Implementation and Analysis of the User Direct Access Programming Library

James Lentini, Vu Pham, Steven Sears, and Randall Smith

***Abstract*—The User Direct Access Programming Library (uDAPL) is a generic application programming interface (API) for network adapters capable of remote direct memory access (RDMA). The uDAPL interface allows user space applications to work with RDMA adapters using a platform and transport independent API. The uDAPL interface has been proposed for use in clustering, distributed systems, and network file systems.**

In this paper we evaluate the uDAPL interface and share our experiences developing an open source implementation using InfiniBand adapters.

***Index Terms*—remote memory access, system area network, memory to memory interconnect.**

I. Introduction

RDMA network adapters are characterized by two important features: allowing user space applications to directly access hardware and zero-copy data movement. A combination of hardware and software allows user space applications to read and write the memory of a remote system without kernel intervention or unnecessary data copies.

These features result in lower CPU utilization per I/O operation and more efficient use of machine resources than typical networking architectures. User space networking improves performance by eliminating domain crossings between user space and kernel space when transferring data. System security is maintained by having the kernel allocate the system and adapter resources up front. As the cost of a data copy becomes a larger portion of a complete data transfer, the need for zero-copy transfers becomes more important. Zero-copy data transfers are possible within traditional protocol stacks [1], but generally involve large modifications and complex interactions with the operating system's virtual memory subsystem. Use of RDMA capable transports directly by an application eliminates these problems.

A number of applications have been developed that take advantage these benefits. Research has demonstrated the feasibility of building high performance clusters using RDMA interconnects [7]. A number of companies, including Network Appliance, have sold products that use RDMA interconnects for clustering. In the area of distributed systems, Software Distributed Shared Memory has been successfully implemented upon RDMA interconnects [10]. The Direct Access File System (DAFS) is a network file system that uses RDMA technology and has demonstrated performance improvements over traditional network file systems [8].

In recent years, several network interconnects with RDMA capabilities have been proposed or developed. These include the Virtual Interface Architecture (VIA) [2], the InfiniBand Architecture (IBA) [6], and RDMA over the Internet protocols [11][12][13] to name a few. Despite sharing similar capabilities, each technology has required applications to use a transport or vendor specific programming interface. As a result, applications have become tied to specific implementations. These incompatible programming models have hampered the spread of RDMA technology.

In an effort to standardize RDMA APIs, the DAT Collaborative, an industry group focused on RDMA technologies, developed kernel [3] and user [4] space specifications in the C programming language for RDMA transports. The kDAPL and uDAPL APIs are an attempt to create a portable set of APIs for all RDMA networks. The relationship between kDAPL, uDAPL, consumer applications, and RDMA hardware is show in Figure 1. The APIs will allow applications to utilize the capabilities of both current and future RDMA networks without becoming tied to a specific implementation. Since these networks are used for high performance applications, the interfaces have been designed to be as efficient as possible. Appendix A is a listing of the uDAPL API v1.0. The DAT Collaborative is currently preparing a new version of the specification, which is anticipated to include a handful of enhancements and address any errata items discovered in version 1.0.

The authors have developed an open source reference implementation of the uDAPL v1.0 interface[1]. The project is

James Lentini, Steven Sears, and Randall Smith are Members of the Technical Staff at Network Appliance, Inc. 375 Totten Pond Rd., Waltham, MA 02451 USA.
(email : jlentini@netapp.com, sjs@netapp.com, rsmith@netapp.com).
Vu Pham is a Software Development Engineer at Mellanox Technologies Inc. 2900 Stender Way, Santa Clara, CA 95054 USA.
(email: vuhuong@mellanox.com)

[1] http://sourceforge.net/projects/dapl

hosted on SourceForge.net, allowing other individuals and corporations to assist with development. Currently the reference implementation supports the IBA as the underlying transport using either an IBM [5] or Mellanox [9] Verbs interface. Internally, the reference implementation uses an abstract interface for all platform dependent functions. Support for Linux and Windows has been implemented.

The design of the uDAPL reference implementation presented a number of interesting problems. The project began before stable IBA hardware and software was available. By beginning early, we were forced to simultaneously debugged the uDAPL reference implementation, the Verbs, the adapter's device driver, and the InfiniBand hardware itself. As a result, we were compelled to approach problems from a number of different perspectives to resolve the underlying problems. Along with the rapidly changing development environment, we were faced with the challenge of supporting uDAPL's semantics with the capabilities available in the IBA. As a result, the implementation was forced to use the provided IBA verbs interfaces in creative ways.

The goal of the reference implementation was to create a portable code base from which RDMA adapter vendors could base products supporting the uDAPL API. As a result, achieving the absolute best possible performance was not our primary goal. However, after completing the reference implementation, the authors were able to characterize the performance using the following metrics: CPU utilization, operation throughput, bandwidth, and latency. Using these numbers and measurements of the native RDMA interface, we were able to quantify the performance penalty imposed by interposing the reference implementation between consumers and the native interface.

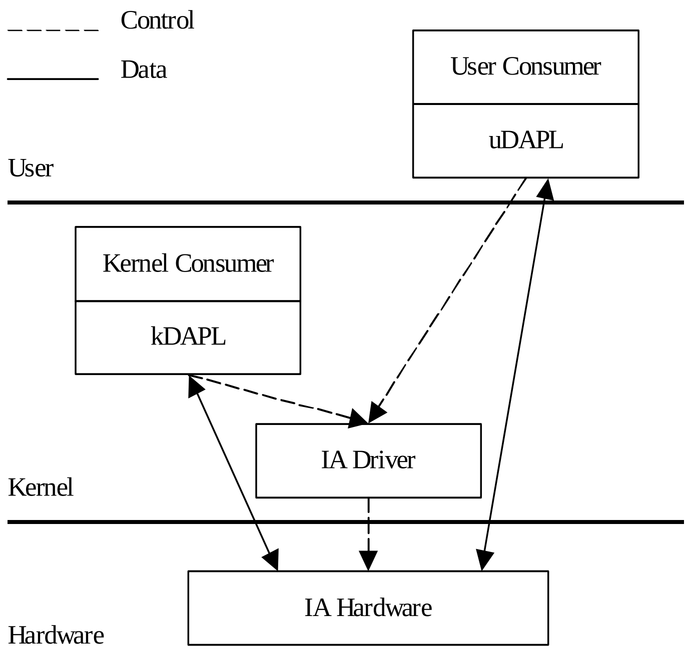

Figure 1: DAT Architecture

II. Architecture of uDAPL

A. Handles and Objects

The uDAPL specification represents the various abstract concepts of RDMA networking as objects. For example, an Interface Adapter (IA) is the object used to represent a RDMA network adapter and an Event Dispatcher (EVD) is the object that queues events for the consumer. These objects are related to one another in an ownership hierarchy. For example, an EVD object is created as the child of a specific IA.

Consumers manipulate these objects through handles. Each object type has creation and destruction functions to allocate and deallocate object resources. The creation functions return a handle with which the consumer can manipulate the object and associate it with other objects.

B. Registry and Providers

As noted earlier, one of the primary goals of the uDAPL specification was to provide a standard interface to RDMA network adapters from multiple vendors. Applications should only need to link against a standard uDAPL library to communicate with a RDMA network adapter. However, vendors require specialized support for their adapter's hardware-software interface. Therefore a mechanism was needed to allow vendors to provide their own architecture dependent implementations of the uDAPL functions.

The solution was to create a level of indirection in the form of a registry between the consumer and the vendor specific software. The later was termed a provider. Providers are statically registered with uDAPL through a platform specific mechanism. On UNIX systems, this mechanism is implemented as a text configuration file. Regardless of the particular approach used to record this information, the consumer application need not be aware of the internal workings of the registry. After opening a provider through the registry, the consumer communicates directly with the provider library on subsequent operations.

Direct communication is achieved through the use of function pointer tables and C macros. Each uDAPL function other than *dat_ia_open()* and *dat_ia_close()* is actually a C macro. These macros use their first parameter to retrieve a standard function pointer table (implemented as a C structure called the DAT_PROVIDER). The macros index into the table and call the appropriate function passing the caller's parameters through to the provider. Unlike the other functions, the *dat_ia_open()* and *dat_ia_close()* functions have standard implementations. These functions allow the consumer to interact with the uDAPL registry. The *dat_ia_open()* function searches the registry for the provider specified by the consumer. If the desired provider is found, the registry calls the providers IA open function. If this call succeeds, the consumer will be given a DAT_IA_HANDLE pointing to one of the function pointer tables described above.

This design allows applications to use the same symbols to invoke different provider implementations at the cost of incurring a pointer dereference on each call. As a result of this design, every provider's uDAPL handles must be implemented as a pointer to the standard function pointer table (see Figure 2).

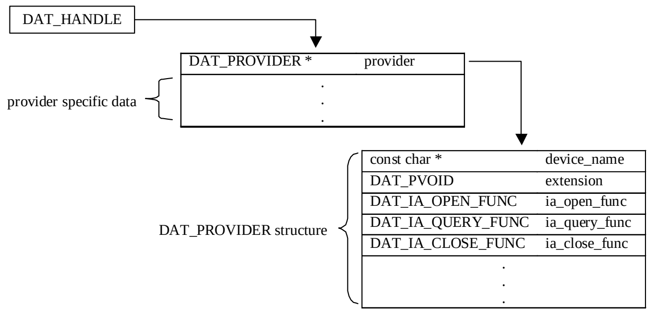

Figure 2: DAT handle design

Consumers identify providers with the IA name string passed to *dat_ia_open()*. If uDAPL naively used the device naming conventions available at the IBA Verbs layer, it would be possible for vendors to introduce conflicting device names. While InfiniBand Verb libraries typically provide an interface to open an adapter, separating the user from the file system device name, they are not as general as needed by the uDAPL interface. For example, the IBM Access API only allows an adapter to be named by an ordinal number, introducing the possibility that two different vendors utilizing this API will have identical device names.

To avoid name collisions among vendors, uDAPL adapter names differ from the names exported by the Verbs library by adding a simple prefix to the name. For example if vendor Xyz uses a Verbs API that exports adapter names 1, 2, and 3 and vendor Abc uses a Verbs API that exports adapter names 1,2, and 3, we have a conflict. uDAPL resolves this by allowing a vendor to specify a simple prefix to the device name, e.g. xyz-1, xyz-2, abc-1, etc., and provides guarantees that each prefix is unique.

C. *Protection Model*

To defend against malicious or faulty applications, RDMA adapters typically provide a mechanism to protect resources allocated on the adapter. This concept is termed a Protection Zone (PZ) in the uDAPL API. PZs are created in the scope of a single IA. All uDAPL objects that require IA resources are associated with a PZ. To associate a set of such objects with one another, a consumer must ensure that all the objects belong to the same PZ. This also ensures that the objects are only associated with one another if they are located on the same IA.

D. *Event Model*

The results of many operations in the uDAPL interface are communicated to the consumer through asynchronous events. Consumers place an operation on queues for processing and either poll or wait on EVD objects for the corresponding events signaling the operation's result.

E. *Connection Model*

uDAPL supports reliable connections using a client-server connection model. The passive side of a connection creates a service point. Two types of service points are defined: reserved service points (RSPs) and public service points (PSPs). The active side creates an Endpoint (EP) object and sends a connection request to the specified address and service point. The remote side is then free to accept or reject the connection request. If the request is accepted, an EP is created on the remote side and communication continues until either party disconnects or the connection is broken due to error.

F. *Exchanging Data: Data Transfer Operations*

uDAPL supports four data transfer operations: send, receive, RDMA read and RDMA write. Send and receive operations are paired with one another; one side of the connection posts a receive and the other posts a send. In contrast, RDMA operations allow for remote memory to be read and written without consuming a remote operation.

G. *Memory Management*

Before an application buffer may be used as the source or destination of an operation, a memory region containing the buffer must be registered with the adapter. The virtual page(s) in which the memory region lies is pinned into main memory to ensure its availability to any operations referencing it. Two levels of memory registration are provided: local and remote. Local memory regions (LMRs) may be used as the source of a send or RDMA write operation, or the destination of a receive or RDMA read operation. Remote memory regions (RMRs) may be the remote target of a RDMA read or write operation.

RMRs provide an additional level of protection for applications using RDMA operations. To use a RMR, the consumer must bind the RMR to an LMR. An arbitrary number of RMRs may be bound to a single LMR. The bind operation produces a key with which remote nodes may identify the region. Only remote operations that specify the proper key are allowed to access the region. To revoke the key, the consumer can unbind the RMR. Using this mechanism, the application can easily restrict remote access to a region.

III. Design of the Reference Implementation

A. *Independent Architecture*

1) *Platform Independent*

The reference implementation used a platform independent interface for all operating system or processor architecture functions. Even the standard C runtime routines were abstracted out and accessed through the platform independent interface. This has resulted in a highly portable implementation that may be used in either popular environments such as Linux and Windows or more specialized embedded operating systems.

2) *Chip Independent*

The implementation was careful not to use any chip dependent features of the IBA adapters on which the software runs. Rather the implementation can be compiled to work with either the IBM or Mellanox Verbs interface. Both of these APIs have been released to the public and have become the two most popular Verbs interfaces. By not using chip specific features, the reference implementation will be portable to future generations of adapters providing either Verbs interface.

B. *Design of the uDAPL Event Subsystem*

The event system is the heart of the DAT model. Nearly all API invocations are asynchronous in nature, the results are returned in an event. Completions are logically grouped into Event Streams, which feed into Event Dispatchers. Event Stream notifications include data transfer completions, connection requests, connection establishment, disconnect notifications, memory bind completions, asynchronous errors, and software generated events.

An Event Dispatcher will organize events from one or more streams into a single queue; events can be dequeued exactly once. Consumers can either poll or wait on an event dispatcher. They can also wait on a set of Event Dispatchers using a Consumer Notification Object (CNO). A CNO can optionally trigger an OS specific synchronization object through the mechanism of OS Proxy Wait Objects.

Since uDAPL was implemented upon an IBA Verbs interface, the reference implementation needed to implement the abstract objects described in the uDAPL specification with the resources found in the IBA. Events in IBA are either delivered through user calls (software generated Events), through IBA Completion Queues (data transfer completions and memory bind completions), or through pre-registered callbacks (everything else). Event Dispatchers in the uDAPL reference implementation must incorporate all of these event stream types into a common interface for the user. Issues that arose in this incorporation include:

- the nature of event storage
- supporting a threshold in *dat_evd_wait()*
- inter-producer synchronization around event storage
- "impedance matching" between blocking and callback models

We discuss each of these in turn.

1) *The Nature of Event Storage*

The information associated with an event (e.g. the amount of data received for a receive data transfer operation) must be stored somewhere between when the event is generated and when the user retrieves it. For CQ associated events, that information may in most cases be left on the CQ, and the operation of dequeueing from the EVD implemented by polling the CQ. For non-CQ associated events, that information must be stored in a uDAPL defined structure on the EVD. Because uDAPL is intended for high-performance applications, it is important to minimize the synchronization overhead associated with that structure.

In the reference implementation, that structure is a producer/consumer circular queue. When an event is generated (via callback or *dat_evd_post_se()*) the producer enqueues it; when an event is requested by the user, it is dequeued. These queues are implemented directly upon architecture specific atomic operations, abstracted through our platform independent layer; no locking is required for enqueueing or dequeueing an event. The circular queue avoids the necessity of explicit synchronization between the event producer and the event consumer. Each may operate independently unless the queue is empty (no events are available) or full (an overflow asynchronous error is generated).

The event consumption operation (*dat_evd_dequeue()* or *dat_evd_wait()*) must check both this internal queue and the contents of the CQ associated with the EVD. Because information may sometimes be copied from the CQ to the EVD's circular queue (see "Supporting a threshold in *dat_evd_wait()* below), the EVD storage is checked first to maintain ordering of completions posted to the CQ.

2) *Supporting a threshold in dat_evd_wait()*

One important difference in semantics between IBA and uDAPL are *dat_evd_wait()*'s arguments threshold and nmore. The threshold input parameter indicates how many events are required before the thread blocking in *dat_evd_wait()* should be awoken. The nmore output parameter indicates, at function exit, how many events remain to be dequeued from the EVD.

IBA has no related concepts; either of blocking for a specified number of events, or of "peeking" at a CQ to determine how many CQEs are present on it at any given time. This has two important implications:

1. On IBA the full benefit of thresholding is not achievable, and
2. any requirement for the number of elements on a CQ can only be satisfied by dequeueing all elements from the CQ.

The DAT model requires this information in the implementation of *dat_evd_wait()* for two reasons. When a notification is received that there are completions available on the CQ, the number of completions is not specified. Hence all entries must be dequeued from the CQ to determine if the number of available entries is greater than that of the passed threshold and thus the *dat_evd_wait()* call may return. Additionally, upon return from the *dat_evd_wait()* call, the nmore parameter must be filled in with the number of entries remaining on the EVD. This disallows the obvious optimization of avoiding the copy when the threshold is equal to 1.

For these reasons, *dat_evd_wait()* is implemented as copying all available data to the internal EVD circular buffer, and then testing its exit conditions and returning its information based on that circular buffer. *dat_evd_dequeue(),* which does not have *dat_evd_wait()*'s event counting requirements, simply checks both the EVD and the CQ, dequeueing directly from the CQ into the output data structure if appropriate.

3) Inter-Event Producer Synchronization

Because, unlike CQs, an EVD may have multiple competing producers for event streams associated with it, inter-producer synchronization becomes relevant. If multiple threads may be enqueueing events to the EVD at the same time, we must synchronize access to the producer side of the queue. Alternatively, if, because of the nature of the EVD, only one producer will be acting on it at a time, we may avoid that synchronization overhead. In the DAT model, the usage of EVDs is fully specified when they are created. Thus it is possible to determine, at EVD creation time, whether or not this synchronization overhead will be required.

The classes of threads that will be delivering events to the EVD circular queue include callbacks from the IBA Verbs layer and user threads in *dat_evd_wait()* copying data as described above[1]. In general, the appropriate criteria for inclusion of the synchronization overhead is whether or not multiple threads will be delivering events to the EVD circular queue at the same time. In other words, if callbacks from the IBA verbs layer may be occurring simultaneously with each other, or with user threads copying data from CQ to EVD, producer side locking on the EVD queue should be enforced.

Because the general thread context model for user callbacks within IBA Verbs implementations is often not completely specified, and because the non-CQ related event streams are usually not performance critical, we chose to enable producer side locking whenever any non-CQ related event streams are associated with the EVD.

4) "Impedance Matching" between Blocking and Callback Models

Most systems for data transmission have some way to yield the CPU when it is not needed, and reclaim it when an operation completes. In uDAPL this is done through blocking; the functions *dat_evd_wait()* and *dat_cno_wait()* take an existing user thread and block it until the next event arrives. In IBA, a callback is specified, and the IBA verbs layer calls that callback when the event occurs.

The callback model fits well into the goal of minimizing context switches in a kernel context. The hardware mediating the data transmission will interrupt the CPU when the data transmission completes, and the callback thread may be invoked directly from that interrupt thread. In contrast, the blocking model is better suited to the goal of minimizing context switches in a user model. At least one context switch will be required from the hardware interrupt (since interrupt threads will not call into user space). That context switch could be to a provider thread, which returns to user space and does a pre-registered callback, or it could be to a user thread previously blocked in the kernel. Simplicity of the user programming model argues in favor of the blocking approach; all work is done within threads the user controls, and no extra thread resources need to be managed by the provider.

The above paragraph describes the ideal implementation, of a uDAPL provider implemented directly upon the data transmission hardware. The reference implementation is implemented on top of the IBA verbs, and hence must translate between the callback model of IBA and the blocking model of uDAPL. It does this through internal wait objects inside of both EVDs and CNOs. These objects are OS-dependent, and are mediated through our platform independent layer. The IBA callback wakes up any thread(s) blocked on the EVD or CNO; those threads then return from their blocking call to *dat_evd_wait()* or *dat_cno_wait().*

For event streams for which these callbacks deliver the data associated with events (non-CQ affiliated event streams) these callbacks must be always enabled. If they do not occur, there is no way for a caller to the EVD to know that an event has occurred. However, for CQ affiliated event streams, the only function of these callbacks is waking up blocked user threads, and if there is no need for this function (because no user threads are blocked) they should be disabled.

Ideally, this would mean that CQ callbacks were enabled upon entry to either *dat_evd_wait()* or *dat_cno_wait()* and disabled upon exit. In the case of EVDs this is in fact exactly what the reference implementation does. Unfortunately, because an arbitrarily large, and dynamically varying, number of EVDs may be associated with a CNO, this algorithm does not scale very well for CNOs. Thus if an EVD has an associated CNO, the CQ associated with that EVD has its callbacks always enabled.

C. Design and Tradeoffs of the Connection Model

An Endpoint (EP) is the fundamental channel abstraction in the uDAPL API. An application communicates and exchanges data using an Endpoint. Most of the time Endpoints are explicitly allocated, but there is an exception whereby a connection event can yield an Endpoint as a side effect.

The connection model for uDAPL is strictly a client-server model. Clients initiate connections using an Endpoint. Servers advertise connection points using one of two types of Service Points: a *Public Service Point* (PSP), or a *Reserved*

[1] The copying of data from the CQ to the EVD is the responsibility of the user thread rather than the CQ notification thread to simplify the synchronization around the CQ because of the exclusionary requirements of the DAT API (*dat_evd_dequeue()* will fail if there is a *dat_evd_wait()* in progress). As a result, only a single thread dequeues from the CQ at any given time.

Service Point (RSP). A PSP creates a persistent listener that can service any number of connections, while a RSP listens for a single connection and connects the EP when it is created.

All DAT connections are point to point; there is no notion of unicast versus multicast addressing in the DAT model. This may appear in future work as most transports provide some support for multicast addresses.

1) *Mapping an Endpoint to an InfiniBand QP*

The properties of a uDAPL *EP* do not exactly match those of an IBA *QP*. The differences introduce constraints that are not obvious. There are three primary areas of conflict between the DAT and IBA models:

- EP and QP creation
- Provider created passive side EPs
- Connection timeouts

a) *EP and QP creation*

The most obvious difference between an EP and a QP is the presence of a protection handle when an object is created. IBA requires a Protection Domain be specified when a QP is created. In uDAPL, a Protection Zone (PZ) maps to an IBA Protection Domain. uDAPL does not require a PZ to be present when an EP is created, and that introduces two problems:

- If the PZ is not provided when an EP is created, a QP will not be bound to the EP until *dat_ep_modify()* is used to assign the PZ. A PZ is required before RECV requests can be posted and before a connection can be established.
- If a uDAPL consumer changes the PZ on an EP before it is connected, uDAPL must release the current QP and create a new one with a new Protection Domain.

b) *Provider created passive side EPs*

The second area where the uDAPL and IBA models conflict is a direct result of the requirement to specify a Protection Domain when a QP is created.

uDAPL allows a PSP to be created in such a way that an EP will automatically be provided to the user when a connection occurs. This is not critical to the uDAPL model but in fact does provide some convenience to the user. IBA provides a similar mechanism, but with an important difference: IBA requires the user to supply the Protection Domain for the passive connection point. This Protection Domain is supplied to all QPs created as a result of connection requests. In contrast, uDAPL mandates a NULL PZ and requires the user to change the PZ before using the EP.

The reference implementation does not use the IBA mechanism because of the problems cited. When a connection request arrives, if the PSP has set the appropriate flag to create an EP upon connection, the connection handler will create the EP and hand it off to the user process through the event mechanism. The EP is created without binding it to a QP. The uDAPL specification requires the user to modify the EP by binding a PZ to it before *dat_cr_accept()* can complete the connection, so a QP will be bound to the EP in a *dat_ep_modify()* operation.

c) *Connection Timeouts*

The third difference between the uDAPL and IBA models has to do with timeouts on connections. InfiniBand does not provide a way to specify a connection timeout, it will wait indefinitely for a connection to occur. This is implemented using a separate timeout thread that cancels the connection request and awakens a blocked thread, if there is one waiting.

2) *Connecting Endpoints*

To help explain the connection model, a description of the steps taken to establish a connection is given below. This example will discuss the various EP states and reveal the mechanisms used when mapping uDAPL semantics to IBA. The differences between connecting to a PSP and connecting to a RSP will be noted.

A server application will create a PSP with *dat_psp_create()*. The interesting parameters to this function include: the connection qualifier, which provides a unique identifier for incoming connections (similar to a Transmission Control Protocol (TCP) port number), the EVD handle, where connection events will be reported, and the PSP flags, indicating if the user or the implementation will provide EPs for connections. This is an asynchronous call, the application may poll or block on the connection EVD until a connection request arrives.

When a connection is made to a RSP service point, the reception of a connection request will cause it to refuse all further connections.

A connection request is initiated when the client application invokes *dat_ep_connect()*. Like most uDAPL functions, this is asynchronous and the completion will arrive as an event on the connection EVD. The EP will be put into the CONNECTION_PENDING state until the CONNECTED event arrives.

A connection request will arrive on the server node resulting in a callback to the PSP connection handler routine. The request will be verified by the consumer and rejected if a problem is detected. Otherwise, connection processing begins.

Once a connection request is in progress, a Connection Request (CR) is created by the uDAPL implementation. A Service Point is simply a mechanism to receive connection requests and provide events to the application. The application will in fact interact with a CR when establishing a connection. The CR contains the address from which the request originated along with any private data sent in the connection request.

If the psp_flags specify DAT_PSP_PROVIDER_FLAG, an EP will be created and attached to the CR at this time. The CR is then delivered to the application in an event.

The application will supply an EP, or modify the provided EP to the correct PZ. It may elect to change other EP parameters as well. The application will then invoke *dat_cr_accept()* to accept the connection.

The accept operation results in a CONNECTED event on the client node, which in turn will send a message back to the server verifying the connection and generating a CONNECTED event on the server. Either side is now free to send data on the EP.

D. Addressing and Naming

The uDAPL Specification calls for a DAT_IA_ADDRESS_PTR to be an Internet Protocol (IP) address, either IPv4 or IPv6. On most systems this is implemented as struct sockaddr. This discussion focuses on the Linux implementation, and can be easily adapted to other systems.

InfiniBand addressing uses a dynamically assigned address called a LID; often referred to SLID for Source LID, and DLID for Destination LID. LIDs are bound to GIDs, which are similar to MAC Addresses used by Ethernet adapters. Using the IBM Access API, the application must know the remote GID in order to connect. Mapping an IP address to a GID is necessary to meet the requirements of naming.

The long-term solution to resolving a uDAPL address, which is in fact an IP address, to an IBA address (a GID) is to make use of an IPoIB implementation. The reference implementation has defined a simple API to access the mapping information maintained by an IPoIB implementation. However, IPoIB implementations are not yet common. Until IPoIB is in widespread use, the uDAPL implementation provides a simple name service facility. This depends on two things: valid IP addresses registered and available to standard name service calls, such as gethostbyname(), and a name/GID mapping file.

IP addresses may be set up by system administrators simply by editing the values into the /etc/hosts file.

A simple mapping of names to GIDs is maintained in the an ibhosts file, currently located at /etc/dapl/ibhosts. The format of the file is:

```
<hostname>        0x<GID Prefix>  0x<GUID>
```

For example:

dat-linux3-ib0p0	0xfe800000	0x1730000003d11
dat-linux3-ib0p1	0xfe800000	0x1730000003d11
dat-linux3-ib1	0xfe800000	0x1730000003d52
dat-linux5-ib0	0xfe800000	0x1730000003d91

For each hostname, there must be mapping from hostname to IP address. We have adopted the convention of naming each IBA interface using the following format:

```
<node_name>-ib<device_number>[port_number]
```

Such conventions are outside the scope of the uDAPL standard and are completely up to the local administrator. In the above example we can see that the machine dat-linux3 has three IBA interfaces with two ports on one HCA and another port on a second HCA.

E. Design and tradeoffs of Data Transfer Operations

In IBA, operations are termed work request elements, and vendors are free to implement their own unique formats. The reference implementation needed to convert from the generic format of a uDAPL operation to these specialized formats. The result is a straightforward translation of the DAT_LMR_TRIPLET to a particular vendor's work request format. Slightly different translations are performed depending upon the Verbs API being used.

To help consumers match completion events to a corresponding DTO, both uDAPL and the IBA allow consumers to insert a cookie value into the DTO that is returned in the subsequent completion event. uDAPL consumers specify these DAT_DTO_COOKIE values when posting a DTO. Rather than store the consumer's DAT_DTO_COOKIE directly in the work request element, the uDAPL reference implementation must store a pointer to its own internal cookie structure, a DAPL_DTO_COOKIE. Different values will be placed in the cookie, according to the type of operation and the type of completion data required. This is a simple scheme to associate uDAPL data with the DTO and corresponding completion event.

One of the fields held in the DAPL_DTO_COOKIE structure is a length, necessary to bridge another gap between uDAPL and IBA. uDAPL specifies that all DTO operations return a length; IBA only returns a length for receive operations. Therefore uDAPL must keep track of send and RDMA write lengths and return them in the appropriate completion events.

At first DAPL_DTO_COOKIE structures were allocated during the posting of a DTO. An obvious performance improvement was to create a pre-allocated pool of cookie structures to minimize the time on the performance critical path for posting an operation.

F. Memory Management Design

The memory management subsystem allows consumers to register and unregister memory regions. Registered regions are needed for DTO operations.

In the reference implementation, uDAPL LMRs are mapped onto IBA Memory Regions. LMR creation produces two values: a DAT_LMR_CONTEXT and a DAT_LMR_HANDLE. The DAT_LMR_CONTEXT value is used to uniquely identify the LMR when posting a DTO. These DAT_LMR_CONTEXT values are mapped directly onto IBA L_KEYs. The uDAPL API exposes these DAT_LMR_CONTEXT values to consumers to allow sharing of memory registrations between multiple address spaces (e.g. between processes). The mechanism by which such a feature would be implemented does not yet exist. Consumers may be able to take advantage of this feature on future transports.

Since some uDAPL functions need to translate a

DAT_LMR_CONTEXT value into a DAT_LMR_HANDLE (e.g. *dat_rmr_bind()*), a dictionary data structure was used to associate DAT_LMR_CONTEXT values with their corresponding DAT_LMR_HANDLE. Each time a new LMR is created, the DAT_LMR_HANDLE is inserted into the dictionary with the associated DAT_LMR_CONTEXT as the key.

A hash table was chosen to implement this data structure. Since the L_KEY values are being used by the RDMA adapter hardware for indexing purposes, the distribution is expected to be uniform and hence ideal for hashing.

The reference implementation maps RMR objects onto IBA Memory Windows. The uDAPL API for binding an LMR to a RMR is *dat_rmr_bind()*. Among this functions parameters is a DAT_LMR_CONTEXT value (this is actually a member of the DAT_LMR_TRIPLET structure). As described in the IBA Specification, the Bind Memory Window verb takes both an L_KEY and Memory Region Handle among other parameters. Therefore the dictionary data structure described above must be used to map a DAT_LMR_CONTEXT (L_KEY) value to a DAPL_LMR so that the needed Memory Region handle can be retrieved. Binding a RMR to an LMR produces a DAT_RMR_CONTEXT. DAT_RMR_CONTEXT values are mapped to IBA R_KEYs.

G. Future Functionality and Optimizations

The uDAPL reference implementation has been provided in a public repository to enable vendors to adapt this work to their RDMA capable devices. The reference implementation is intended for use on multiple operating systems, multiple Verbs interfaces, and multiple chip sets. As such, it must use generic and portable techniques that may not always yield the highest performance implementation.

Part of the reference implementation includes a documentation repository of design notes, specifications, and a document advising vendors of functional elements that are best implemented in a driver and uDAPL optimizations possible by OS or chip specific functionality which has been deemed undesirable for the reference implementation. We divide functionality changes into two categories

- Areas in which functionality is lacking in the reference implementation.
- Areas in which the functionality is present in the reference implementation, but needs improvement.

We divide performance improvements into three types:

- Reducing context switches
- Reducing copying of data[1]
- Eliminating subroutine calls

An area of missing functionality in the reference implementation bears mentioning here.

[1] Note that the data referred to in "reducing copying of data" is the meta data describing an operation (e.g. scatter/gather list or event information), not the actual data to be transferred. No data transfer copies are required within the uDAPL reference implementation.

The uDAPL share memory model can be characterized as a peer-to-peer model since the order in which consumers register a region is not dictated by the programming interface. When creating a LMR, uDAPL consumers may share registration resources by setting the memory type parameter to DAT_MEM_TYPE_SHARED_VIRTUAL and specifying a shared memory identifier in the region description. If a region with the given identifier does not exist, one will be created. Otherwise, the resources used by the existing region will be shared with the new registration.

In contrast, the IBA shared memory interface requires the shared region to first be registered using the standard memory registration verb. All subsequent registrations must use the shared memory registration verb, and provide to that verb the memory region handle returned from the initial call. This means that the first process to register the memory must communicate the memory region handle it receives to all the other processes that wish to share this memory. This is a master-slave model of shared memory registration; the initial process (the master), is unique in its role, and it must tell the slaves how to register the memory after it.

To translate between these two models, uDAPL implementations are required to determine the first registration of a shared region and map from a shared memory identifier to a memory region handle across processes. To satisfy these requirements, uDAPL must maintain a system wide database to store this information. Since multiple processes will concurrently access the database, inter-process synchronization mechanisms are needed to protect its integrity. If a process exits abnormally, its memory regions will be deregistered. The database must be informed of these occurrences and update any relevant entries.

The appropriate place to store this information is in the IA hardware's associated device driver. As part of the operating system kernel, the driver is well positioned to arbitrate access to the database between multiple processes. The driver can also easily track a the existence of processes (a "close()" will be received when a process exits). Since the reference implementation was able to implement all other features in user space, we decided to leave the implementation of shared memory to the adaptor vendors.

IV. SourceForge.net Development

This was the authors' first experience setting up and developing a project on SourceForge.net, the host of many popular open source projects. Developing on SourceForge.net has been a worthwhile experience and has clearly benefited a large number of individuals and companies.

One of the stated goals of the DAT Collaborative was to provide an interface that would be freely available to anyone who wanted it and useable in any application or product without intellectual property concerns or other encumbrances. The reference implementation is distributed under the IBM

Common Public License (CPL) to provide the most flexibility for those who wish to incorporate the reference implementation in their own work. Since the entire project can be downloaded anonymously at any time, there is no record of how many people have looked at the code. However, we have received emails from fifteen different institutions that are exploring or implementing software based on the reference implementation. These emails range from questions on design and implementation to bug fixes and code snippets, which project members have happily incorporated. The combination of an amenable licensing policy and transparent development environment has contributed to uDAPL's popularity.

V. Performance Measurements

Measurements were conducted to gauge the level of performance achievable using the uDAPL reference implementation and for comparison to a native Verbs implementation. We choose to measure the following performance characteristics:

- CPU utilization: The amount of time necessary to post a DTO to the hardware and process the corresponding completion from the hardware. (units: time/operation)
- operation throughput per physical port: The number of operations that can be executed in a given amount of time. (units: operations/time)
- bandwidth per physical port: The amount of data that can be transferred on a single port in a given amount of time. (units: data/time)
- latency: The period of time between posting a DTO and posting the corresponding completion to an EVD. (units: time)

Two tools were developed to support this effort: one operating at the uDAPL API level and another operating at the Mellanox Verbs API level. Both tools were capable of measuring all of the performance metrics described above with the exception of operation throughput. In this case, only the uDAPL test tool was able to gather these measurements. The test programs were slightly different in other ways due to semantic differences between the uDAPL API and the Mellanox Verbs API. Despite these difference, comparisons between these measurements are still valid because each tool defined the performance criteria in the same way.

The tests were conducted on

- Two SuperMicro machines each with a ServerWorks Grand Champion LE chipset, 1.8 Ghz Intel Xeon CPU, and 512 MB RAM
- Two 4x Mellanox HCAs (one in each machine) connected point to point
- RedHat 7.3 2.4.18-10 UP kernel

The results are shown in the figures below.

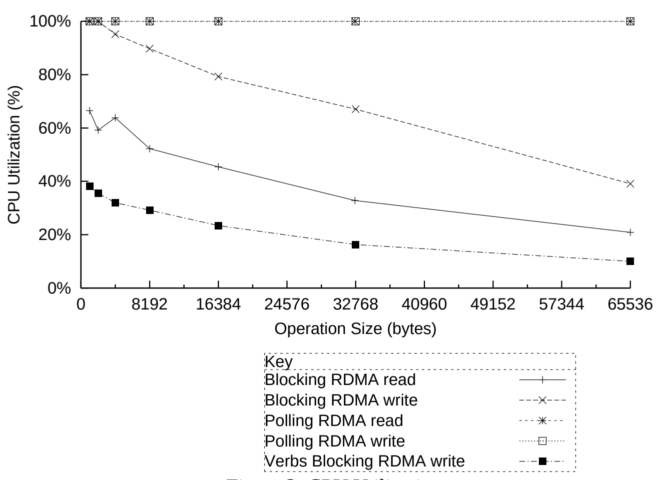

Figure 3: CPU Utilization

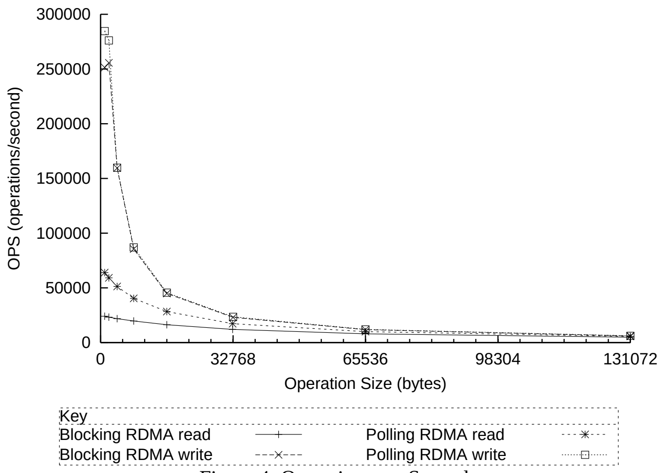

Figure 4: Operations per Second

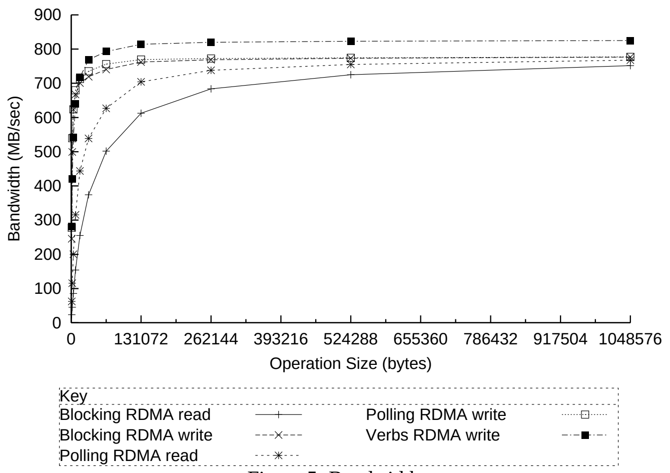

Figure 5: Bandwidth

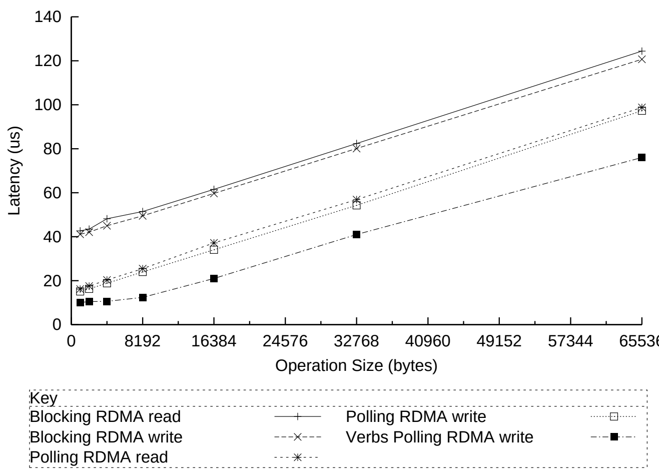

Figure 6: Latency

These results show that the uDAPL reference implementation does add some overhead to a native Verbs interface. However, the overhead is relatively small and we believe that if the uDAPL API was implemented directly on the RDMA hardware as described above, this overhead would be eliminated.

VI. CONCLUSION

The uDAPL API is a vendor independent interface for RDMA transports. The design of the uDAPL reference implementation provides interesting lessons on how to implement a generic API for RDMA networks.

ACKNOWLEDGMENT

The authors would like to thank Arkady Kanevsky and Anthony Topper for their suggestions and feedback. We would also like to thank Philip Christopher, Randy Pafford, Dave Mitchell, and Mark Natale for assisting with the uDAPL implementation.

REFERENCES

[1] H. J. Chu. "Zero-Copy TCP in Solaris," in *Proc. of the USENIX Technical Conference, San Diego, CA*, January 1996.

[2] Compaq, Intel, Microsoft. *Virtual Interface Architecture Specification, Version 1.0*. December 1997.

[3] DAT Collaborative. *kDAPL: Kernel Direct Access Programming Library, Version 1.0*. June 2002. [Online] Available: http://www.datcollaborative.org/kDAPL_062102.pdf

[4] DAT Collaborative. *uDAPL: User Direct Access Programming Library, Version 1.0*. June 2002. [Online] Available: http://www.datcollaborative.org/uDAPL_062102.pdf

[5] IBM. Access API. October 2002. [Online] Available: http://www.datcollaborative.org/ibm.html

[6] InfiniBand Trade Association. *InfiniBand Architecture Specification, Release 1.1*, November 2002.

[7] J.-S. Kim, K. Kim, S.I. Jung. "Building a High Performance Communication Layer over Virtual Interface Architecture on Linux Clusters," *Proc. of the 15th ACM International Conference on Supercomputing (ICS)*, pp.335-347, June 2001.

[8] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, D. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. "Structure and Performance of the Direct Access File System (DAFS)," in *Proc. of the USENIX Technical Conference, Monterrey, CA*, June 2002.

[9] Mellanox. Verbs API. July 2002. [Online] Available: http://www.datcollaborative.org/mellanox.html

[10] M. Rangarajan, L. Iftode. "Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance," in *Proc. of 4th Annual Linux Conference, Atlanta, GA*, October 2000.

[11] RDDP Working Group. "Remote Direct Data Placement Charter," Internet Engineering Task Force. December 2002. [Online] Available: http://www.ietf.org/html.charters/rddp-charter.html

[12] R. Recio, P. Culley, D. Garcia, J. Hilland. *An RDMA Protocol Specification (Version 1.0)*. October 2002. [Online]. Available: http://www.rdmaconsortium.org/home/draft-recio-iwarp-rdmap-v1.0.pdf

[13] H. Shah, J. Pinkerton, R. Recio, P. Culley. *Direct Data Placement over Reliable Transports*. October 2002. [Online] Available: http://www.rdmaconsortium.org/home/draft-shah-iwarp-ddp-v1.0.pdf

VII. APPENDIX A-UDAPL API V1.0

```
DAT_RETURN
dat_ia_open(
   IN const DAT_NAME_PTR             ia_name_ptr,
   IN       DAT_COUNT                async_evd_min_qlen,
   INOUT    DAT_EVD_HANDLE           *async_evd_handle,
   OUT      DAT_IA_HANDLE            *ia_handle );

DAT_RETURN
dat_ia_close(
   IN    DAT_IA_HANDLE               ia_handle,
   IN    DAT_CLOSE_FLAGS             ia_flags );

DAT_RETURN
dat_ia_query(
   IN    DAT_IA_HANDLE               ia_handle,
   OUT   DAT_EVD_HANDLE              *async_evd_handle,
   IN    DAT_IA_ATTR_MASK            ia_attr_mask,
   OUT   DAT_IA_ATTR                 ia_attributes,
   IN    DAT_PROVIDER_ATTR_MASK      provider_attr_mask,
   OUT   DAT_PROVIDER_ATTR           provider_attributes );

DAT_RETURN
dat_set_consumer_context(
   IN    DAT_HANDLE                  dat_handle,
   IN    DAT_CONTEXT                 context );

DAT_RETURN
dat_get_consumer_context(
   IN    DAT_HANDLE                  dat_handle,
   OUT   DAT_CONTEXT                 *context );

DAT_RETURN
dat_get_handle_type(
   IN    DAT_HANDLE                  dat_handle,
   OUT   DAT_HANDLE_TYPE             *handle_type );

DAT_RETURN
dat_cno_create(
   IN    DAT_IA_HANDLE               ia_handle,
   IN    DAT_OS_WAIT_PROXY_AGENT agent,
   OUT   DAT_CNO_HANDLE              *cno_handle );

DAT_RETURN
dat_cno_free(
   IN    DAT_CNO_HANDLE              cno_handle );

DAT_RETURN
dat_cno_wait(
   IN    DAT_CNO_HANDLE              cno_handle,
   IN    DAT_TIMEOUT                 timeout,
   OUT   DAT_EVD_HANDLE              *evd_handle );

DAT_RETURN
dat_cno_modify_agent(
    IN     DAT_CNO_HANDLE               cno_handle,
    IN     DAT_OS_WAIT_PROXY_AGENT      agent );

DAT_RETURN
dat_cno_query(
    IN     DAT_CNO_HANDLE               cno_handle,
    IN     DAT_CNO_PARAM_MASK           cno_param_mask,
    OUT    DAT_CNO_PARAM                *cno_param );

DAT_RETURN
dat_evd_create(
    IN     DAT_IA_HANDLE                ia_handle,
    IN     DAT_COUNT                    evd_min_qlen,
    IN     DAT_CNO_HANDLE               cno_handle,
    IN     DAT_EVD_FLAGS                evd_flags,
    OUT    DAT_EVD_HANDLE               *evd_handle );

DAT_RETURN
dat_evd_free(
    IN     DAT_EVD_HANDLE               evd_handle );

DAT_RETURN
dat_evd_query(
    IN     DAT_EVD_HANDLE               evd_handle,
    IN     DAT_EVD_PARAM_MASK           evd_param_mask,
    OUT    DAT_EVD_PARAM                *evd_param );

DAT_RETURN
dat_evd_modify_cno(
    IN     DAT_EVD_HANDLE               evd_handle,
    IN     DAT_CNO_HANDLE               cno_handle );

DAT_RETURN
dat_evd_enable (
    IN     DAT_EVD_HANDLE               evd_handle );

DAT_RETURN
dat_evd_disable (
    IN     DAT_EVD_HANDLE               evd_handle );

DAT_RETURN
dat_evd_resize (
    IN     DAT_EVD_HANDLE               evd_handle,
    IN     DAT_COUNT                    evd_min_qlen );

DAT_RETURN
dat_evd_wait (
    IN     DAT_EVD_HANDLE               evd_handle,
    IN     DAT_TIMEOUT                  timeout,
    IN     DAT_COUNT                    threshold,
    OUT    DAT_EVENT                    *event,
    OUT    DAT_COUNT                    *nmore );

DAT_RETURN
dat_evd_dequeued(
    IN     DAT_EVD_HANDLE               evd_handle,
    OUT    DAT_EVENT                    *event );

DAT_RETURN
dat_evd_post_se(
    IN     DAT_EVD_HANDLE               evd_handle,
    IN const DAT_EVENT                  *event );

DAT_RETURN
dat_psp_create(
    IN     DAT_IA_HANDLE                ia_handle,
    IN     DAT_CONN_QUAL                conn_qual,
    IN     DAT_EVD_HANDLE               evd_handle,
    IN     DAT_PSP_FLAGS                psp_flags,
    OUT    DAT_PSP_HANDLE               *psp_handle );

DAT_RETURN
dat_psp_free(
    IN     DAT_PSP_HANDLE               *psp_handle );

DAT_RETURN
dat_psp_query(
    IN     DAT_PSP_HANDLE               *psp_handle,
    IN     DAT_PSP_PARAM_MASK           psp_param_mask,
    OUT    DAT_PSP_PARAM                *psp_param );

DAT_RETURN
dat_rsp_create(
    IN     DAT_IA_HANDLE                ia_handle,
    IN     DAT_CONN_QUAL                conn_qual,
    IN     DAT_EP_HANDLE                ep_handle,
    IN     DAT_EVD_HANDLE               evd_handle,
    OUT    DAT_RSP_HANDLE               *rsp_handle );

DAT_RETURN
dat_rsp_free(
    IN     DAT_RSP_HANDLE               rsp_handle );

DAT_RETURN
dat_rsp_query(
    IN     DAT_RSP_HANDLE               rsp_handle,
    IN     DAT_RSP_PARAM_MASK           rsp_param_mask,
    OUT    DAT_RSP_PARAM                *rsp_param );

DAT_RETURN
dat_cr_query(
    IN     DAT_CR_HANDLE                cr_handle,
    IN     DAT_CR_PARAM_MASK            cr_param_mask,
    OUT    DAT_CR_PARAM                 *cr_param );

DAT_RETURN
dat_cr_accept(
    IN     DAT_CR_HANDLE                cr_handle,
    IN     DAT_EP_HANDLE                ep_handle,
    IN     DAT_COUNT                    private_data_size,
    IN const DAT_PVOID                  private_data );

DAT_RETURN
dat_cr_reject(
    IN     DAT_CR_HANDLE                cr_handle );

DAT_RETURN
dat_cr_handoff(
    IN     DAT_CR_HANDLE                cr_handle,
    IN     DAT_CONN_QUAL                handoff );

DAT_RETURN
dat_ep_create(
    IN     DAT_IA_HANDLE                ia_handle,
    IN     DAT_PZ_HANDLE                pz_handle,
    IN     DAT_EVD_HANDLE               recv_evd_handle,
    IN     DAT_EVD_HANDLE               request_evd_handle,
    IN     DAT_EVD_HANDLE               connect_evd_handle,
    IN     DAT_EVD_HANDLE               rmr_bind_evd_handle,
    IN     DAT_EP_ATTR                  *ep_attributes,
    OUT    DAT_EP_HANDLE                *ep_handle );

DAT_RETURN
dat_ep_free(
    IN     DAT_EP_HANDLE                ep_handle );

DAT_RETURN
dat_ep_get_status(
    IN     DAT_EP_HANDLE                ep_handle,
    OUT    DAT_EP_STATE                 *ep_state,
    OUT DAT_BOOLEAN                 *recv_idle,
    OUT DAT_BOOLEAN                 *request_idle );

DAT_RETURN
dat_ep_query(
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_EP_PARAM_MASK        ep_param_mask,
    OUT DAT_EP_PARAM                *ep_param );

DAT_RETURN
dat_ep_modify(
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_EP_PARAM_MASK        ep_param_mask,
    IN     DAT_EP_PARAM             *ep_param );

DAT_RETURN
dat_ep_connect(
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_IA_ADDRESS_PTR       remote_ia_address,
    IN     DAT_CONN_QUAL            remote_conn_qual,
    IN     DAT_TIMEOUT              timeout,
    IN     DAT_COUNT                private_data_size,
    IN const DAT_PVOID              private_data,
    IN     DAT_QOS                  qos,
    IN     DAT_CONNECT_FLAGS        connect_flags );

DAT_RETURN
dat_ep_dup_connect(
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_EP_HANDLE            dup_ep_handle,
    IN     DAT_TIMEOUT              timeout,
    IN     DAT_COUNT                private_data_size,
    IN const DAT_PVOID              private_data,
    IN     DAT_QOS                  qos );

DAT_RETURN
dat_ep_disconnect(
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_CLOSE_FLAGS          disconnect_flags );

DAT_RETURN
dat_ep_post_send(
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_COUNT                num_segments,
    IN     DAT_LMR_TRIPLET          *local_iov,
    IN     DAT_DTO_COOKIE           user_cookie,
    IN     DAT_COMPLERTION_FLAGS    completion_flags );

DAT_RETURN
dat_ep_post_recv(
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_COUNT                num_segments,
    IN     DAT_LMR_TRIPLET          *local_iov,
    IN     DAT_DTO_COOKIE           user_cookie,
    IN     DAT_COMPLERTION_FLAGS    completion_flags );

DAT_RETURN
dat_ep_post_rdma_read(
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_COUNT                num_segments,
    IN     DAT_LMR_TRIPLET          *local_iov,
    IN     DAT_DTO_COOKIE           user_cookie,
    IN     DAT_RMR_TRIPLET          *remote_buffer,
    IN     DAT_COMPLERTION_FLAGS    completion_flags );

DAT_RETURN
dat_ep_post_rdma_write(
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_COUNT                num_segments,
    IN     DAT_LMR_TRIPLET          *local_iov,
    IN     DAT_DTO_COOKIE           user_cookie,
    IN     DAT_RMR_TRIPLET          *remote_buffer,
    IN     DAT_COMPLERTION_FLAGS    completion_flags );

DAT_RETURN
dat_pz_create(
    IN     DAT_IA_HANDLE            ia_handle,
    OUT DAT_PZ_HANDLE               *pz_handle );

DAT_RETURN
dat_pz_free(
    IN     DAT_PZ_HANDLE            pz_handle );

DAT_RETURN
dat_pz_query(
    IN     DAT_PZ_HANDLE            pz_handle,
    IN     DAT_PZ_PARAM_MASK        pz_param_mask,
    OUT DAT_PZ_PARAM                *pz_param );

DAT_RETURN
dat_lmr_create(
    IN     DAT_IA_HANDLE            ia_handle,
    IN     DAT_MEM_TYPE             mem_type,
    IN     DAT_REGION_DESCRIPTION   region_description,
    IN     DAT_VLEN                 length,
    IN     DAT_PZ_HANDLE            pz_handle,
    IN     DAT_MEM_PRIV_FLAGS       mem_privileges,
    OUT DAT_LMR_HANDLE              *lmr_handle,
    OUT DAT_LMR_CONTEXT             *lmr_context,
    OUT DAT_VLEN                    *registered_size,
    OUT DAT_VADDR                   *registered_address );

DAT_RETURN
dat_lmr_free(
    IN     DAT_LRM_HANDLE           lmr_handle );

DAT_RETURN
dat_lmr_query(
    IN     DAT_LRM_HANDLE           lmr_handle,
    IN     DAT_LMR_PARAM_MASK       lmr_param_mask,
    OUT DAT_LMR_PARAM               *lmr_param );

DAT_RETURN
dat_rmr_create(
    IN     DAT_PZ_HANDLE            pz_handle,
    OUT DAT_RMR_HANDLE              *rmr_handle );

DAT_RETURN
dat_rmr_free(
    IN DAT_RMR_HANDLE               rmr_handle );

DAT_RETURN
dat_rmr_query(
    IN     DAT_RMR_HANDLE           rmr_handle,
    IN     DAT_RMR_PARAM_MASK       rmr_param_mask,
    OUT DAT_RMR_PARAM               *rmr_param );

DAT_RETURN
dat_rmr_bind(
    IN     DAT_RMR_HANDLE           rmr_handle,
    IN     DAT_LMR_TRIPLET          *lmr_triplet,
    IN     DAT_MEM_PRIV_FLAGS       mem_privileges,
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_RMR_COOKIE           user_cookie,
    IN     DAT_COMPLETION_FLAGS     completion_flags,
    OUT DAT_RMR_CONTEXT             *rmr_context );
```