uDAPL Extension Design and API Proposal

December 2005

Contents

1. Design Overview

The uDAPL architecture splits services up into two libraries: *libdat.[a,so]* and *libdapl.so*. The *libdat* library is linked to the uDAPL application. The primary role of this library is to dynamically load one or more specific *libdapl.so* “transport provider” libraries and implement a very thin veneer to the APIs implemented in *libdapl*.

Most *libdat* calls are actually implemented as macros that use an indirect call table provided by dynamically loaded *libdapl*. Allowing the requested transport provider to be loaded dynamically allows an uDAPL based application to use more than one type of interconnect technology simultaneously.

To keep the impact of adding extensions to *libdat* to a minimum while allowing the largest degree of freedom for extensions APIs, one additional entry point is added to the end of the provider call table that takes an extension ID followed by a va_arg list. This retains the ability to implement extension calls in *libdat* as macros with the small additional overhead of parsing the va_arg list within the provider specific *libdapl*.

To allow for complete compatibility between versions of *libdat* and *libdapl* that do and do not support the extension interface, *libdat* is modified to probe a loaded *libdapl* to see if it is supports the extension call table entry point. A Boolean is set according to the response and tested within the dat_api extension call to prevent making an invalid reference in the call table. When the provider does not support the extension interface, DAT_NOT_IMPLEMENTED is returned.

Even though a provider may support the extension interface, it is not required to support all extension services. An application can query the provider for a list of extensions services using the *dat_ia_query()* interface and iterating through the provider specific key-value pairs for supported services. A provider will return DAT_NOT_IMPLEMENTED if an unsupported extension service is requested.

The other impact to *libdat* and associated include files is the addition of a new DAT_EVENT number, DAT_EXTENSION_EVENT and a new event data type, DAT_EXTENSION_DATA. The new data type has the following structure:

```
typedef struct _dat_extension_data {
      DAT_DTO_COMPLETION_EVENT_DATA       dto;
      DAT_EXT_EVENT_TYPE                  type;
      union {
            … /* event specific data */
            … /* event specific data */
      };
} DAT_EXTENSION_DATA;
```

The union section is where extension specific data is defined. The DAT_EXTENSION_DATA type, along with extension call interface macros, DAT_EXT_TYPE defines, and extension specific data is defined in ~/include/dat/dat_extensions.h. This file may be supplied by the uDAPL provider to define the extensions supported. However, it is expected that all providers will follow the same interface for any specific extension specified in this document.

This document is organized as follows: Section 1 provides an overview of the uDAPL architecture and how the extension interface will be added in a backward compatible manner. Section 2 specifies the changes made to existing uDAPL data structures and types as well as new data structures and types required to implement extension APIs. Finally, Section 3 specifies a series of specific extension APIs that cover general, atomic, and collective operations.

2. Extension Data Structures and Types

All prototypes, macros, types, and defines for these extension APIs are defined in ~/include/dat_extensions.h. Adding extension capabilities to uDAPL requires a few minor modifications to existing include files and data structures with the bulk of new extension types and definitions contained in the file dat_extensions.h.

2.1 Modified uDAPL Structures and Types

2.1.1 struct dat_provider

The dat_provider structure (udat_redirection.h) is native to uDAPL and defines the provider redirection call table. To accommodate the core extension call, a single entry is added to the end as follows:

```
struct _dat_provider {
      . . .
      . . .

      /* extensions */
      DAT_EXTENSION_FUNC        extension_func;
};
```

2.1.2 DAT_EVENT_DATA

The DAT_EVENT_DATA type (dat.h) is native to uDAPL as a union of all returned event data types. To accommodate extension data types, it is augmented with an additional entry to cover extension data types.

```
typedef union dat_event_data {
      . . .
      . . .
      DAT_EXTENSION_DATA        extension_data;
} DAT_EVENT_DATA;
```

2.1.3 DAT_EVENT_NUMBER

To define the format of event data, at DAT event contains an event number type. To specify the extension data types, an additional entry (dat.h) is added to the DAT_EVENT_NUMBER enum.

```
typedef enum dat_event_number {
      . . .
      . . .
      DAT_EXTENSION_EVENT       = 0x20001;
} DAT_EVENT_NUMBER;
```

2.2 Provider Specific Query Types

Provider specific attribute strings for extension support are returned with dat_ia_query() using DAT_PROVIDER_ATTR_MASK set to DAT_PROVIDER_FIELD_PROVIDER_SPECIFIC_ATTR where DAT_NAMED_ATTR name is set to “extended operation” and value is set to “TRUE” when extended operation is supported. The following definitions are used for these new extensions.

```
#define DAT_EXT_ATTR                        "DAT_EXTENSION_INTERFACE"
#define DAT_EXT_ATTR_RDMA_WRITE_IMMED       "DAT_EXT_RDMA_WRITE_IMMED"
#define DAT_EXT_ATTR_RECV_IMMED             "DAT_EXT_RECV_IMMED"
#define DAT_EXT_ATTR_RECV_IMMED_EVENT       "DAT_EXT_RECV_IMMED_EVENT"
#define DAT_EXT_ATTR_RECV_IMMED_PAYLOAD     "DAT_EXT_RECV_IMMED_PAYLOAD"
#define DAT_EXT_ATTR_FETCH_AND_ADD          "DAT_EXT_FETCH_AND_ADD"
#define DAT_EXT_ATTR_CMP_AND_SWAP           "DAT_EXT_CMP_AND_SWAP"
#define DAT_EXT_ATTR_TRUE                   "TRUE"
#define DAT_EXT_ATTR_FALSE                  "FALSE"
```

2.3 Sample Extension Structures and Types

2.3.1 DAT_EXTENSION_FUNC

The prototype for the extension call is defined (dat_redirection.h) as follows:

```
typedef DAT_RETURN (*DAT_EXTENSION_FUNC) (
      IN    DAT_HANDLE,       /* DAT handle                 */
      IN    DAT_EXT_OP,       /* DAT extension operation    */
      IN    va_list);         /* va_list, variable arguments*/
```

2.3.2 DAT_EXT_OP

The DATA_EXT_OP enum specifies the type of extension operation.

```
typedef enum dat_ext_op {
      DAT_EXT_RDMA_WRITE_IMMED,
      DAT_EXT_RECV_IMMED,
      DAT_EXT_FETCH_AND_ADD,
      DAT_EXT_CMP_AND_SWAP,
} DAT_EXT_OP;
```

The enumeration will be extended as new extension operations are added.

2.3.3 DAT_EXT_EVENT_TYPE

The DATA_EXT_EVETN_TYPE enum specifies the type of extension data contained in the DAT_EVENT.

```
typedef enum dat_ext_event_type {
      DAT_EXT_RDMA_WRITE_IMMED_STATUS,
      DAT_EXT_RECV_NO_IMMED,
      DAT_EXT_RECV_IMMED_DATA_EVENT,
      DAT_EXT_RECV_IMMED_DATA_PAYLOAD,
      DAT_EXT_FETCH_AND_ADD_STATUS,
      DAT_EXT_CMP_AND_SWAP_STATUS,
} DAT_EXT_EVENT_TYPE;
```

The enumeration will be extended as new extension data types are added.

2.3.4 DAT_EXT_FLAGS

The DATA_EXT_FLAGS enum specifies the extended flags for operations.

```
typedef enum dat_ext_flags {
      DAT_EXT_WRITE_IMMED_FLAG      = 0x1,
      DAT_EXT_WRITE_CONFIRM_FLAG    = 0x2,
} DAT_EXT_FLAGS;
```

2.3.5 DAT_IMMEDIATE_DATA

The DAT_IMMEDIATE_DATA type contains the 32 bites of immediate data associated with an RDMA write.

```
typedef struct dat_immediate_data {
      DAT_UINT32  data;
} DAT_IMMEDATE_DATA;
```

2.3.6 DAT_EXTENSION_DATA

When a DAT_EVENT specifies an event type of DAT_EXTENSION_EVENT, event data is defined as follows:

```
typedef struct dat_extension_data {
        DAT_DTO_COMPLETION_EVENT_DATA    dto;
        DAT_EXT_EVENT_TYPE               type;
        union {
                DAT_RDMA_WRITE_IMMED_DATA         immed;
        };
} DAT_EXTENSION_DATA;
```

The union will be extended as new extension APIs are added.

2.4 Sample Extension APIs

The following function prototypes are actually implemented as pre-processor macros. The macro validates that extensions are supported and then calls the DAT_EXTENSION_FUNC vector in the *dat_provider* structure. The type definition for the core extension call is as follows:

```
typedef DAT_RETURN (*DAT_EXTENSION_FUNC) (
      IN    DAT_HANDLE,       /* DAT handle                 */
      IN    DAT_EXT_OP,       /* DAT extension operation    */
      IN    va_list);         /* va_list, variable arguments*/
```

Each API below details input/output arguments and completion semantics. Explicit return codes are not given but they can be assumed to be logical uses of existing DAT return codes.

A uDAPL application can determine which extensions are supported by a uDAPL provider by making the *ep_ia_query()* call and iterating the DAT_NAMED_ATTR array pointed to by the *provider_specific_attr* member in DAT_PROVIDER_ATTR. The DAT_NAMED_ATTR type contains two string pointers of *name* and *value*. The table below specifies the *name*/extension relationship. In most cases, simply having the name defined implies support and the *string* value does not supply additional context.

Extension	Name Attribute
Indicates general support for extensions	DAT_EXTENSION_INTERFFACE
Indicates immediate data delivered in event	DAT_EXT_RECV_IMMED_EVENT
Indicates immediate data delivered in payload	DAT_EXT_RECV_IMMED_PAYLOAD
dat_ep_post_write_immed	DAT_EXT_RDMA_WRITE_IMMED
dat_ep_post_recv_immed	DAT_EXT_RECV_IMMED
dat_ep_post_fetch_and_add	DAT_EXT_FETCH_AND_ADD
dat_ep_post_cmp_and_swap	DAT_EXT_CMP_AND_SWAP

2.4.1 Immediate Data Extensions

2.4.1.1 dat_ep_post_write_immed()

```
DAT_RETURN
dat_ep_post_write_immed(
      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_TRIPLE       *remote_iov,
      IN DAT_UINT32           immediate_data,
      IN DAT_COMPLETION_FLAGS completion_flags);
```

This asynchronous call performs a normal RDMA write to the remote endpoint followed by a post of an extended immediate data value to the receive EVD on the remote endpoint. The immediate data will consume a posted receive immediate buffer at the remote endpoint.

Extended Flags:

DAT_EXT_WRITE_IMMED_FLAG requests that the supplied 'immediate' value be sent as the payload of a four byte send following the RDMA Write, or any transport-dependent equivalent thereof.

DAT_EXT_WRITE_CONFIRM_FLAG requests that this DTO not complete until receipt by the far end is confirmed. Event completions are as follow:

Endpoint	**EVD**	**Extension Type**	**Extension Event Data Type**
Initiator	Request DAT_EXTENSION_EVENT	`DAT_EXT_RDMA_WRITE_IMMED_STATUS`	N/A
Remote	Receive DAT_EXTENSION_EVENT	`DAT_EXT_RECV_IMMED_DATA_EVENT`	`DAT_RDMA_WRITE_IMMED_DATA`

2.4.1.2 dat_ep_post_recv_immed()

```
DAT_RETURN
dat_ep_post_write_immed(
      IN DAT_EP_HANDLE        ep_handle,
      IN DAT_COUNT            size,
      IN DAT_LMR_TRIPLET      *local_iov,
      IN DAT_DTO_COOKIE       user_cookie,
      IN DAT_COMPLETION_FLAGS completion_flags);
```

This call performs a normal post receive message to the local endpoint that includes additional 32-bit buffer space to receive immediate data. Event completion for the request completes as follow:

Endpoint	**EVD**	**Extension Type**	**Extension Event Data Type**

Initiator	Receive
	DAT_EXTENSION_EVENT
		`DAT_EXT_RECV_IMMED_DATA_EVENT or`
		`DAT_EXT_RECV_IMMED_DATA_PAYLOAD`
			`DAT_RDMA_WRITE_IMMED_DATA`

2.4.2 Atomic Extensions

2.4.2.1 dat_ep_post_cmp_and_swap()

```
DAT_RETURN
dat_ep_post_cmp_and_swap(
      IN DAT_EP_HANDLE        ep_handle,
      IN DAT_UINT64           cmp_value,
      IN DAT_UINT64           swap_value,
      IN DAT_LMR_TRIPLE       *local_iov,
      IN DAT_DTO_COOKIE       user_cookie,
      IN DAT_RMR_TRIPLE       *remote_iov,
      IN DAT_COMPLETION_FLAGS completion_flags);
```

This asynchronous call is modeled after the InfiniBand atomic Compare and Swap operation. The *cmp_value* is compared to the 64 bit value stored at the remote memory location specified in *remote_iov*. If the two values are equal, the 64 bit *swap_value* is stored in the remote memory location. In all cases, the original 64-bit value stored in the remote memory location is copied to the *local_iov*.

Endpoint	**EVD**	**Extension Type**	**Extension Event Data Type**
Initiator	Request DAT_EXTENSION_EVENT	DAT_EXT_CMP_SWAP_STATUS	N/A
Remote	N/A	N/A	N/A

DAT_EXT_FETCH_AND_ADD_STATUS

2.4.2.2 dat_ep_post_fetch_and_add()

```
DAT_RETURN
dat_ep_post_fetch_and_add(
      IN DAT_EP_HANDLE        ep_handle,
      IN DAT_UINT64           add_value,
      IN DAT_LMR_TRIPLE       *local_iov,
      IN DAT_DTO_COOKIE       user_cookie,
      IN DAT_RMR_TRIPLE       *remote_iov,
      IN DAT_COMPLETION_FLAGS completion_flags);
```

This asynchronous call is modeled after the InfiniBand atomic Fetch and Add operation. The *add_value* is added to the 64 bit value stored at the remote memory location specified in *remote_iov*. The original pre-added 64 bit value stored in the remote memory location is copied to the *local_iov*.

Endpoint	**EVD**	**Extension Type**	**Extension Event Data Type**
Initiator	Request DAT_EXTENSION_EVENT	DAT_EXT_FETCH_ADD_STATUS	N/A
Remote	N/A	N/A	N/A