IBA Software Architecture
uDAPL
High Level Design

Draft 2

August 2002

Revision History and Disclaimers

Rev.	Date	Notes
Draft 1	July 2002	Internal review.
Draft 2	August 2002	Feedback incorporated from internal review.

Abstract

The uDAPL over IB provides standardized user mode API over IBA fabrics as defined by the DAT Collaborative. Implemented as a standard Linux shared object, it interfaces to the IB – Access Layer. The uDAPL gains access the HCA and subnet management services through the Abstraction Layer. All uDAPL data transfers use the reliable connection service.

The primary responsibilities of the uDAPL library are performing name to address translation, establish connection and transfer the data reliably.

Contents

Figures

1. Introduction

1.1 Purpose and Scope

This HLD defines the implementation of all uDAPL components described in the "uDAPL *Specification*", including inter-component dependencies, and provides sufficient design detail that will satisfy the product requirements as specified.

1.2 Audience

Anyone interested in understanding this implementation of the Architecture Specification should read this document, including:

- Software developers who are integrating the separate modules into their own software projects
- Hardware developers who need an understanding of the software behavior to optimize their designs
- Evaluation engineers who are developing tests for Infiniband-compliant devices
- Others in similar roles who need more than a basic understanding of the software

1.3 Acronyms and Terms

DAT:	Direct Access Transport
DAPL:	Direct Access Providers Library
SDP:	Sockets Direct Protocol (A Socket emulation protocol specified for Infiniband)
TOE:	TCP Offload Engine (Hardware that supports offloading TCP/IP protocol from host)
IPoIB:	IP-over-Infiniband (and IETF defined RFC to send IP packets on Infiniband fabric)
IBA:	Infiniband Architecture
CNO	Consumer Notification Object
EVD	Event Dispatcher
DTO	Data Transfer Operation
LMR	Local Memory Region
RMR	Remote Memory Region

1.4 References

UDAPL

User-mode Direct Access Providers Library Version 1.0

Infiniband

Infiniband Architecture Specification, Version 1.0a, http://www.infinibandta.org/

IP over IB IETF draft: http://www.ietf.org/ids.by.wg/ipoib.html

Infiniband Specification Annex A4 - Sockets Direct Protocol (SDP), Release 1.0.a

Device Drivers

Rubini, Alessandro and Corbet, Johathan. Linux Device Drivers Book, 2nd Edition: O'reilly, June 2001. ISBN: 0-59600-008-1. http://www.xml.com/ldd/chapter/book/

1.5 Conventions

This document uses the following typographical conventions and icons:

Italic	is used for book titles, manual titles, URLs, and new terms.
Bold	is used for user input (in the Installation section).
`Fixed width`	is used for code definitions, data structures, function definitions, and system console output. Fixed width text is always in Courier font.

NOTE

Is used to alert you to an item of special interest.

DESIGN ISSUE

Is used to alert you to unresolved design issues that may impact the module's design, function, or usage.

1.6 Before You Begin

Please note the following:

This document assumes that you are familiar with the *Infiniband Architecture Specification*, which is available from the Infiniband Trade Association at http://www.infinibandta.org.

2. Design Overview

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

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

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

UDAPL is intended to support following type of application

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

2.1 Requirement for uDAPL

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

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

2.2 System Structural Overview

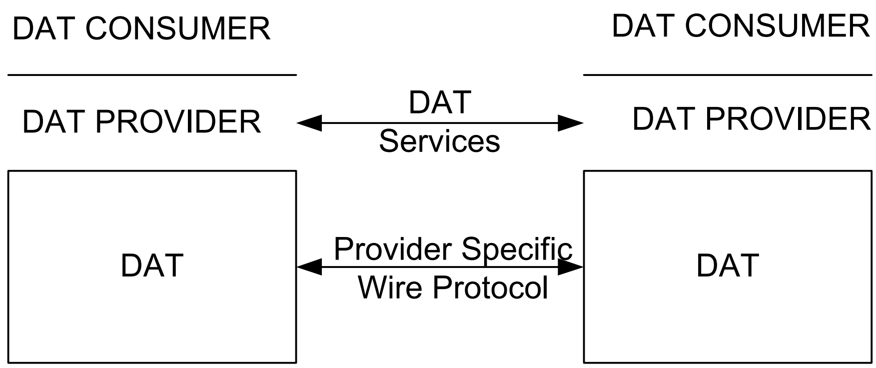

Figure 1 uDAPL Model

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

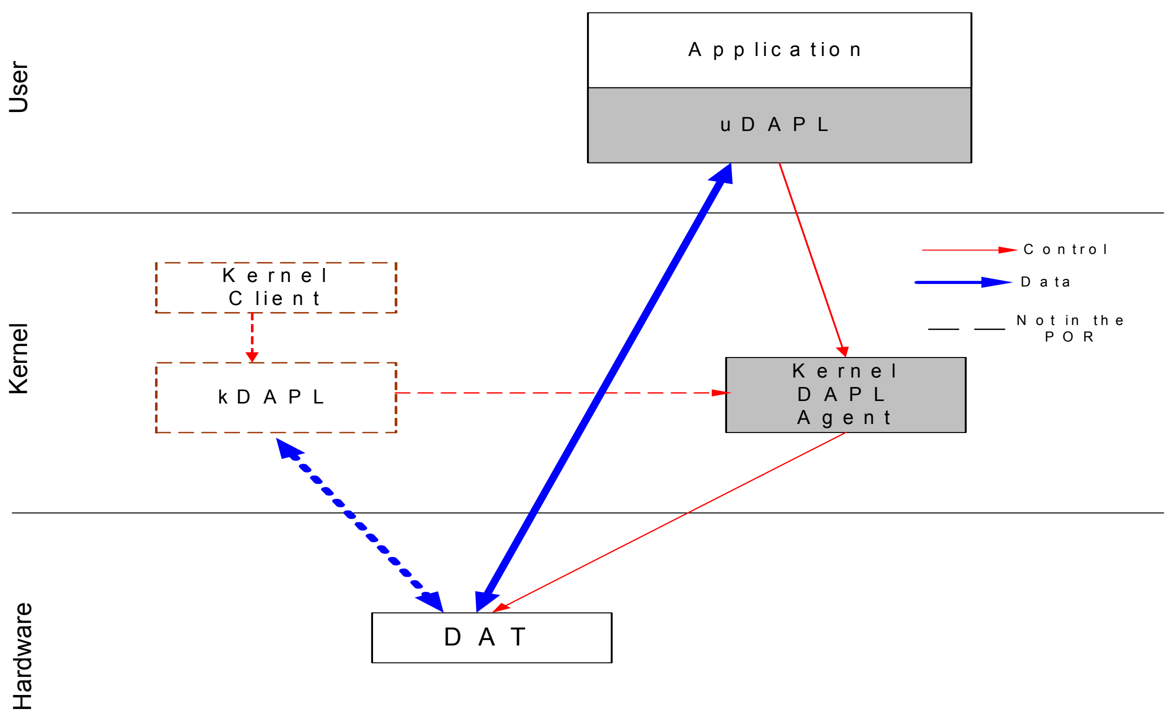

Figure 2 uDAPL Overview

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

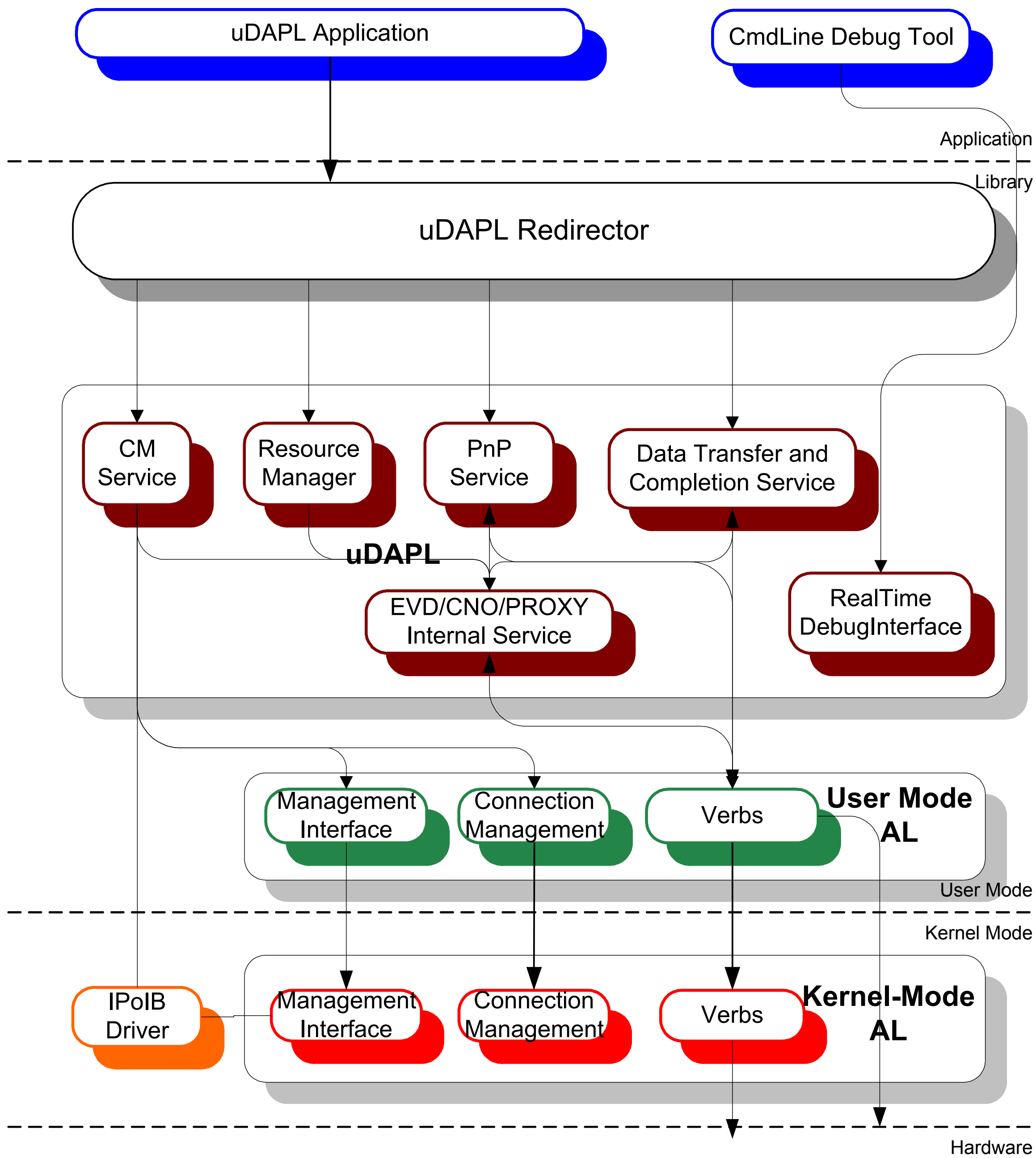

Figure 3 uDAPL Components

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

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

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

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

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

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

3. Design Details

This document will deal with design of only uDAPL library and uDAPL redirection mechanism is described in Appendix-B of the uDAPL specification.

3.1 Resource Manager

Following are the main uDAPL resources

1. Event Dispatcher
2. Consumer Notification Object
3. End Point
4. Service Point
5. Local Memory Region & Remote Memory Region
6. Protection Zone

Resource Manager creates & destroys these resources on demand from consumers. UDAPL resources are combination of uDAPL private data structure & IB resources such as QP, CQ, PD and TPT etc. To enable resource manager & other services handle DAT objects properly, each DAT object consist of

1. Doubly link list pointers for resource management & locating the resource
2. Spin lock for thread safe operation
3. Object type Identifier
4. State of the Object

Following is the mapping of uDAPL resources with respect to IB access layer Resources.

UDAPL Resource	IB/Access Layer Resource
Event Dispatcher	CM Callback hander, CQ callback hander, Error Handler & CQ
Consumer Notification Object	Wait Object
End Point	Queue Pair
Service Point	Service ID
Local Memory Region & Remote Memory Region	Memory Region & Memory Window
Protection Zone	Protection Domain
Interface Adapter	IB Port

A detail about each of these resources is described in the following sub-sections.

3.1.1 Interface Adapter

UDAPL Interface Adapter is mapped to IB port. All the DAT resource created belongs to the interface adapter. Interface Adapter (IA) has following main components.

1. Reference to Access Layer's CA handle
2. CA GUID
3. Port GUID
4. Err Callback handler
5. Spin Lock
6. Doubly linked List of EP/SP/EVD
7. Etc

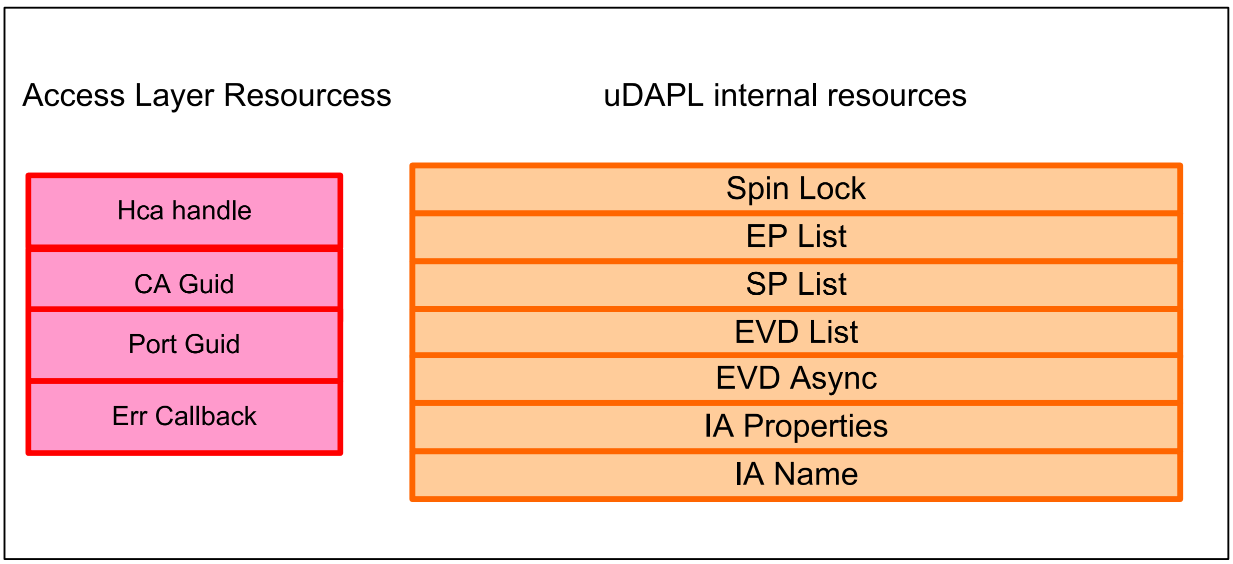

Figure 4 Major Components of IA

3.1.2 Event Dispatcher

The event dispatcher is the prime mover of the uDAPL. Event dispatcher is implemented using IB Access's layer's callback Handler & Event dispatcher structure. So Event Dispatcher is primarily consist of

1. Callback handler
    a. Completion Queue Callback Handler
    b. Connection Manager Callback handler
    c. Timer Callback handler

   d. Error Callback handler

2. IB completion queue
3. FIFO for Software Events
4. Timer for Timeout
5. Maintenance information

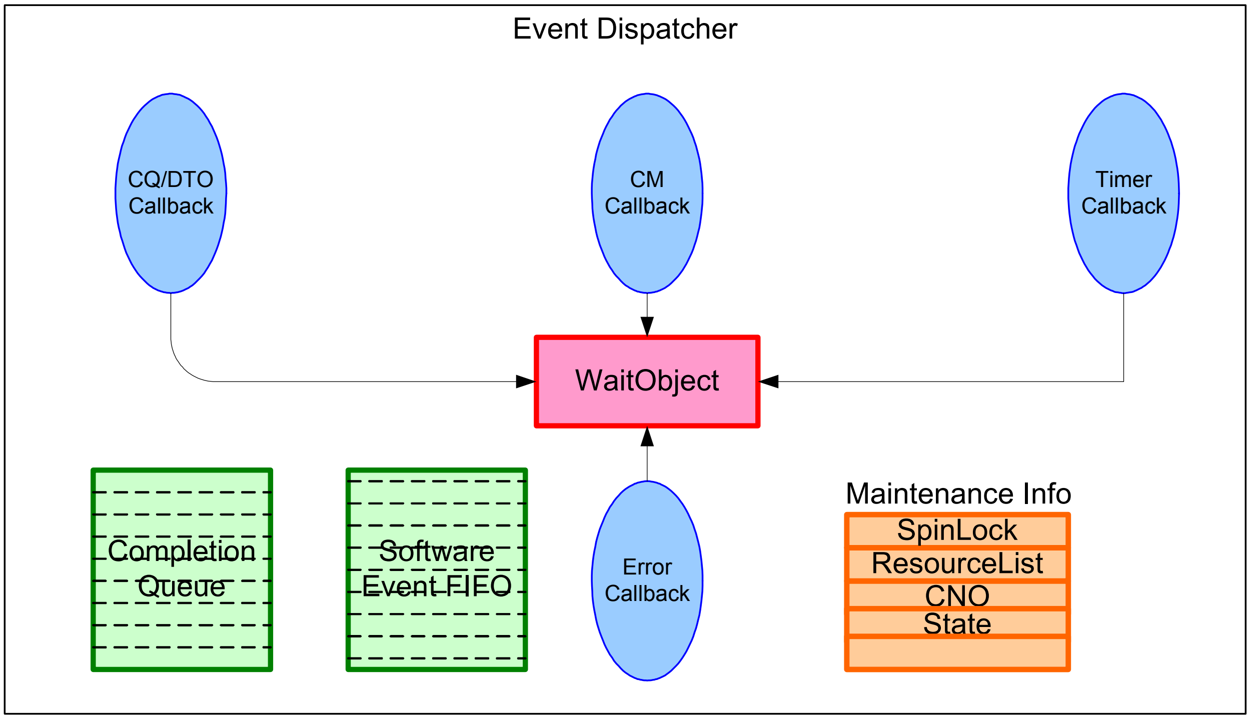

Figure 5 Major Components of EVD

To optimize IB resources, CQ is created only if DTO flag is set. If no DTO flag is set this EVD won't be used for DTO completion and hence no CQ is required.

Also software Event FIFO is created only on demand, which is indicated by EVD flag.

In addition to above components resource manager also creates the synchronization object (spin lock) etc for thread safe operation and maintenance. Refer to EVD structure at the end of this document.

3.1.3 Consumer Notification Object

When consumer wants to wait on multiple event dispatchers simultaneously, same CNO is associated with all Event Dispatcher.

The Consumer Notification Object basically consists of CNO WaitObject, Timer, optional proxy agent & other maintenance objects. Once CNO is created, it is doubly link listed with Interface adapter context for maintenance purpose.

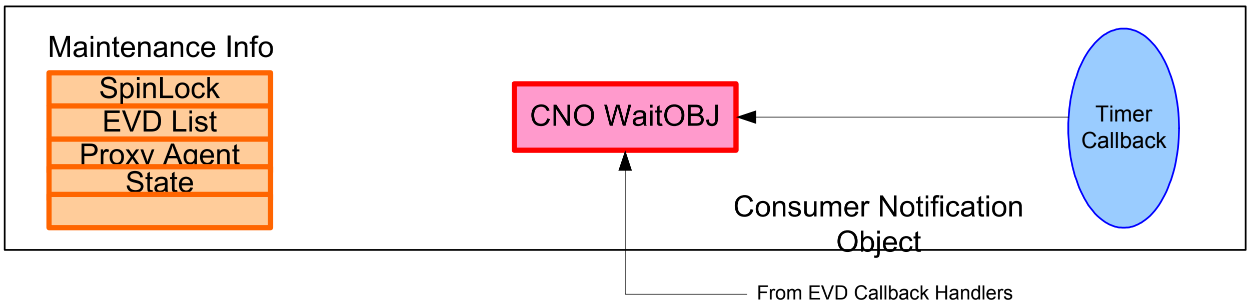

Figure 6 Major Components of CNO

3.1.4 EndPoint

The endpoint supports Data transfer operation and whose completions are posted to specified event dispatchers.

UDAPL endpoint is combination of

1. IB queue pair,
2. Association to Tx, Rx, Connection & Bind Event Dispatcher
3. Other maintenance data structure such as sate, spinlock etc

Queue pair is not created during endpoint creation but delayed until it is actually required i.e., during connection establishment. Endpoint also hides various IB specific QP states including QP TimeWait state.

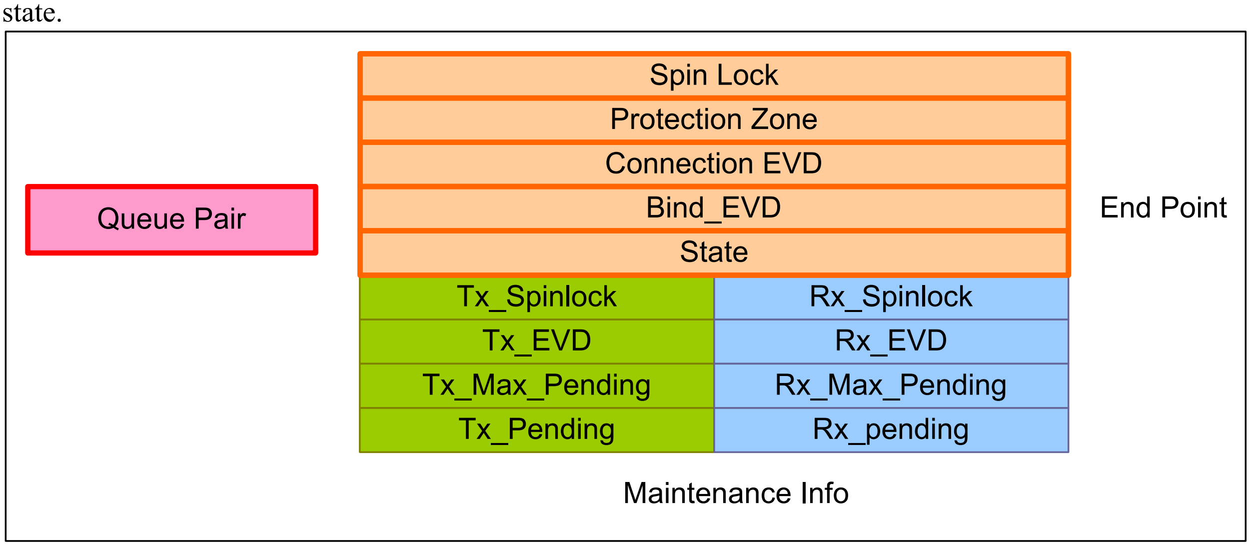

Figure 7 Major Components of EP

3.1.5 Local Memory Region & Remote Memory Region

Local memory region is an arbitrarily sized, virtually contiguous area of memory in the consumer's address space that was registered, enabling Interface Adapter local access and, optionally, remote access.

Registering the memory involves locking down the memory, creating LKEY (LMR context), RKEY (RMR context) and TPT etc.

UDAPL invokes Access layer to do this straightforward resource creation. The only subtle point to be noted is for registration DAT_MEM_TYPE_SHARED_VIRTUAL uDAPL invokes ib_reg_shmid( ) and for other types it uses ib_reg_mem.

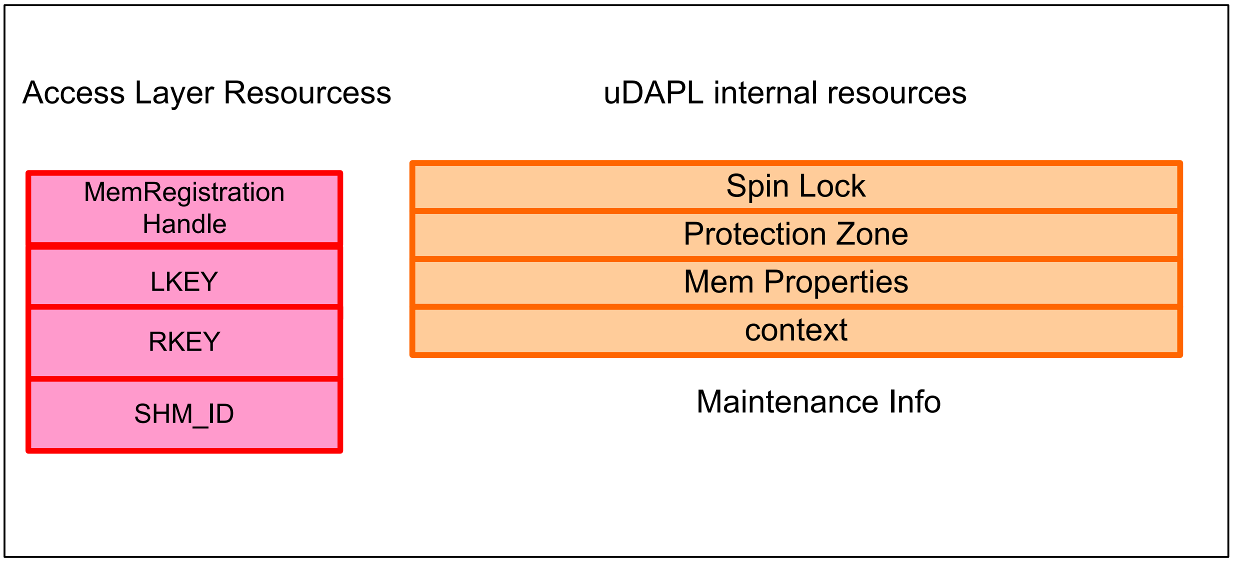

Figure 8 Major Components of LMR

IB Memory window is mapped uDAPL Remote Memory Region (RMR). Creating RMR is basically creation of RMR Context for given Protection Zone. This newly created RMR is not bound to any specific LMR until it is specifically bound using dat_rmr_bind. So RMR create return RMR handle that refers to following info in addition to standard maintenance info.

1. LKEY
2. RKEY
3. Protection Domain
4. EndPoint (after bind is done)
5. Etc

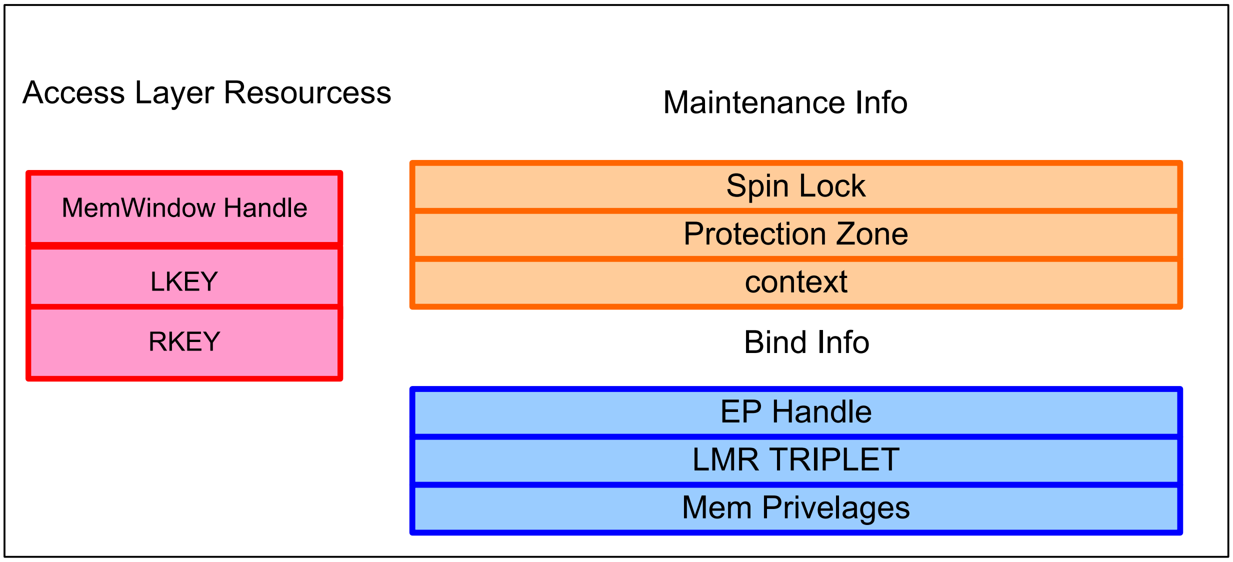

Figure 9 Major Components of RMR

3.2 CM Service

UDAPL Connection manager is based on implied two-way handshake mechanism. This poses no serious problem while implementing over Infiniband connection manager, which is based on three-way handshake mechanism.

However uDAPL CM needs to address connection qualifier mapping, Address translation etc.

3.2.1 Connection Qualifier

Connection Qualifier is mapped to Infiniband Service ID (SID). Since SIDs are used by all IB application, it is the responsibility of the application to make sure its service ID doesn't collide with other application's SID such as SDP. UDAPL directly translates ConnectionQualifier into SID and no effort is made to identify any collision.

3.2.2 Address Translation

UDAPL uses IP address to establish IB connection to take advantage existing name service such as DNS without any domain name import.

Since IP address is not a GID, uDAPL depends IPoIB driver's private ioctl interface to convert IP address to GID. Then uDAPL uses this GID to obtain pathrecord from access layer using ib_query( ) by IB_QUERY_PATH_REC_BY_GIDS.

Interface Adapter Address is mapped to IP address. In IPoIB, each PKEY/GID can be assigned an IP address and each IP address can have multiple aliases IP address. So each can have multiple IP address.

This multiple L3 address for IA poses a problem in identifying the connection request originated from which IA address. It is the responsibility of the consumer to exchange SourceIP address in dat_ep_connect( ) private data. Exact format & location of this information in the private data is application dependent and uDAPL will not decode it.

3.2.3 Connection Protocol

UDAPL supports active / passive connection method where one side takes active (client) mode and other takes passive (server mode).

Based on this active/passive method, uDAPL defines two models of connection establishment

1. Client / Server Connection similar to Socket (public service point). PSP will sink all matching connection request until it is explicitly freed which is similar to close (socket).
2. Client /Server Connection similar to VI. RSP will sink only one incoming connection and reject any further connection request.

UDAPL connection manager can be split into two

1. Passive Side State Machine
2. Active Side State Machine

Connection callback handlers drive both state machines. These callback handlers are invoked by Access Layer CM whenever IB CM message arrives or error or during message timeouts.

Following two sections describes each state machine in detail.

3.2.3.1 Passive Side

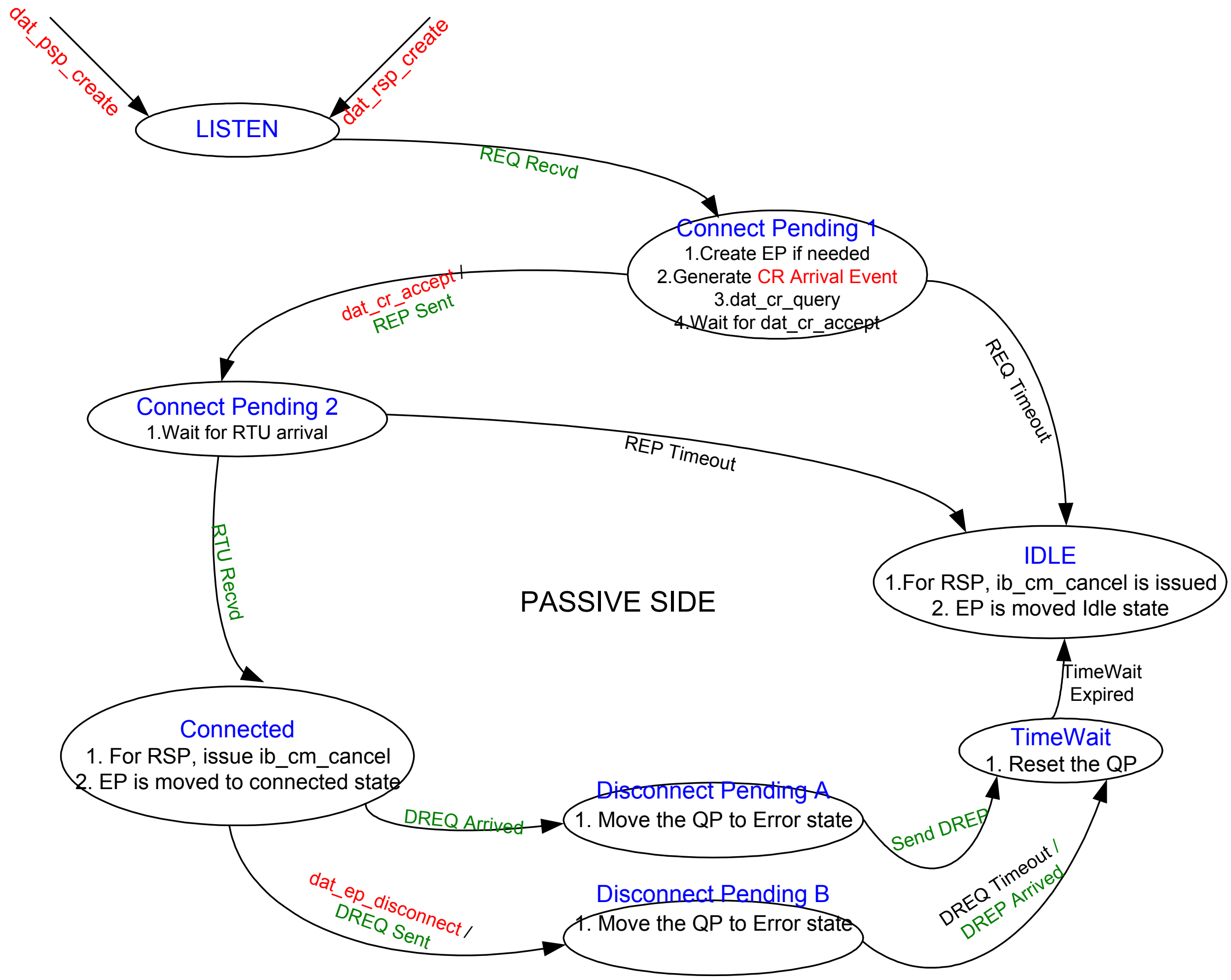

Figure 10 Passive Side States

Passive Side state machine can be also called as server side state machine and which can initiated by server using dat_psp_create ( ) or dat_rsp_create( ).

1. UDAPL maps dat_psp_create / dat_rsp_create to ib_cm_listen and puts the Service point into listen state. UDAPL specifies its internal passive side state machine handler as callback handler to ib_cm_listen_api( ). The size of EVD determines the backlog size for PSP . For RSP backlog size is always 1.
2. On the Arrival of the of the connection request Access Layer will invoke uDAPL callback handler which will move the service point to “Connection Pending 1” state and post CR arrival event to EVD. The handle may also decide to signal EVD wait object or CNO wait object or invoke proxy agent depending on the situation.
3. If uDAPL consumer accepts the connection by invoking dat_cr_accept, uDAPL issue ib_cm_accept( ) which generate REP message. Then EP/SP state will be moved “Connect Pending State” and dat_cr_accept will wait on the EP wait object.

4. When RTU message arrives or Connection Establishment event arrives, Access Layer will invoke the callback handler again. Callback handler will move the EP to “Connected” State and free the CR handle. After resource cleanup, dat_cr_accept will wakeup & return.
5. If any DREQ arrives any time, AL will invoke callback handler, which will move EP to “Disconnect Pending 1” state and move the QP to error state by invoking ib_modify_qp. This will flush any pending workrequest. EP will be move to “TimeWait” state
6. After sending DREP, EP will be moved to “idle” sate after passing through “TimeWait” state.
7. If Application invokes dat_ep_disconnect( ), EP will be moved “Disconnect Pending 2” state and DREQ message will be sent by invoking ib_cm_dreq( )
8. Once DREP message arrives or DREQ message times out, QP will be moved to error state to flush all the descriptors using ib_modify_qp( ).
9. EP will be moved to “idle” sate after passing through “TimeWait” state.

3.2.3.2 Active Side

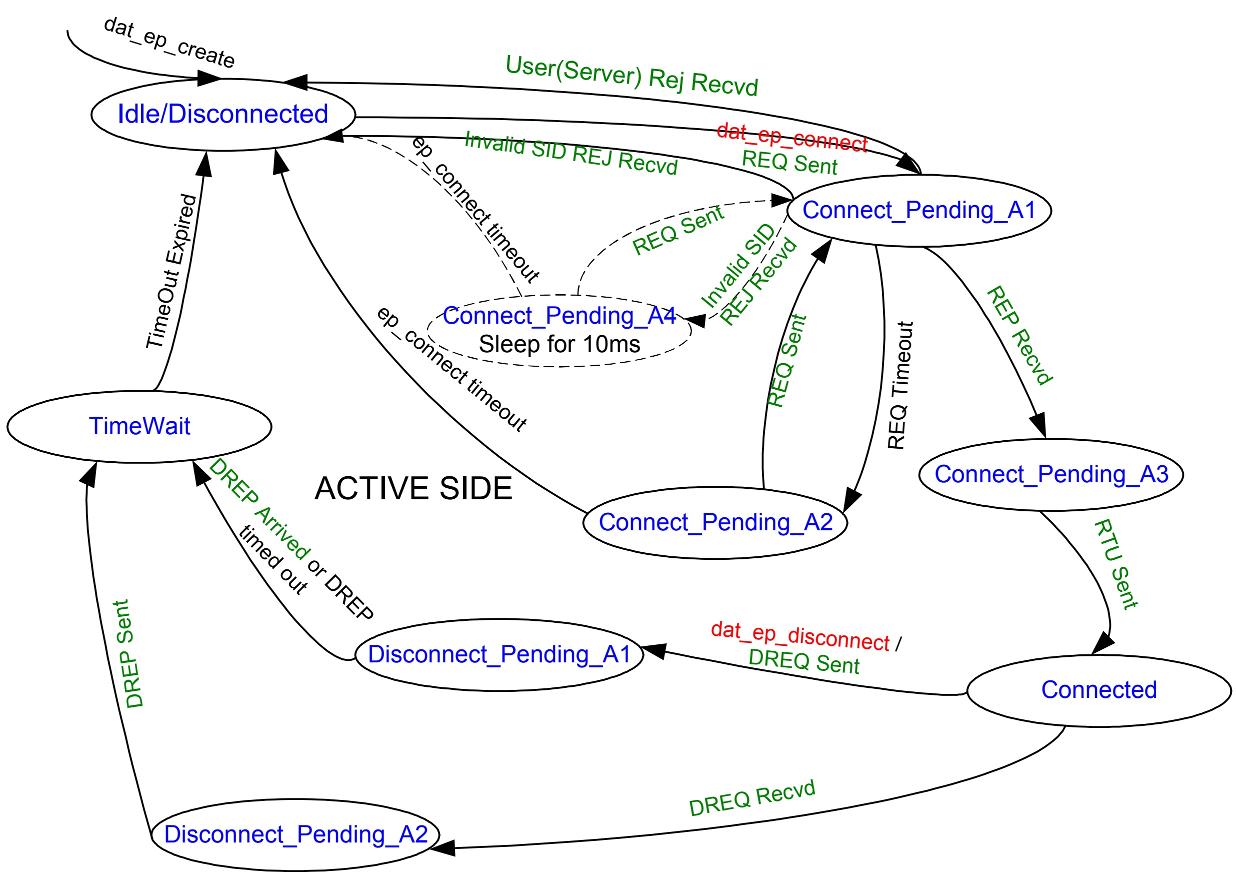

Figure 11 Active Side States

Active Side state machine can be also called as client side state machine and which can initiated by server using dat_ep_connect ().

1. UDAPL maps dat_ep_create to ib_create_qp and puts the Endpoint into idle state. UDAPL specifies its internal active side state machine handler as callback handler to ib_cm_req ( ).
2. When consumer invokes dat_ep_connect,
    a. remote address is converted to GID by invoking IPoIB driver ioctl interface
    b. Path record is obtained by invoking access layer using ib_query( ) by IB_QUERY_PATH_REC_BY_GIDS.
    c. Invoking ib_cm_req sends REQ message . The parameter for this API includes callback message for all CM messages & errors.
    d. EP is moved "Connect Pending A1" state.
    e. dat_ep_connect waits on WaitObj
3. If User(Server)REJ message is received, AL will invoke REJ callback which will move the EP back to "Disconnected" state and signal the WaitObject. Dat_ep_connect will wake up & return with error.
4. If REP message is received AL will invoke REP callback, which will move the QP to RTR state and move the EP to "Connect Pending A3" state.
5. After QP is moved to RTS state and RTU message is sent, EP is moved to connected state.
6. "Connect Pending 2" is retry state. UDAPL will attempt to establish connection until timeout occurs
7. "Connect Pending 4" is also optional retry state. In some implementation it is desirable for uDAPL to retry connection until server comes online. This should be build time option.
8. If any DREQ arrives any time, AL will invoke callback handler, which will move EP to "Disconnect Pending 1" state and move the QP to error state by invoking ib_modify_qp. This will flush any pending workrequest. EP will be move to "TimeWait" state
9. After sending DREP, EP will be moved to "idle" sate after passing through "TimeWait" state.
10. If Application invokes dat_ep_disconnect( ), EP will be moved "Disconnect Pending 2" state and DREQ message will be sent by invoking ib_cm_dreq( )
11. Once DREP message arrives or DREQ message times out , QP will be moved to error state to flush all the descriptors using ib_modify_qp( ).
12. EP will be moved to "idle" sate after passing through "TimeWait" state.

3.2.3.3 Connection Management Callback Handler

As described in above two sections, callback handler is the prime mover of uDAPL CM. This section describes the how callback handlers managed & how it is used to change the EP/SP state.

On Passive side

1. REQ callback is specified in ib_cm_listen( ) parameter.
2. RTU callback is specified in ib_cm_rep( ) parameter
3. DREQ callback is specified in ib_cm_rep( ) parameter

On Active side

1. REP callback is specified in ib_cm_req( ) parameter
2. REJ callback is specified in ib_cm_req( ) parameter
3. DREQ callback is specified in ib_cm_rtu( ) parameter

For easy representation , all these six callback handlers are mapped to single callback handler i.e, for example

```
void req_cb(ib_cm_req_rec_t *p_cm_req_rec )
{
        udapl_cm_callback(REQ_MSG,(void *)p_cm_req_rec);
}
```

dat_ep_connect / dat_ep_disconnect / dat_cr_accept / dat_cr_reject API works in tandem with these callbacks to endpoint from Unconnected state to connected state & vice versa.

For detailed operation of udpl_cm_callback refer to following figure.

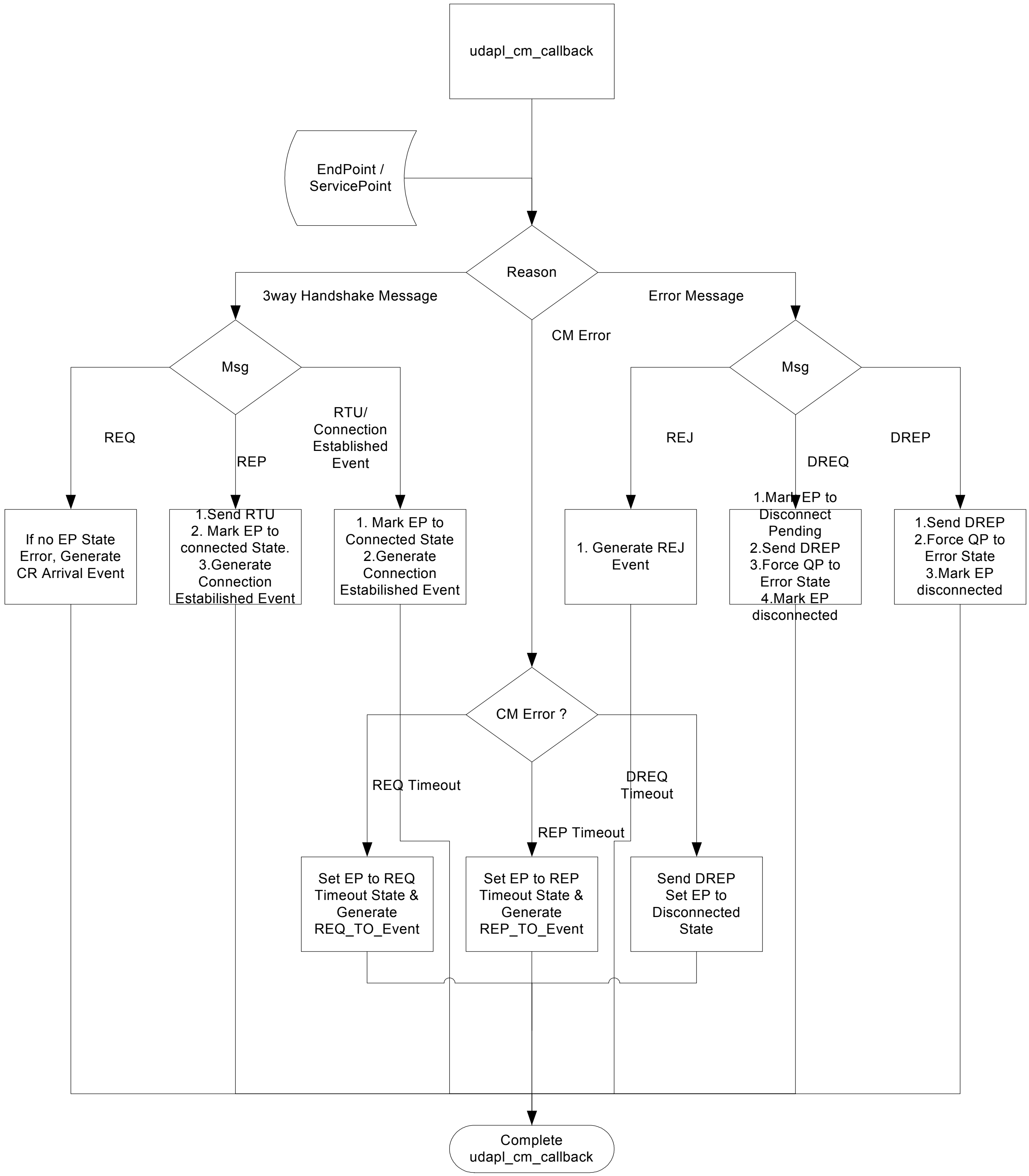

Figure 12 CM Callback handler Flow

3.3 Data Transfer & Completion Service

Data transfer operation involves converting DTO into IB work request and posting it to access layer using ib_post_send( ) & ib_post_recv( ) API's.

DTO completion can be reaped either using dat_evd_dequeue or using dat_evd_wait. An application chooses to use CNO to wait on multiple EVDs. Application may also choose to use OS proxy agent to trigger any CNO .

3.3.1 Data Transfer Service

Data transfer operation involves following operation

1. Check and make sure EP is right state.
    a. dat_ep_post_send( ) can be successful only EP is in connected state
    b. dat_ep_post_recv( ) can be successful even EP is in un connected state
2. Convert DTO into workrequest of format ib_recv_wr_t / ib_send_wr_t .
3. Acquire directional spinlock i.e TxSpinlock for post_send dto & RxSpinlock for post_recv dto.
4. Invoke ib_post_send or ib_post_recv depending on dto
5. Release any resource & spinlocks

RMR bind operation also requires to be posted using ib_post_send.

3.3.2 Completion Service

Figure below describes how various events such as connection events, DTO events and errors etc are funneled.

EVD funnels following completions/events

1. out_dto completions
2. in_dto completions
3. rmr_bind completions
4. connection request events
5. async errors

CNO funnels all EVD completions to which it is associated. However CNO reaping API dat_cno_wait only return the EVD for which completion is available but not actual event.

OS proxy funnels all CNOs to which it is associated. OS proxy agent is not DAT resource so it can funnel completions from multiple CNOs from multiple providers.

Below is the pictorial representation of how completions are funneled through.

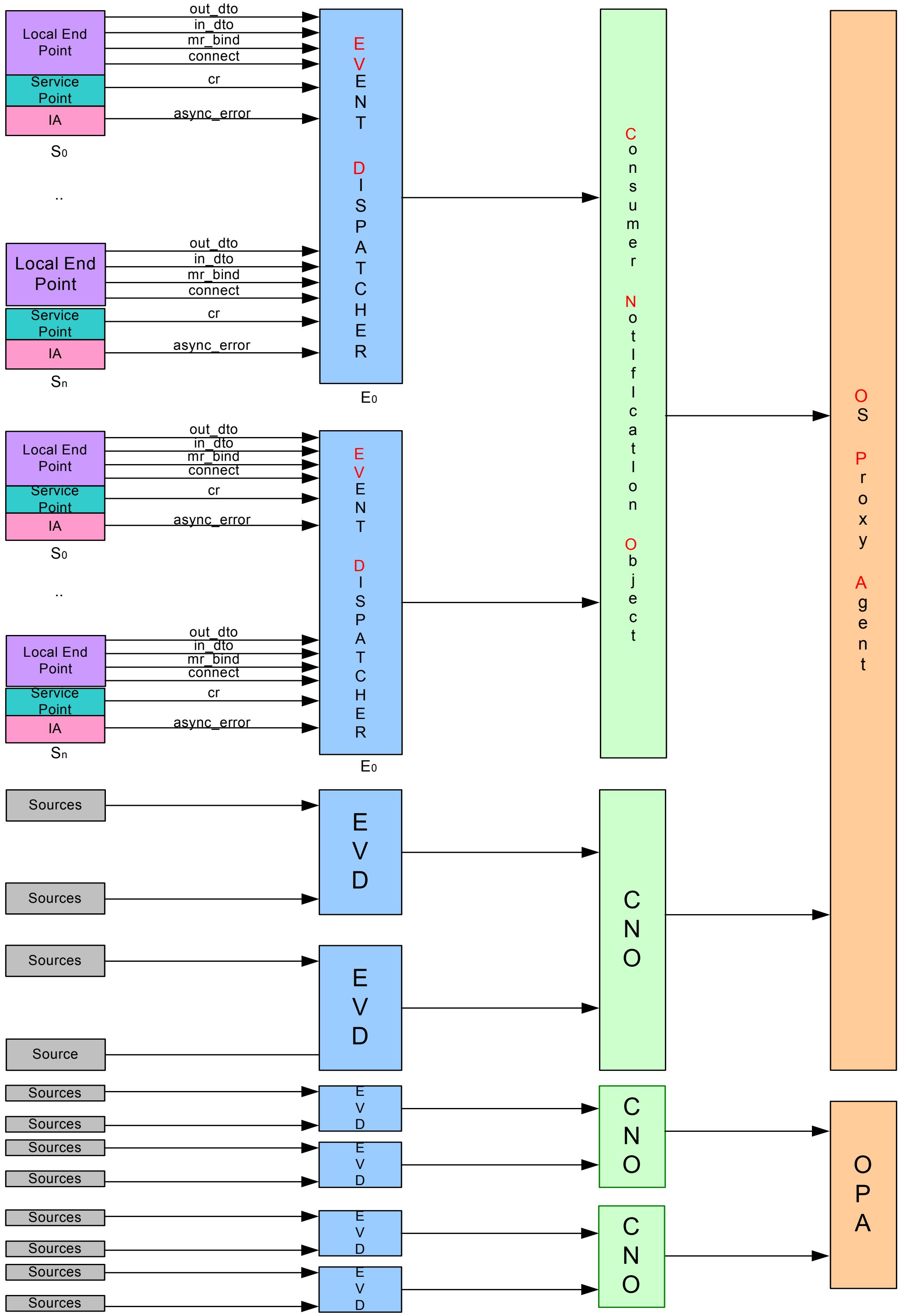

Figure 13 Completion Flow

Work request/DTO completion callback plays very vital in reaping the result in low latency manner & also provides scalability to the low latency reaping.

DTO Callback handler is registered with IB Access layer while creating Completion Queue. If Event Dispatcher is created with flag DAT_EVD_DTO_FLAG, Completion Queue is created(using ib_create_cq( ) API) and associated with EVD.

This callback is invoked by access layer if any new completion is posted to CQ and if CQ is armed. CQ can be armed using ib_rearm_cq( ). Once Handler is invoked ,

1. handler extracts the EVD information using contect
2. If EVD is not associated with CNO and if outstanding completion has reached threshold and if any thread is waiting on EVD, EVD wait object is signaled.
3. If EVD is associated with CNO and CNO is not associated with OS proxy agent and if any thread is waiting on the CNO, CNO wait object is signaled
4. If EVD is associated with CNO and CNO is associated with OS proxy agent, OS proxy agent is invoked and OSPA is marked busy.

Flow chart given below describes detailed operation of DTO callback.

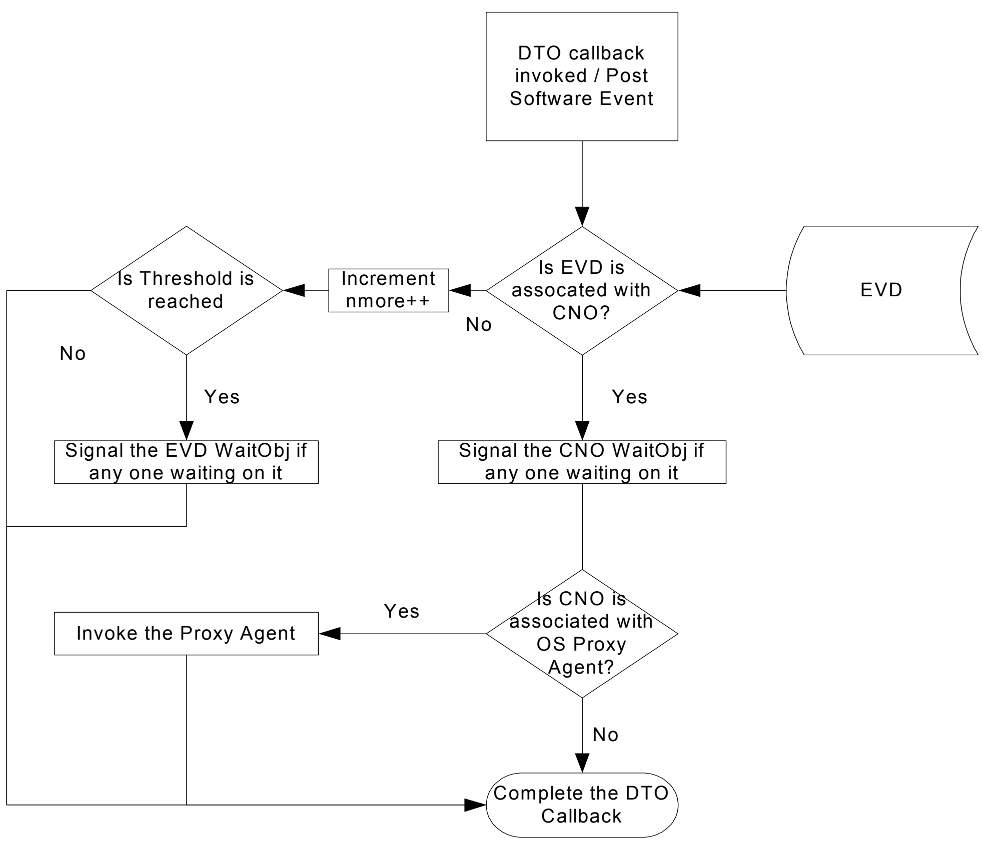

Figure 14 DTO Callback Handler Flow

This uDAPL is implimented over industry standard Access layer which confirms to the IB verbs specification. Verbs specification doesnot provide API to probe how many completions are pending without actualy dequeing them from the completion queue. So threshold in dat_evd_wait( ) may require unnecessary caching of the completion. The best recommended way of doing is define MAX_THRESHOLD as one and avoid caching the DTO completions.

Following flow chart also defines how dat_evd_wait can be implemented.

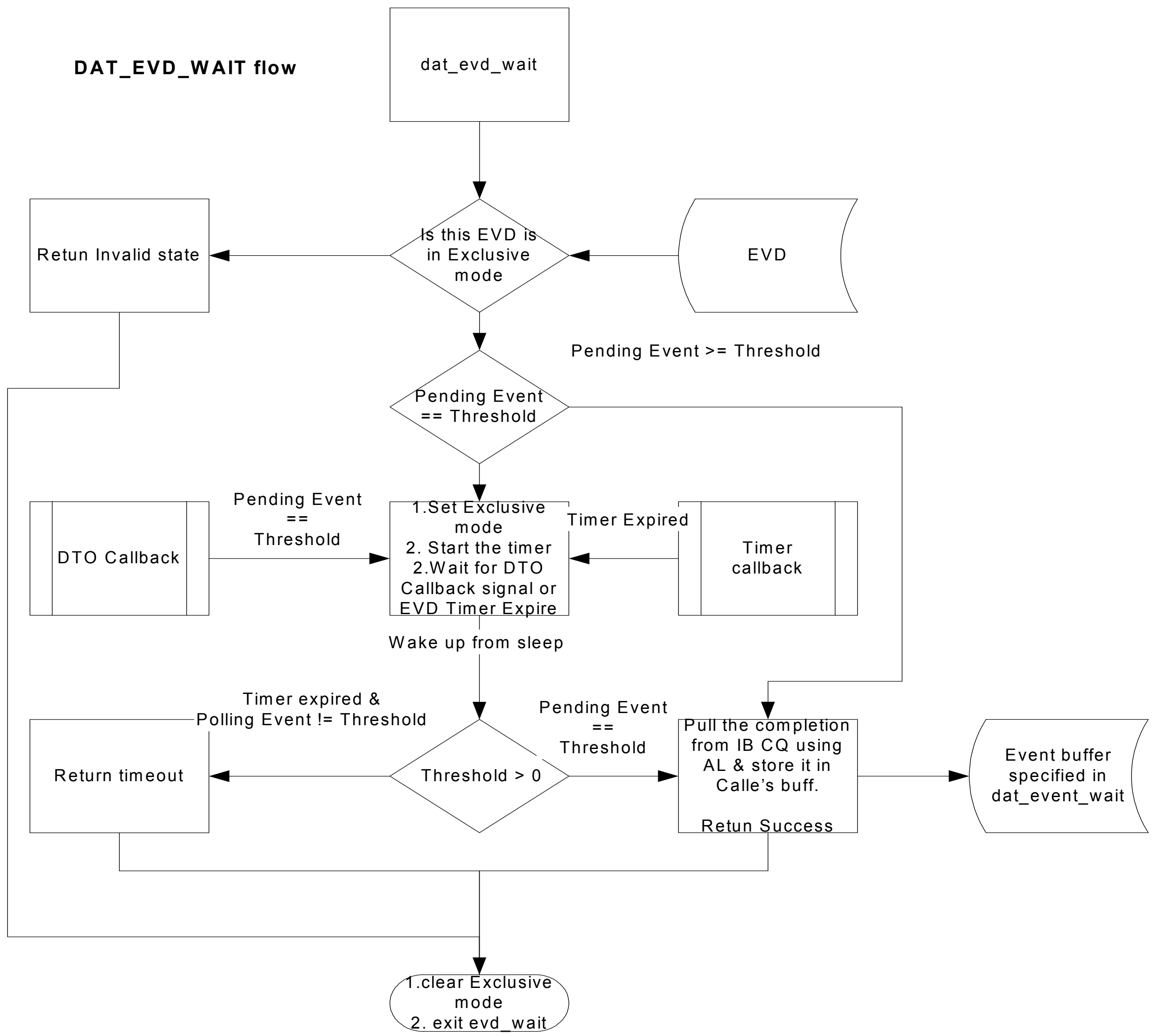

Figure 15 EVD Wait flow

3.4 API Mapping – Summary

This table summarizes the uDAPL API mapping to IB access layer APIs.

API TYPE	UDAPL	IB Access Layer Requirement
Interface Adapter	DAT_IA_Open	ib_open_ca
	DAT_IA_Close	ib_close_ca
	DAT_IA_Query	ib_query_ca
	DAT_Set_Consumer_Context	
	DAT_Get_Consumer_Context	
Event Management	DAT_EVD_Create	ib_create_cq
	DAT_EVD_Free	ib_destroy_cq
	DAT_EVD_Query	ib_query_cq
	DAT_EVD_Modify_CNO	ib_modify_cq
	DAT_EVD_Enable	ib_rearm_cq
	DAT_EVD_Disable	
	DAT_EVD_Resize	ib_modify_cq
	DAT_EVD_Wait	
	DAT_EVD_Dequeue	ib_poll_cq
	DAT_EVD_Post_SE	
Consumer Notification Object	DAT_CNO_Create	
	DAT_CNO_Free	
	DAT_CNO_Wait	
	DAT_CNO_Modify_Agent	
	DAT_CNO_Query	
Connection Management	DAT_PSP_Create	ib_cm_listen
	DAT_PSP_Free	ib_cm_cancel
	DAT_PSP_Query	
	DAT_RSP_Create	ib_cm_listen
	DAT_RSP_Free	ib_cm_cancel
	DAT_RSP_Query	
	DAT_CR_Query	
	DAT_CR_Accept	ib_cm_rep / ib_cm_rtu

	DAT_CR_Reject	ib_cm_rej
	DAT_CR_Handoff	
End Point	DAT_EP_Create	ib_create_qp
	DAT_EP_Free	ib_destroy_qp
	DAT_EP_Get_Status	ib_query_qp
	DAT_EP_Query	ib_query_qp
	DAT_EP_Modify	ib_modify_qp
	DAT_EP_Connect	ib_cm_req
	DAT_EP_Dup_Conn	
	DAT_EP_Disconnect	ib_cm_dreq/ib_cm_drep
	DAT_EP_Post_Send	ib_post_send
	DAT_EP_Post_Recv	ib_post_recv
	DAT_EP_Post_RDMA_Read	ib_post_send
	DAT_EP_Post_RDMA_Write	ib_post_send
Memory Management	DAT_PZ_Create	ib_alloc_pd
	DAT_PZ_Free	ib_dealloc_pd
	DAT_PZ_Query	
	DAT_LMR_Create	ib_reg_mem / ib_reg_shmid
	DAT_LMR_Free	
	DAT_LMR_Query	
	DAT_LMR_Modify	
	DAT_RMR_Create	ib_create_mw
	DAT_RMR_Free	ib_destroy_mw
	DAT_RMR_Query	ib_query_mw
	DAT_RMR_Bind	ib_bind_mw

3.5 Debug Services

TBD

4. Data Structures and APIs

All the uDAPL resources are doubly linked to Interface Adapter for easy maintenance. The following diagram provides a schematic of the structures used in the uDAPL library.

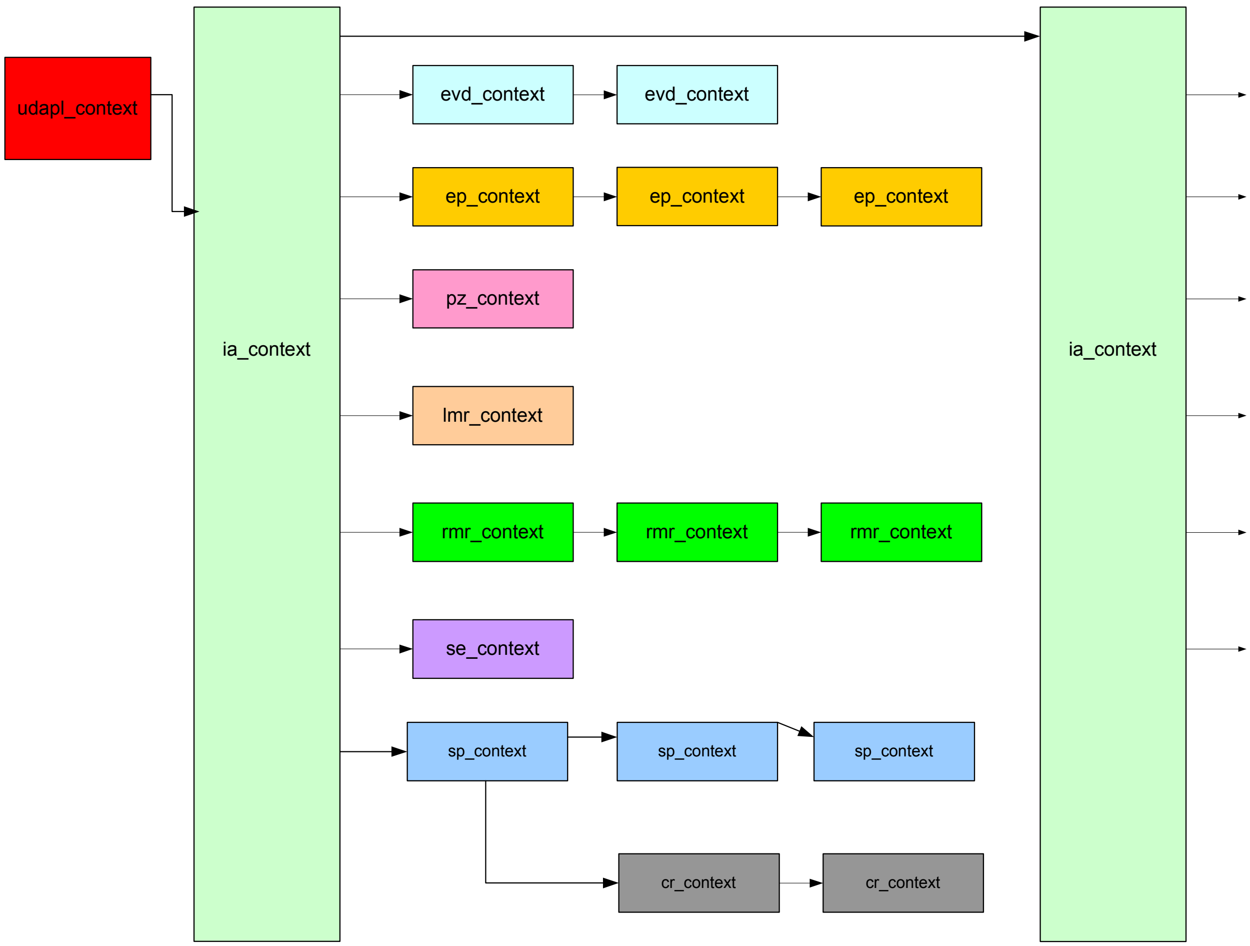

Figure 16 Structure/Context Relationship

 To improve the performance, uDAPL can allocate these resources from registered memory to avoid swapping in & out of physical memory. The actual performance gain can gauged only after experimenting with resource allocation using registered memory.

Following is the udapl_internal.h content

```
#ifndef _UDAPL_INTERNAL_
```

```
#define  _UDAPL_INTERNAL_

#include dat.h
#include ib_al.h
#include ib_types.h

typedef enum _dapl_handle_type
{
      IA_HANDLE_TYPE  ='_IA_',
      EVD_HANDLE_TYPE ='_ED_',
      EP_HANDLE_TYPE  ='_EP_',
      CR_HANDLE_TYPE  ='_CR_',
      PZ_HANDLE_TYPE  ='_PZ_',
      CNO_HANDLE_TYPE ='_CN_',
      LMR_HANDLE_TYPE ='_LR_',
      RMR_HANDLE_TYPE ='_RR_',
      PSP_HANDLE_TYPE ='_PS_',
      RSP_HANDLE_TYPE ='_RS_'
}dapl_handle_type;


typedef enum _evd_state
{
      DAT_EVD_INIT,
      DAT_EVD_ENABLED,
      DAT_EVD_EXCLUSIVE,         //cannot be polled or waited by any
other thread
      DAT_EVD_PAUSED,
      DAT_EVD_DISABLED,
      DAT_EVD_ERROR
}evd_state;


typedef enum _ep_state
{
```

```
      //For both Active & Passive
      DAT_EP_INIT,

      //Active side
      DAT_EP_LISTEN,                              //???
      DAT_EP_CONNECT_PENDING_REQ_RECVD,
      DAT_EP_CONNECT_PENDING_REP_SENT,
      DAT_EP_CONNECT_PENDING_REP_TO,        //???
      DAT_EP_CONNECT_PENDING_RTU_RECVD,

      //Passive side
      DAT_EP_CONNECT_PENDING_REQ_SENT,
      DAT_EP_CONNECT_PENDING_REQ_TO,        //REQ TimeOut
      DAT_EP_CONNECT_PENDING_REP_RECVD,
      DAT_EP_CONNECTED,

      //For both Active & Passive
      DAT_EP_DISCONNECT_DREQ_SENT,
      DAT_EP_DISCONNECT_DREQ_RECVD,
      DAT_EP_DISCONNECT_TIMEWAIT,
      DAT_EP_DISCONNECTED,

      DAT_EP_TX_ERR,
      DAT_EP_RX_ERR

}ep_state;

typedef enum _cr_state
{
      DAT_CR_ACTIVE,
      DAT_CR_TIMEOUT
}cr_state;

typedef enum _cno_state
{
```

```
    DAT_CNO_ENABLED,
    DAT_CNO_DISABLED
}cno_state;

typedef enum _proxy_agent_state
{
    PROXY_AGENT_IDLE,
    PROXY_AGENT_RUNNING
}proxy_agent_state;




typedef  struct  _udat_ia
{
    //ia maintenance variables
    cl_list_item_t        next;
    dapl_handle_type type;
    cl_spinlock_t         lock;

    //AL association
    ib_ca_handle_t        hca;
    ib_guid_t             guid;
    ib_pfn_err_cb_t       err_cb;
    void *                     hca_context;

    //uDAPL association
    DAT_NAME_PTR          ia_name;
    DAT_CONTEXT                context;
    DAT_QLEN              async_evd_qlen;
    udat_evd              async_evd_handle;

    //ia resource list
    cl_list_item_t        eplist;
    cl_list_item_t        connlist;
    cl_list_item_t        evdlist;
```

```
	cl_list_item_t		splist;

	//ia resource max
	uint32_t		max_ep;
	uint32_t		max_conn;
	uint32_t		max_evd;

}udat_ia;


typedef  struct  _udat_evd
{
	//ia maintenance variables
	cl_list_item_t			next;
	dapl_handle_type	type;
	cl_spinlock_t			lock;
	evd_state			state;
	udat_ia					ia;

	//AL/CL association
	ib_cq_handle_t			cq;
	ib_pfn_err_cb_t			cq_err_cb;
	void* const					cq_context;
	cl_event_t			wait_obj;
	ib_pfn_comp_cb_t	cq_cb;

	cl_timer_t			timer;
	cl_pfn_timer_callback_t	timer_cb;

	uint32_t			evd_wait_threshold;
	uint32_t			timer_cb_threshold;
	uint32_t			event_nmore;

	//software evd
	struct{
```

```
        void*      evd_buff;
        void*      head;
        void*      tail;
        void*      size;
    }sw_evd;

    //uDAPL association
    DAT_COUNT                  evd_len;
    udat_cno                   cno_handle;
    udat_evd                   evd_flags;
    cl_list_item_t             resource_association;
    //resources associated with this evd
    DAT_BOOLEAN                        localy_created;
    DAT_CONTEXT                        context;

}udat_evd;

typedef  struct  _udat_cno
{
    //ia maintenance variables
    cl_list_item_t             next;
    dapl_handle_type      type;
    cl_spinlock_t              lock;
    cno_state                  state;
    udat_ia                            ia;

    //AL association
    cl_wait_obj_handle_t  wait_obj;
    cl_timer_t                 timer;
    cl_pfn_timer_callback_t    timer_cb;


    //uDAPL association
    udat_ia               ia_handle;
    DAT_OS_PROXY_AGENT         agent;
```

```
    cl_list_item_t              evd_list;
    DAT_CONTEXT                      context;
    proxy_agent_state           pa_state;
}udat_cno;


typedef  struct  _udat_ep
{
    //ia maintenance variables
    cl_list_item_t              next;
    dapl_handle_type      type;
    cl_spinlock_t               lock;
    ep_state                    state;
    udat_ia                          ia;

    sockaddr_in6                local;
    sockaddr_in6                remote;

    //AL association
    ib_qp_handle_t              qp;
    ib_pfn_err_cb_t             qp_err_cb;
    void* const                      qp_context;
    cl_spinlock_t               tx_lock;
    cl_spinlock_t               rx_lock;


    //uDAPL association
    DAT_PZ_HANDLE               pz_handle;
    udat_evd              recv_evd_handle;
    udat_evd              request_evd_handle;
    udat_evd              connect_evd_handle;
    udat_evd              rmr_bind_evd_handle;
    DAT_EP_ATTRIBS              ep_attribs;
    DAT_CONTEXT                      context;
```

```
    ib_cm_req                  req;
    ib_cm_rep                  rep;
    ib_cm_rtu                  req;
    ib_cm_rej                  req;

    //
    uint32_t                   max_tx_pending;
    uint32_t                   max_rx_pending;
    uint32_t                   tx_pending;
    uint32_t                   rx_pending;

}udat_ep;




typedef  struct  _udat_sp
{
    //ia maintenance variables
    cl_list_item_t             next;
    dapl_handle_type      type;      //indicates PSP / RSP
    cl_spinlock_t              lock;
    ep_state                   state;



    //uDAPL association
    udat_ia                         ia_handle;
    udat_evd                   connect_evd_handle;        //size of
evd is size of backlog
    DAT_PSP_FLAGS              flags;
    DAT_CON_QUAL               ConnQual;

    udat_ep                         *ep;
}udat_sp;
```

```
typedef  struct  _udat_cr
{
          cl_list_item_t             next;
          dapl_handle_type      type;
          cl_spinlock_t              lock;
          cr_state                   state;
          udat_ia                         ia;

          udat_sp                         *sp;
          ib_cm_req                  req;
}udat_cr



typedef  struct  _udat_pz
{
     //ia maintenance variables
     cl_list_item_t        next;
     dapl_handle_type type;
     cl_spinlock_t         lock;
     pz_state              state;
     udat_ia                    ia;

     //AL association
     ib_pd_handle_t             pd;
     void* const                     pd_context;



     //uDAPL association
     udat_ia                    ia_handle;
     cl_list_item_t        resource_association;      //resources
associated with this pz
     DAT_CONTEXT                context;

}udat_pz;
```

```
typedef  struct  _udat_lmr
{
      //ia maintenance variables
      cl_list_item_t        next;
      dapl_handle_type type;
      cl_spinlock_t         lock;
      udat_ia                    ia;

      //AL association
      ib_mr_handle_t             ph_mr;
      uint32_t*                  p_lkey;
      uint32_t*                  r_lkey;
      ib_mr_create_t             mr;

      //uDAPL association
      DAT_MEM_TYPE                    mem_type;
      DAT_REGION_DESCRIPTION          region_description;
      DAT_VLEN                        length;
      DAT_PZ_HANDLE                   pz;
      DAT_MEM_PRIV_FLAGS              mem_privilages;
      DAT_LMR_CONTEXT                 lmr_context;
      DAT_VLEN                        registered_size;
      DAT_VADDR                       registered_address;
      DAT_CONTEXT                          context;

}udat_lmr;


typedef  struct  _udat_rmr
{
      //ia maintenance variables
      cl_list_item_t        next;
      dapl_handle_type type;
      cl_spinlock_t         lock;
```

```
    udat_ia                     ia;

    //AL association
    ib_mw_handle_t        ph_mw
    uint32_t*             p_lkey;
    uint32_t*             r_lkey;

    //uDAPL association
    DAT_PZ_HANDLE         pz;
    DAT_CONTEXT                 context;
}udat_rmr;

#endif _UDAPL_INTERNAL_
```

4.1.1 RAS Support

TBD

5. Installing, Configuring, and Uninstalling

5.1 Installing

TBD

5.2 Configuring

TBD

5.3 Uninstalling

TBD