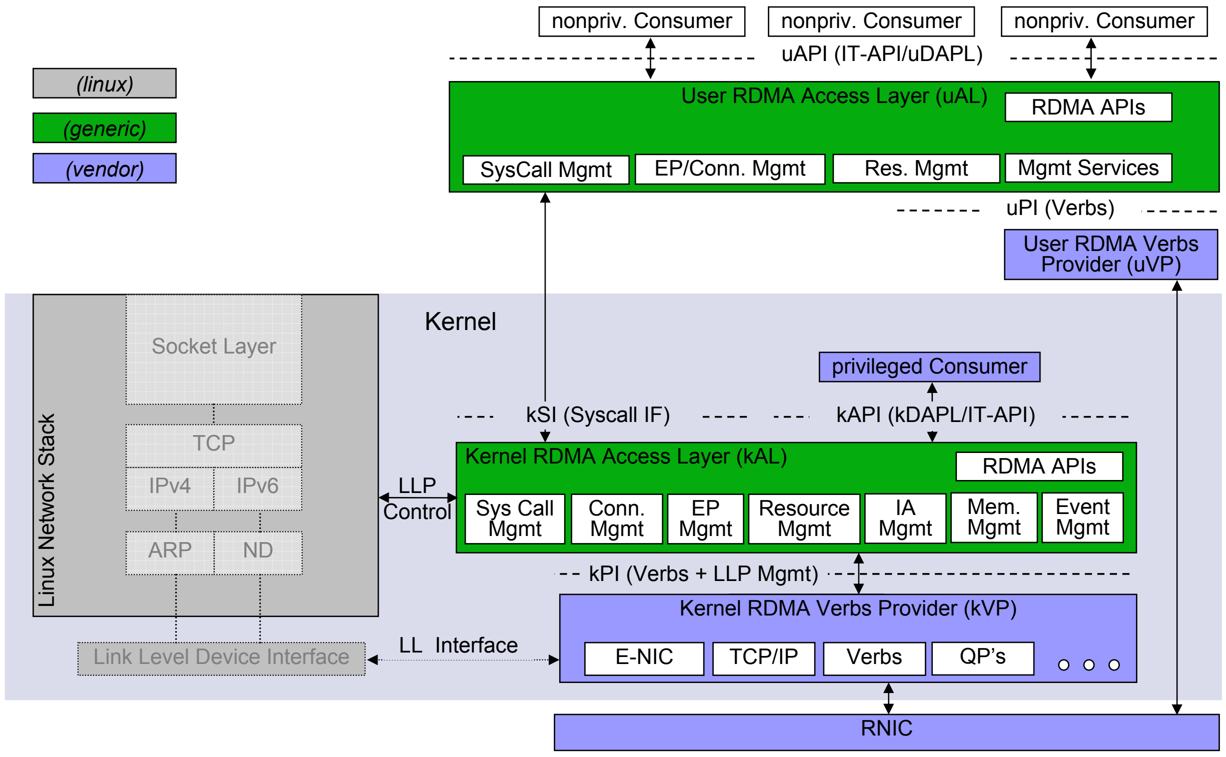


Figure 2: OpenRDMA Architecture Components

- System Call Management: implements the kAL/uAL interface, manages the execution of privileged operations requested by the uAL
- LLP Management: manages Lower Layer Protocol connections (setup, context migration and takedown for TCP/SCTP), manages coexistence of host resident LLP stack and offloaded stack (routing, link level address resolution, stack statistics for MIB/SNMP).
- RDMA-APIs: implements RDMA-APIs for kernel consumers.
- Endpoint Management: manages state of connection endpoints (EPs) and synchronizes state between EPs, LLP objects and QPs.
- Event Management: registration of Event Handlers with the kVP, event dispatching and demultiplexing for privileged consumers and uAL(s).
- Resource Management: implements functionality of *Privileged Resource Manager* as introduced in [3]. Keeps track of all active OpenRDMA resources, polices access to resources. Resources are managed hierachically to allow for robust clean-up of child resources in case of a failing parent object.
- Memory Management: exports memory pinning service as expected to be specified for the RNIC-PI. Cooperates with Resource Manager to enforce system stability and fair resource sharing.
- Interface Adapter Management: loading/unloading and attaching/detaching RNIC specific kVP, lists available RNICs upon consumers request.

2.1.2 User Access Layer

The User RDMA Access Layer (uAL) provides a functional subset of the kAL to user level consumers. While all non-privileged operations are executed in direct interaction with the User RDMA Verbs Provider, privileged operations needs for additonal intearction with the kAL and the Kernel RDMA Verbs Provider.

2.1.3 RDMA API's

The implementation of the RDMA API's is part of the OS generic infrastructure. Thus, the kAL may implement DAPL and/or IT-API to serve kernel level consumers and uAL may implement and expose these services to the non privileged consumer.

2.2 IHV Private OpenRDMA Components

It is expected that every RNIC vendor will provide a kernel-level RDMA Verbs Provider module. Optionally, it may provide a user-level RDMA Verbs Provider library for efficient user-level RDMA services. RDMA Verbs Provider modules encapsulate functions that are private to the RNIC implementation, such as QP management and conversion of work requests into WQEs.

2.2.1 Kernel RDMA Verbs Provider (kVP)

This module provides vendor-specific software implementing the semantics of the RDMA verbs as defined in [1, 2]. It translates the verb calls issued by the kAL into appropriate actions such as the creation and management of QPs and the insertion and removal of WQEs. All QP, CQ and SRQ data structures are under direct control of the module and are not accessible to the consumer. Additional code is necessary to maintain offloaded network stack control information such as the ARP table, routing table or MIB informations.

While the module's interfaces to the RNIC hardware are IHV-private, its upper interface to the kAL will implement the RNIC-PI currently being defined within the Open Group. It is expected that the RNIC-PI specification will include the definition of the interface between User RDMA Verbs Provider and Kernel RDMA Verbs Provider to support the execution of privileged operations on behalf of the non privileged user module. As depicted in Fig.1, this interface can become integrated into the uAL/kAL interface.

It should be noted that while this document discusses the functionality of a generic RNIC, the actual implementation may differ in the supported functionality, since an RNIC will typically implement only a subset of the defined RDMA transports (TCP/IP, SCTP/IP, InfiniBand). Furthermore, the actual RNIC may provide other network functionality outside the scope of OpenRDMA such as raw Ethernet for non-offloaded connections or may implement other IP based network services. Dependent on the implementation, this functionality may be part of the software module which implements the kVP or be kept in another module.

2.2.2 User RDMA Verbs Provider (uVP)

This library contains vendor-specific software for the provision of user-accessible RDMA Verbs services to the uAL and for direct access to the RNIC hardware. It establishes a fast path to the RNIC hardware for all performance critical operations that can be performed without holding exclusive access permissions. This includes all send and receive type operations. As with the Kernel RDMA Verbs Provider, all QP, CQ and SRQ data structures are under direct control of the library and are not accessible to the consumer.

As introduced for the kVP, the RNIC-PI interface with the uAL will also support the interaction with the kVP for private RNIC resource management, such as the mapping of doorbell registers and the creation of QPs.

3 Interfaces

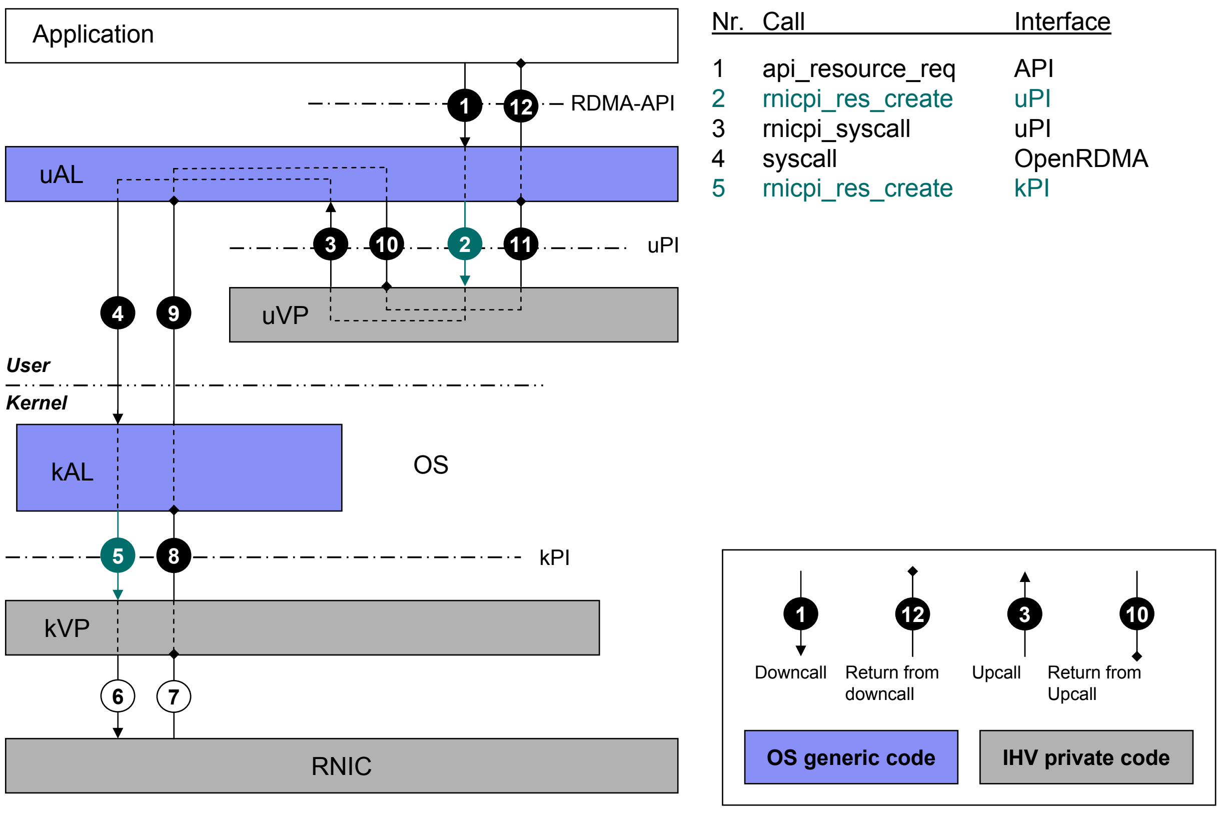
This section gives an overview on the module interfaces to be implemented for the OpenRDMA architecture. The open standard interfaces intended to be used for the projet are still in the specification phase: the RNIC-PI for interfacing the IHV supplied RNIC driver code from the uAL and kAL and the RDMA capable APIs for exposing iWARP based RDMA services to the consumers (IT-APIv2, u/kDAPL). For the RNIC-PI, a first version of the specification is expected to be available at the end of the year or in early 2005. A version of the IT-API including iWARP specifics (IT-APIv2) is also expected to be available early 2005.

3.1 The RNIC Programming Interface (RNIC-PI)

The RNIC-PI will be used to integrate the IHV supplied user space (uVP) and kernel space (kVP) modules for RNIC system integration. The RNIC-PI, following the RDMA Verbs specification [1] and the InfiniBand Verbs specifiaction including its extensions [2], defines the semantics for both control and data type interactions between the consumer of RDMA services and the RNIC. Besides of this semantics defined in the Verbs, it is likely that the RNIC-PI will make the following extensions:

- Segregation of memory registration from pinning & translation. Pinning is specified as an OS service, accessed via an upcall from the kVP. Following the OpenRDMA architecture, this call would be provided as part of the kAL.

- Full specification of the interface of the uVP to access support for privileged operations. In OpenRDMA, this service is implementable as an upcall from the uVP into the uAL.



Nr.	Call	Interface
1	api_resource_req	API
2	rnicpi_res_create	uPI
3	rnicpi_syscall	uPI
4	syscall	OpenRDMA
5	rnicpi_res_create	kPI

Figure 3: Call Sequence for privileged Operation from User Space

3.1.1 Privileged Mode

The kVP provides the IHV specific implementation of the privileged mode RNIC-PI to the kAL. This interface directly supports all operations of a privileged consumer. Within OpenRDMA, a privileged consumer of RDMA services will reside in the OS kernel (see Fig. 1). Typically, it will be a loadable service module like an iSER or NFS client. Besides of the support of kernel based comsumers, this interface is also used to execute privileged control type operations of a non-privileged consumer which cannot be performed solely in user space.

3.1.2 Non privileged Mode

The uVP provides the IHV specific implementation of the non-privileged mode RNIC-PI to the uAL. This interface directly supports all fast path operations of a non privileged consumer. These operations are typically completed in local interaction of uAL, uVP and IHV hardware.

The RNIC-PI Syscall Extension The original Verbs document [1] does not specify an interface between the uVP and its corresponding kVP for the support of privileged calls. It rather distinguishes between non privileged and privileged verbs calls, where privileged calls are available to a non privileged consumer only via a privileged intermediate. The RNIC-PI WG deviates from this semantic by defining the privileged calls also for the non privileged uVP. To allow for its sucessfull completion, the uVP must obtain privileged (system) support. The RNIC-PI specifies a corresponding *rnicpi_syscall()* call which is issued by the uVP during the execution of a privileged operation. The calls arguments are type and arguments of the original verbs call and some uVP/kVP private data. This information must be used by the system to reconstruct the original verbs call as a now privileged call to be issued to the kVPs RNIC-PI (private data are handed over transparently).

Translated to the OpenRDMA architecture, the *rnicpi_syscall()* will be exported from the uAL to the uVP as a callback, and the execution of privileged calls involves further interaction with the kAL and the IHV's kVP modules. Fig. 3 gives an overview on the expected call sequence:

- The consumer is requesting the operation via an RDMA-API-call (1).
- The uAL translates this call into an RNIC-PI call (2).
- The uVP is calling back the uAL to get privileged support for executing the operation (3).
- The uAL is calling the kAL via the uAL/kAL interface (4).
- The kAL is calling the privileged mode RNIC-PI to get the requested operation performed by the kVP (5).
- After typically calling the IHVs hardware (6), the call stack gets finished via (7,8,9,10,11,12) call returns, finally giving back the calls result to the consumer.

It should be noted that it is currently an open issue in the RNIC-PI WG [6], if exposing the privileged calls to non privileged consumers will be mandatory or optional. If the implementation becomes optional and the uVP is not exposing a certain call, the API implementation would have to forward the call to the privileged mode RNIC-PI implementation. With OpenRDMA and following Fig. 3, the uAL would not issue the call to the uVP (2), but directly call the kAL (4).

3.2 RDMA APIs

RDMA APIs are responsible to expose the RDMA services to both privileged and non privileged consumers. The OpenRDMA Group agreed upon implementing both the IT-API in its latest version and the u/kDAPL in its latest version.

3.2.1 The IT-API

The IT-APIv2 makes available to the consumer RDMA services based on both iWARP and InfiniBand transports. The OpenRDMA Group plans to implement the IT-APIv2 as part of the uAL to provide RDMA services to non-privileged consumers. While the IT-APIv2 specification is initially focused on user level consumer support, only minor changes would be necessary to support kernel level consumers. Therefore, an IT-APIv2 implementation as part of the kAL is also planned.

3.2.2 The DAPL

While initially targeting the InfiniBand transport, the DAPL API currently gets extendend to support also iWARP transports. The OpenRDMA Group plans to implement the uDAPL as part of the uAL to provide RDMA services to non-privileged consumers. To provide RDMA services to privileged (in kernel) applications it will integrate a kDAPL implementation into the kAL. Section 5.2 discusses the integration of legacy DAPL solutions with the OpenRDMA DAPL implementation.

3.3 The uAL/kAL Interface

In the OpenRDMA architecture, uAL and kAL together form the host extension for the support of RDMA services. To support user space RDMA service consumers, uAL and kAL must cooperate. This cooperation is necessary to (1) support privileged operations for non-privileged (user level) consumers, (2) for the efficient implementation of user level event handling, and (3) for the implementation of the RDMA APIs where for some operations cooperation with a kernel based API entity is needed (for example, to initiate a TCP stream setup on a kernel socket from user space).

The definition of the uAL/kAL interface is local to OpenRDMA. While the Linux OS provides several methods to structure the communication between a kernel level service (kAL) and a user level consumer (uAL), the classical approach of a kernel device access seems to be most appropriate. With this approach interaction with the kAL gets implemented via device operation on the e.g. */dev/krdma* device (open(), close(), ioctl(), select(), poll(), read(), write(), ...).

3.4 Loading IHV specific Components

The OpenRDMA architecture supports the dynamic loading of the IHV specific components. Being a kernel module, the kVP registers itself with the kAL via a standardized RNIC-PI registration call exported by the kAL. During the execution of this call the kAL creates an RNIC specific interface object as a generic representation of the IHV specific RNIC-PI implementation. With this mechanism the kAL is able to support the simultaneous presence of several RNICs from potentially different IHVs.

The uAL will provide a mechanism to attach the appropriate uVP library on demand. To support this, the RNIC-PI specification currently proposes to pass the RNIC specific library code path as additional information during the kVP/kAL registration call. This information can be signalled from the kAL to the uAL, which is now able to load the appropriate uVP library on demand.

All IHV code loading mechanisms are transparent to the user level and kernel level RDMA service consumers. An RNIC available in the system is selected by the consumer by its network address.

3.5 Generic Interfaces to the Linux System

To integrate the OpenRDMA stack with the Linux OS, the kAL implements all necessary interfaces to the kernel. This includes interfaces to the memory management services, to the kernel network stack implementation and to the process management. Where available, the project aims at using the Linux standard kernel interfaces.

3.5.1 Network Stack Interface

The integration of the OpenRDMA stack into the Linux OS with its host resident network stack has two aspects. First, the OpenRDMA stack relies upon services provided by the host stack implementation, and second, the coexistence of two stacks partially providing the same service must be coordinated. Such, the OpenRDMA stack must maintain kernel networking state and transfer state information between kAL and IHV's kernel modules.

The interfaces to the kernel stack should not be part of the IHV's private code, but implemented as part of the kAL functionality. It is a goal of the implementation to avoid any changes to the original kernel code or, if this cannot be avoided, to keep those changes at their minimum.

The actual transfer of the necessary information (routing entry, next hop MAC adress, LLP context, MIB counters... – see below) is to be carried out between kAL and IHV's kernel module. At the current state of the RNIC-PI specification, it is not clear if some or all of this semantic will become part of the RNIC-PI. All semantics not covered by the upcoming RNIC-PI should go into an open interface, preferable defined by the OpenRDMA project.

LLP Connection Management Interface The OpenRDMA stack interfaces with the host stack to access LLP connection managment services. This interaction is necessary for LLP connection establishment, handover and takedown:

- The kernel stack is used to initiate and accept connections. This functionality can be accessed via the standard socket interface (both kernel level and user level socket interface are usable).
- LLP stream/context handover and takedown: The OpenRDMA stack maintains the transfer of LLP endpoint state information during the RDMA mode transition of the stream. This includes the maintainance of host data structures reflecting the presence and state of the connection (socket layer, timer interface, endpoint lookup hash lists etc.) as well as the handover of the actual context state.

  The OpenRDMA architecture should also support the complete offloading of the connection establishment. In this case, the connection establishment is performed on the RNIC.

  During the takedown of an offloaded connection, the host network state must be updated again.

Routing support It is expected that the offloaded stack will not implement a routing protocol to maintain RNIC local routing information. This information will rather be provided by the host network stack.

MAC address resolution/next hop discovery It is expected that the offloaded stack will not implement ARP/ND. Necessary information will rather be provided by the host network stack.

MIB Interface The SNMP protocol (Simple Network Management Protocol) of the Internet allows to retrieve and set variables, addressed as objects in a Management Information Base (MIB). Linux implements the MIB-2 set of TCP/IP variables containing variables related to the TCP/IP protocol suite including both IPv4 and IPv6. [1] Furthermore, Linux supports the draft variable set for the SCTP protocol and it defines a private 'extended' set of TCP related MIB variables not being part of any current standard.

All MIB variables are written by the kernel just at the moment when the corresponding event happens. Due to the existence of a second entity of at leat parts of the networking stack at the RNIC, the statistics collected by both entities must be merged. This merging can be deferred until an actual request to read the MIB values arises.

[1]The available MIB objetcs are defined in RFC 1213, 2011, 2012, 2863, 2465, 2466.

3.5.2 Memory Management Interface

The kAL unifies the memory management of the OpenRDMA stack. It accesses the Linux memory management via the available kernel interfaces for memory translation and pinning. The unified memory management avoids non coordinated interactions of the IHV's code with the host memory management and, thus, allows for safe pinning and unpinning of communication buffers even in the presence of multiple RNIC's sharing communication buffer memory.

3.5.3 Process Management Interface

All OpenRDMA application process management is related to the uAL/kAL interface implemented as a device access (see Sec. 3.3).

Process management is necessary to control the passing of control during event notification. The Linux OS offers well defined interfaces to put to sleep and wakeup the uAL event handling thread(s) accessing the kAL device, or to inform a selecting thread upon the arrival of a new event record. (see Section 4.3 for the uAL's event handling support).

Furthermore, since implementing the device interface, the kAL will detect unexpected process termination. This allows for a safe cleanup of any resources related to this former RDMA service consumer. Compared to IHV private solutions of this problem, a centralized RDMA resource management will provide advanced system stability and also free the RNIC vendor from implementing OS specific solutions.

4 Event Handling

4.1 Event Types

Events are used to asynchronously inform a consumer about state changes within the protocol stack. The RDMA verbs specification defines two types of events: one type is related to previously issued work requests which could complete in success or error (Work Completion Events), the other type is not related to a certain work request, but to one or more QP's and limited to the report of failures (Asynchronous Events). Tab. 1 gives an overview on the properties of both event types.

Table 1: Event Types

	Work Completion	**Affiliated**	**Non Affiliated**
relates to	single signalled WR	single QP	multiple/all QPs
state signalled in	CQE	Event Record	Event Record
polling	possible	N/A	N/A
Event Handler	CEH	AEH	AEH
# Handlers per RNIC	one or multiple	one	one

Aside of events defined in the Verbs specification, a third type of events must be supported by the protocol offloading architecture. This type is related to the establishment of the LLP stream (the TCP association, for example) which will be used by the RDMA protocol later.

4.2 kAL Support

The kAL is reponsible for all Event Handler registration with the RNIC(s) available in the system. The kAL registers the single Asynchronous Event Handler (AEH) supported per RNIC as well as one or more Completion Event Handlers (CEHs). The maximum number of CEHs which might be registered can be polled from the RNIC using the Query RNIC Verb.

The kAL registers its Event Handlers using the Set Completion Event Handler resp. Set Asynchronous Event Handler verbs. A sucessfull call to this verbs will return a Completion Event Handler ID resp. Asynchronous Event Handler ID. The kAL keeps these IDs to map them later to consumer-registered event handlers.

LLP related events which are not part of the RNIC-PI must be generated by the kAL itself. The LLP Stack Management being part of the kAL will issue these events. The kAL is responsible to forward the corresponding information to consumers or to react locally. Currently, connetion setup events are considered to be passed via this additional, non verbs mechanism.

4.2.1 Kernel Verbs Provider Interaction

Event processing always needs the support of an privileged intermediate responsible for the scheduling of the consumer for event notification. Thus, the notification path will naturally start with the event information transfer between RNIC and the kVP. The kVP will typically install an ISR which gets called after the RNIC placed event records within the event queue located in the context of the kVP. The ISR is responsible for transferring control and information to its consumer, which in the OpenRDMA model is the kAL. To allow for efficient event processing and potential software interrupt accumulation, it is expected that the kVP will split its interrrupt service into a short-path Top Half ISR which is responsible to gather the event records and a Bottom Half scheduled by the ISR which runs the registered Event Handlers. In the current model, the registered Event Handlers represent callbacks to the kAL.

If the kAL gets control during event passing (callback), it is able to demultiplex events to individual consumers based on the information passed with the call:

- Asynchronous Events are described by an Event Record which contains the type of the corresponding resource (QP, CQ, RNIC, S-RQ), a resource identifier (e.g. QP ID) and the Event Identifier which indicates the reason of the event. This information is sufficient to retrieve the consumer or group of consumers.

- A CEH is called with the RNIC Handle and the CQ Handle as arguments. The consumer can be directly derived from this information.

For kernel level consumers the kAL will run the registered event handler. User level clients will get informed via the kAL/uAL Interface. In this case the uAL will run the registered event handler.

4.3 uAL Support

The uAL learns from the kAL the ID's of all Event Handlers (AEH and one or more CEHs), which the kAL registered with the kVP and the kAL is willing to expose to the current uAL. The user application will, dependent on the employed User API, create or register a number of private API objects for reaping work completions and/or for receiving notifications. Such API objects may includer Simple Event Dispatchers (SEVDs), Aggregate Event Dispatchers (AEVDs) and callback functions.

This registration process will initially create only local state in the context of the uAL. If an event dispatcher gets assigned to a local resource like during CQ or QP creation, then the uAL is mapping one appropriate event handler ID (EVH_ID) received from the kAL to this dispatcher and informs both the uVP (e.g. with cq_create()) and the kAL about this mapping. The kAL in turn will invoke the kVP to register the resource handler with the resource (e.g. cq_create()).

The uAL implements a main event handling thread which is constantly awaiting event notifications on the kAL device. Upon event passing, the main event handler thread demultiplexes the events received from the kAL to dedicated local event handlers or simply invokes pre-registered callback functions. In any case, the main event handler thread activates a thread from a pool of idling threads to handle the event. The working thread goes back to the thread pool after finishing execution.

5 Other Issues

5.1 Supporting Software-based RNICs and Partial Offloading

The use of the standardized RNIC-PI as the interface to integrate IHV private RNIC driver code allows for any level of partial offloading of RNIC functionality or the complete software based implementation of an RNIC on top of an Ethernet device. From the viewpoint of the OpenRDMA system, it is of no difference if an RNIC is implemented partially or completely in software or all protocol layers functionality is completely offloaded to the network interface card.

At the other hand, it is currently not the intention of the OpenRDMA project to support the system integration of an explicit TCP offloading only mode (TOE mode). The RNIC-PI will be the only interface to communicate with the IHV's modules.

5.2 Integration of Legacy DAT Solutions

With DAPL (Direct Access Transport API) an RDMA capable API for user level (uDAPL) and kernel level (kDAPL) was defined by the DAT collaboration. The OpenRDMA Group agreed upon the support of the DAPL standard as one of the RDMA capable API's. It is furthermore acknowledged that today a significant amount of DAPL applications do exist under Linux and that those applications should benefit from the OpenRDMA services. Incorporating legacy DAPL based RDMA applications involves three issues:

- Binding the applications with the DAPL provided by the OpenRDMA (uAL/kAL),
- co-existing with DAPL applications which are not using the OpenRDMA service, and
- using legacy DAT provider modules.

The binding of available DAPL applications to the OpenRDMA stack will involve the recompilation of the DAPL application. Thus, only applications where the source code is available can be adapted to OpenRDMA.

Existing implementations of the DAPL interface will typically maintain a system wide registry for the DAPL methods of the providers. This method allows for the flexible assignment of provider specific DAPL implementations without polluting the name space of an application. The OpenRDMA Group agreed upon the approach, that OpenRDMA will maintain this system wide registry and assign itself to this list as a DAPL provider.

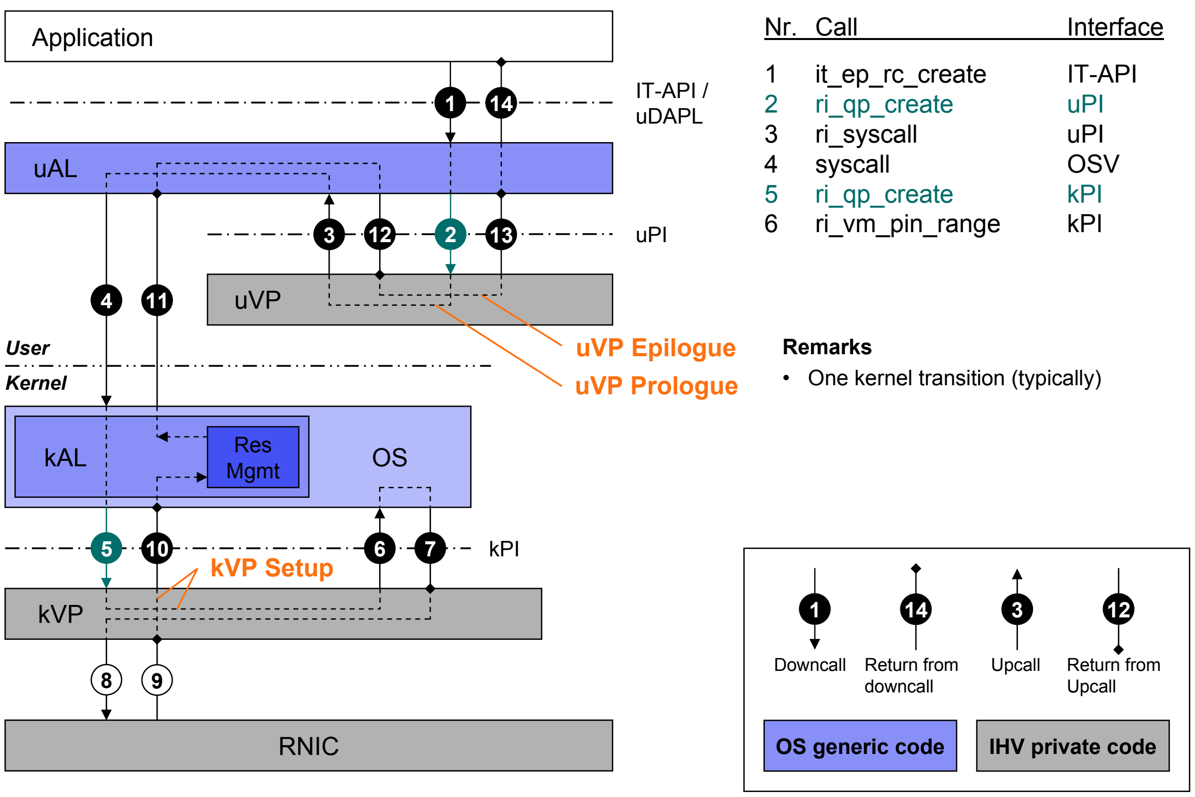
The OpenRDMA Group refuses the integration of legacy DAPL providers into the OpenRDMA stack, since typically the source code of the providing modules will not be available or would need significant recoding. DAPL providers are encouraged to write the appropriate modules to fit into the OpenRDMA architecture. The co-existence with legacy DAPL providers remains unaffected.

6 Operational Examples

This section is intended to illustrate the operation of the OpenRDMA architecture. Using sequence charts, some typical sequences of operation from initiation until completion will be considered.

6.1 Queue Pair Creation for non privileged Consumer

Creation of a Queue Pair for a user space consumer is one of the typical operations during the setup of an RDMA association. It is a control type operation, which potentially creates new state at the RNIC, within the kVP and uVP modules, and in both uAL and kAL. From a user level consumers perspective, this process involves the execution of privileged operations the application is not entitled to do without support by the privileged kernel proxy.



Nr.	Call	Interface
1	it_ep_rc_create	IT-API
2	ri_qp_create	uPI
3	ri_syscall	uPI
4	syscall	OSV
5	ri_qp_create	kPI
6	ri_vm_pin_range	kPI

Remarks

- One kernel transition (typically)

Figure 4: QP Creation for non privileged Consumer

Fig. 4 illustrates the sequence of interface calls and operations which are typically involved during the creation of a Queue Pair from user space. It is assumed, that the application is using the IT-APIv2 for accessing the RDMA services. Following Fig. 4, the following steps would follow: [2]

- IT-API Consumer invokes it_ep_rc_create (1).
- uAL sets up an ri_qp_attributes_t data structure and invokes ri_qp_create (2) across the uPI.
- uVP may allocate data structures for control data and/or queue data in virtual memory, by calling malloc.
- After the optional malloc, uVP sets up an ri_qp_param_t that contains both QP attributes and IHV's opaque hdm_data and invokes ri_syscall (3). In ri_syscall (3), the uVP provides op == RI_CREATE_QP and a pointer "param" to the ri_qp_param_t struct.
- uAL invokes the OS-provided syscall (4), passing the unmodified ri_syscall parameters from (3), and possibly more parameters, down to the kAL.
- kAL translates syscall (4) to a call of ri_qp_create (5) across the kPI. This call also passes the IHV's opaque hdm_data to the kVP.
- kVP may use the ri_vm_pin_range upcall (6) in case virtual memory allocated in the uVP needs to be pinned.
- kVP sets up RNIC state (8,9) and returns QP handle to kAL (10).

[2]The description follows the terminology used in [4, 6]

- kAL performs generic resource management, which may include generating a generic Endpoint handle and associating that Endpoint handle with the IHV-specific QP handle returned with (10). kAL returns from OS syscall to uAL (11).
- uAL associates the generic Endpoint handle, which may have been passed through (11), with the QP created through call (1). Then uAL returns from ri_syscall to uVP (12).
- uVP performs final setup operations (uVP Epilogue) and returns from ri_qp_create to uAL (13).
- uAL returns from it_ep_rc_create to Application (14).

6.2 Event Handler Registration

TBD: kAL/kVP interaction, consumer/uAL/kAL interactions

6.3 Event Handling

TBD: kAL/kVP interaction, consumer/uAL/kAL interactions

6.4 RDMA Mode Transition

TBD: MPA, LLP Context Handover, ...

6.5 Data Transfer Operations

7 Open Issues

- ip chains
- ip configuration
- context handover
- ...?

Abbrevations

Table 2: Abbreviations

AEVD	Aggregate Event Dispatcher
ARP	Address Resolution Protocol
CEH	Completion Event Handler
CEH_ID	Completion Event Handler Identifier
CQ	Completion Queue
CQE	Completion Queue Element
EVH	Event Handler
IHV	Independent Hardware Vendor, provider of RNIC hardware and related driver software
iWARP	A suite of wire protocols comprised of RDMAP, DDP, and MPA. The iWARP protocol suite may be layered above TCP, SCTP, or other transport protocols.
kAL	kernel mode Access Layer
kVP	kernel RDMA Verbs Provider
LLP	Lower Layer Protocol. The protocol layer beneath the protocol layer currently being referenced. For this document, LLP is used to reference the protocol providing the transport service for an RDMA protocol, including a TCP and SCTP connection, and a Infiniband association.
ND	Neighbor Discovery Protocol
QP	Queue Pair
RNIC	RDMA capable Network Interface Card
RNIC-PI	RNIC Programming Interface. Specification of syntax and semantic of a standard programming interface. The RNIC-PI specifies the implementation of both iWARP and InfiniBand RDMA verbs.
SEVD	Simple Event Dispatchers
SRQ	Shared Receive Queue
uAL	user mode Access Layer
uVP	user RDMA Verbs Provider
RDMA Verbs	description of the semantics of the RNIC-PI calls
WQE	Work Queue Element

References

[1] J. Hilland et al. RDMA Protocol Verbs Specification. *Internet draft, draft-hilland-rddp-verbs-00.txt*, April 2003.

[2] InfiniBand Trade Association. Supplement to InfiniBand Architecture Specification Volume 1, Annex SWG9, Verbs Extensions, downloadable at *http://www.infinibandta.org/specs/*

[3] J. Pinkerton et al. DDP/RDMAP Security *Internet draft, draft-ietf-rddp-security-05.txt*, August 2004.

[4] The Open Group/ICSC. IT-API Interface Specification Version 2 *Work in progress*, October 2004 IT-API Version 1 online at *http://www.opengroup.org/icsc/native/*

[5] Direct Access Transport APIs. uDAPL online at *http://www.datcollaborative.org/udapl12_091504.zip* and kDAPL online at *http://www.datcollaborative.org/kDAPL12_091504.zip*

[6] The Open Group/ICSC. RNIC-PI Interface Specification Version 1 *Work in progress*, October 2004

[7] UNH-IOL iWARP Consortium. *http://www.iol.unh.edu/consortiums/iwarp/*