RPC Bulk Transfer Task Plan

This document captures miscellaneous thoughts on how to implement the RPC Bulk transfer task starting in Colibri Sprint 10. The document is not a DLD (that will be the documentation in the code) nor is it an HLD (none exists). But it is pseudo-HLD like...

Revision history

Version	Change	Author	Date
1	Initial draft with client read pseudo-code.	Carl Braganza	3/4/2011
2	Changed memory descriptor to uint64 array. Added blurb on batching support	Carl Braganza	3/7/2011
3	Asynchronous bulk fetch in server/send in client. Transport event queue mechanism. c2t1fs server example. Re-sectioned.	Carl Braganza	3/8/2011
4	Feedback from Nikita	Nikita Danilov	3/9/2011
5	Applied changes suggested by Nikita. Renamed “memory descriptor” to “network buffer descriptor”. Completed client call and network buffer event delivery. Used network buffer in server handler wait.	Carl Braganza	3/9/2011
6	Simplified asynchronous server execution. Dissociated the processing of request and response FOPs with the execution context. Defined desired semantics of the service handler and reply post methods. Dropped special “handler wait” support proposed earlier. Updated c2t1fs server example to illustrate.	Carl Braganza	3/9/2011
7	Sketched the mechanism for asynchronous server/client behavior and implementation of the c2_net_nb_ interfaces. Cleaned up details as the DLD got written.	Carl Braganza	3/11/2011
8	Use separate staged and transfer buffer queues in the network domain. Provided query for max buffer size. Described sunrpc transport implementation, including the FOPs used for buffer transfer. Updated implementation steps.	Carl Braganza	3/14/2011
9	Take2. Applied feedback from Nikita to transform this into an event driven, asynchronous message passing layer with support for asynchronous bulk data transfer. Added logical and functional specification and a typical handshake use case.	Carl Braganza	3/17/2011
10	Applying feedback from Nikita: support for originators without service ids, end point information in incoming messages,	Carl Braganza	3/21/2011

	separate static structure with callbacks. Logical specification complete.		
11	Working on the SUNRPC based implementation.	Carl Braganza	3/22/2011
12	Updates while writing the DLD. More meaningful queue type names (consistently use active/passive for bulk transfer). Separate success/failure event count stats. Consistent apis with existing patterns.	Carl Braganza	3/23/2011
13	Applied feedback from Dave. Added state to the transfer machine and defined state events. Added transfer machine param queries.	Carl Braganza	3/24/2011
14	Applied feedback from Nikita's review of the DLD. Registered buffers tracked in domain, usable in all transfer machines. Buffer state flags. Endpoints tracked by domain too. No ioctl like subs to get/set parameters - use specific subs.	Carl Braganza	3/25/2011
15	More feedback from Nikita. Allow per buffer callbacks. Define the concurrency model.	Carl Braganza	3/28/2011
16	Design sunrpc based messaging over existing sunrpc net layer - functional and logical specifications added. More feedback from Nikita: ntm_q[QT].	Carl Braganza, Dave Cohrs	4/8/2011
17	Change locking semantics to accommodate processor affinity support in the transport.	Carl Braganza, Dave Cohrs	4/11/2011
18	Flushed out in-memory and specified sunrpc as an extension to in-memory.	Carl Braganza	4/12/2011
19	Updated sunrpc and in-memory figure. Defined in-memory communication as taking place across domains.	Carl Braganza	4/14/2011
20	Dave and I realized that we need a FAILED transfer m/c state or else we wouldn't be able to stop a TM that couldn't start. The in-mem and sunrpc work functions need the TM pointer. Extended semantics around C2_NET_BUF_IN_USE.	Carl Braganza	4/15/2011
21	Defined buffer descriptor for in-memory transport, and add a buffer id to the buffer private data.	Carl Braganza	4/19/2011
22	Added nep_addr field to end point. Described changes to the s_handler cookie to provide client connection information.	Carl Braganza	4/26/2011
23	Described serialization during deferred sunrpc network connection creation. Added sunrpc buffer private data structure.	Carl Braganza Dave Cohrs	4/28/2011
24	Added end point information in the RPC messages and active and passive end point information in the descriptor used for sunrpc.	Carl Braganza	4/29/2011
25	Remove c2_net_conn pointer from the bulk sunrpc end point.	Carl Braganza	5/4/2011
26	Accommodate the constraint of only one usunrpc server per	Carl Braganza	5/5/2011

	process by using a single server and 3-tuple addressing.		
27	Added a nev_next_state field to the c2_net_event structure to allow for failure during state change.	Carl Braganza	5/6/2011
28	Specify how an in-mem TM stop serializes with worker thread termination.	Dave Cohrs	5/9/2011
29	sunrpc xo_tm_start has to ensure that the EP is unique in the process.	Carl Braganza	5/9/2011
30	Defined C2_NET_XOP_ERROR_CB for error callbacks in the bulk emulation transports. Added a section on event delivery.	Carl Braganza	5/10/2011
31	Added Kernel Porting Plan section with initial planning notes.	Dave Cohrs	5/11/2011
32	Making c2_net_buffer_del() a void, based on feedback from Nikita. Semantics of the underlying xo_ subs also change.	Carl Braganza	5/13/2011
33	Introduce an event type to help distinguish the event context, based on feedback from Nikita. Replaced the nev_qtype field in the event with the nev_type enumeration.	Carl Braganza	5/16/2011
34	Based on feedback from Nikita: made c2_net_tm_event_post a void, and dropped the tm parameter.	Carl Braganza	5/18/2011
35	Based upon feedback from Nikita, end point address will be a string. Added additional clarification of the serialization between end point create and release, based on feedback from Rajesh.	Carl Braganza	5/19/2011
36	Based on feedback from Rajesh: dropped the tm parameter from c2_net_tm_cb_proc_t	Dave Cohrs	5/19/2011
37	Based upon feedback from Nikita, using a separate event structure for buffer completion events. Also, eliminating the guarantee of serialized event delivery per buffer (remove the C2_NET_BUF_IN_CALLBACK flag).	Carl Braganza Dave Cohrs	5/26/2011
38	Support caching of sunrpc end points to reduce sockets in TIMED_WAIT, using a per-domain skulker thread.	Carl Braganza	6/7/2011
39	Described support for buffer timeout operations in bulk sunrpc emulation, also off the skulker thread. Also described semantic changes in messaging layer to support this.	Carl Braganza	6/30/2011
40	After team discussions in Pune, moving end point management from the domain to the transfer machine.	Carl Braganza	7/18/2011
41	c2_net_tm_fini must hold transfer machine lock while putting the ntm_ep.	Dave Cohrs	7/22/2011

Contents

Component worksheet definition of task

interaction with a transport level rdma. This includes bringing the rdma branch up to date and looking through LNET bulk interfaces in Lustre to understand how rdma interfaces of the future network transport layer will look like. A stub implementation of these interfaces should be provided and rpc-level bulk implemented on top of it. The task outcome is to demonstrate:

- explicit memory descriptor management for the higher level so that a client can get MDs for the data buffers to send, and a server can explicitly identify in the RPC the MDs to fetch. Memory alignment negotiation;
- asynchronous BULK send capability for RPCs (client send);
- asynchronous BULK fetch capability for MDs (server get);
- ability to block waiting for events (client and server).

Rpc layer should not assume that the underlying transport uses RDMA for all transfers."

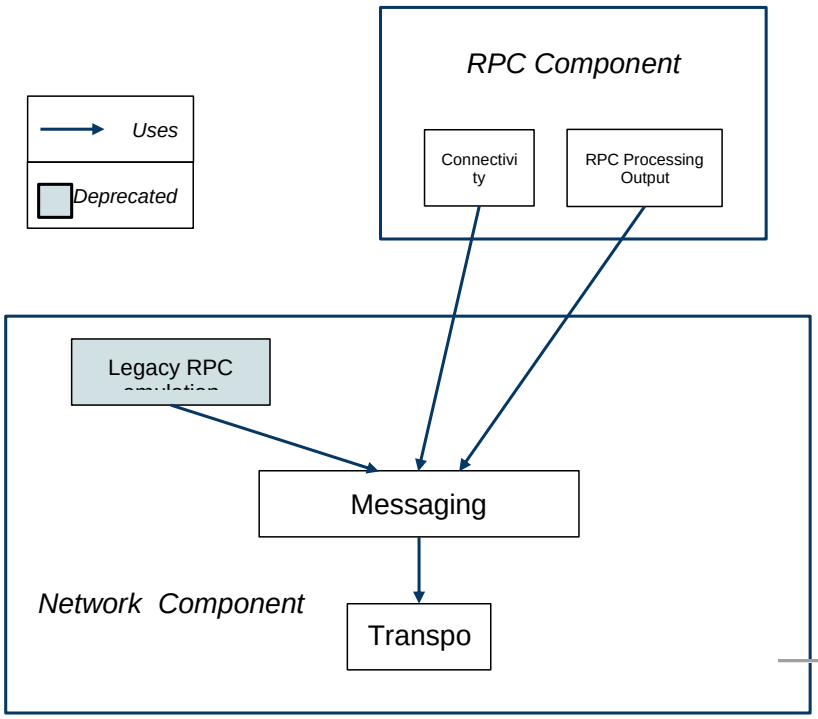
Take 2

My previous attempt at this task preserved RPC semantics at this layer, supporting existing interfaces. Nikita clarified that he wants this layer to provide event-based, asynchronous

message passing of opaque data over a bulk data transport. RPC semantics would be provided by an upper layer - indeed, by this time the HLD of RPC core was released, which further clarified the purpose of the bulk transfer network layer and how it relates to the RPC layer. The upper level will explicitly manage buffers that are to be provided for use by the bulk transport - this avoids the expense of re-registration of buffers with the kernel and NICs (anticipating future LNET). Asynchronous event notification is key to supporting non-blocking behavior in the upper level.

Introduction

The following figure illustrates the “uses” relationship between specific sub-components of the RPC and network components. It shows the RPC sub-components utilizing a messaging sub-component associated with a network domain to perform bulk transfer operations. The arrow is uni-directional but there will be callbacks made to the invoking layer.



The figure also illustrates that the existing legacy but now deprecated “RPC like” semantics provided through the **c2_net_call** and **c2_service** data structures, can be layered above the

Commented [1]: "Legacy" interfaces should be removed as soon as possible, while we have only few users of them. Please don't consider them a constraint. —nikita_danilov

Good. However, I take it that this means we need to replace their existing usage with the new interfaces.
- Carl

Later feedback from Nikita - don’t bother with legacy interfaces
- Carl

messaging component to preserve existing interfaces that are in use by the **c2t1fs** subsystem. Also shown in the figure is the underlying transport sub-component (represented in the existing code by **c2_net_xprt**) like the future LNET or the existing but to-be-deprecated Sunrpc transport.

Accordingly, this task is now redefined to include:

- Creation of the messaging component.
- Implement support using the Sunrpc transport to demonstrate the proof-of-concept.
- ~~Implement support for the legacy **c2_net_call** and **c2_service** based interfaces.~~

Requirements

Support for the RPC module requirements as described in the HLD of the RPC core:

[r.rpccore.flexible] supports synchronous and asynchronous communication;
[r.rpccore.flexible] supports sending various auxiliary information;
[r.rpccore.efficient] the network bandwidth is utilized fully by sending concurrent messages;
[r.rpccore.efficient.bulk] 0-copy, if provided by the underlying network transport, is utilized;

Additional requirements:
[r.messaging.oneway] support for one-way messaging
[r.messaging.efficient.buffer-use] support for efficient management of buffers
[r.messaging.processor_affinity] Provide support for optimal processor affinity allocation.

Design highlights

The design exhibits the following:

- Fully asynchronous message passing (no RPC semantics).
- One-way messaging possible.
- Application controlled buffer management with support for pre-registered memory buffers for efficiency .
- Support for asynchronous bulk data transfer.
- Individual callbacks for specific event types (reduces application processing)
- Collection of run-time statistical information that could be of use to upper layers.

In addition, it is possible to provide legacy support interfaces for the previous RPC like semantics provided by this module.

Functional specification

The network module is composed of a Messaging component and one or more transport components. The Transport component provides the transport technology specific methods for the abstract messaging objects.

Commented [2]: Nikita wants to maintain a pool of registered buffers for efficiency. I'm trying to find out from Isaac when LNET pins/unpins buffers and if this concept is possible with LNET. I'd drop it if there is no benefit. —carl_braganza

Currently, LNet registers a buffer as part of GET or PUT operation. But this has been a plan for a long time (starting with Lustre) to export registration interfaces separately. Even if this is not at the moment possible, separate registration interface (with a no-op implementation) should still be provided.

Good to know that.
- Carl

Heard from Isaac - LNET pinning happens in the LND. —carl_braganza

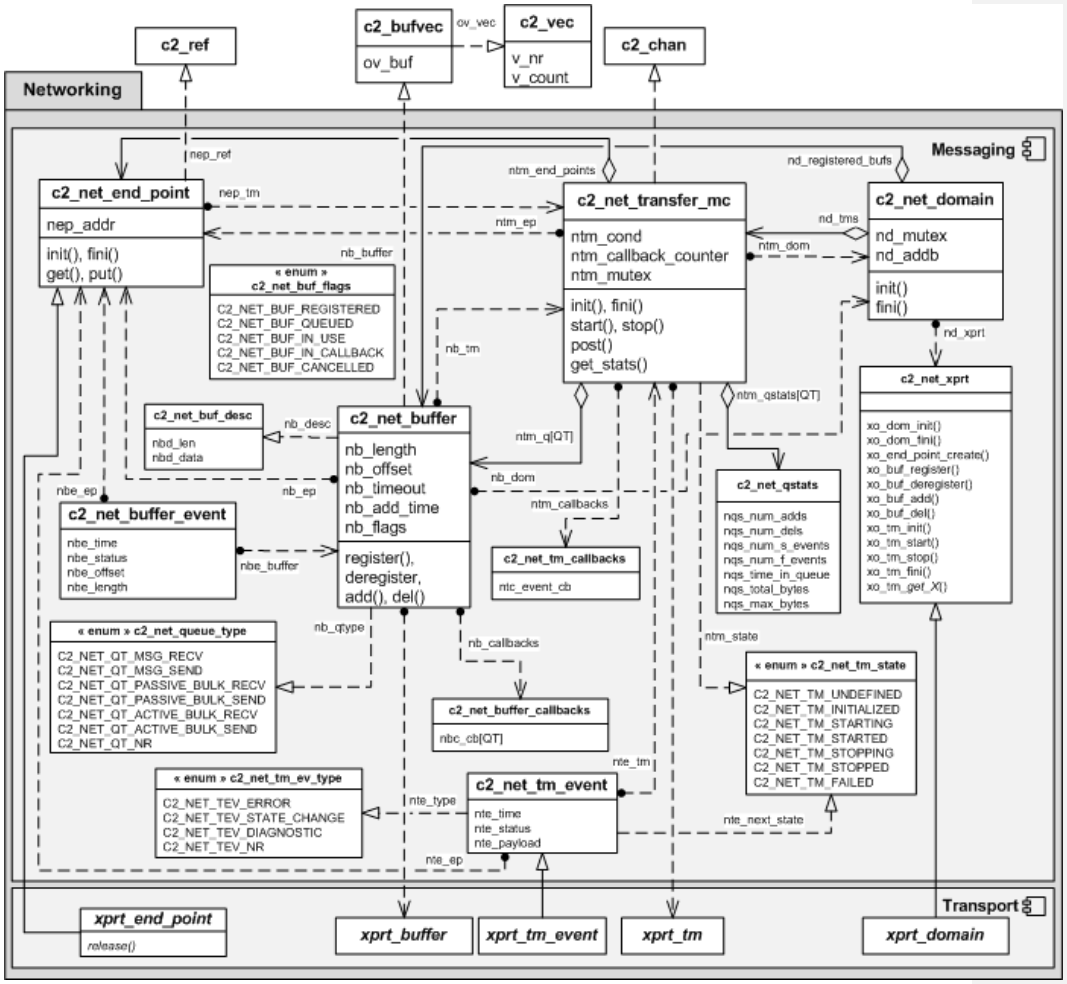
Commented [3]: A request from Nikita. — carl_braganza

Commented [4]: Statistics should be reported via addb. —nikita_danilov

I'm talking about data needed by the RPC formation component - I presume timing information, queue sizes etc. would be of interest.
- carl

The UML specification for the Network module is shown below.

The actual code is in C, so the UML generalization depicted in the figure is realized by pointers to private data and separate method structures set upon initialization. UML realization is implemented by an instance of the referenced data type in the structure. UML dependencies are implemented by pointers. Lists are created using the **c2_list** support, which requires a **c2_list_link** in the element object. The UML specification below does not capture all the possible methods, but focuses more on the properties and relationships.



The major data structures are:

c2_net_xprt This represents a technology specific transport layer. The object indirectly supplies the "**xo**_" methods shown above.

c2_net_domain This represents the network domain - a collection of network resources associated with a specific network transport.

c2_net_transfer_mc
The operational abstraction. This data structure tracks end points, message buffers, and supports callbacks to notify the application of changes in state associated with these buffers. There can be multiple transfer machines within a network domain. A transfer machine must be started before use.

c2_net_end_point This represents an addressable entity involved in message passing. The actual form of an end point is transport specific, but is expressed as a C string. Memory for the end point is handled by the network layer itself, from within the internal transport component, unlike most of the other data structures; this is because reference counting is performed for the structure, and it gets released when the reference count goes to 0. An end point object is only usable with the transfer machine used to create it. The **nep_addr** field is a printable representation of the end point address and can be used to re-create the same end point in another transfer machine.

c2_net_buffer This represents a memory region (described by a **c2_bufvec**) involved in message passing. The transport will likely maintain technology specific data for each buffer.

c2_net_buf_desc
This is an opaque transport specific descriptor for a **c2_net_buffer**. It can be sent to another end point to perform 0 copy bulk data transfer if supported by the transport. XDR routines are available to serialize this data structure.

c2_net_buffer_event
This describes a buffer operation completion event. Separate callback methods are required, one for each type of buffer queue.

c2_net_tm_event
This describes a general transfer machine event. There are multiple event types possible.

The Network module will present the following interfaces:

- `int c2_net_init();`
- `int c2_net_fini();`
- `int c2_net_xprt_init(struct c2_net_xprt *xprt);`
- `int c2_net_xprt_fini(struct c2_net_xprt *xprt);`
- `int c2_net_domain_init(struct c2_net_domain *dom, struct c2_net_xprt *xprt);`
- `int c2_net_domain_fini(struct c2_net_domain *dom);`
- `int c2_net_domain_get_max_buffer_size(struct c2_net_domain *dom, c2_bcount_t *size);`
- `int c2_net_domain_get_max_buffer_segments(struct c2_net_domain *dom, uint32_t *num_segs);`
- `int c2_net_tm_init(struct c2_net_transfer_mc *tm, struct c2_net_domain *dom);`
- `int c2_net_tm_start(struct c2_net_transfer_mc *tm, const char *addr);`
- `int c2_net_tm_stop(struct c2_net_transfer_mc *tm, bool abort);`
- `int c2_net_tm_fini(struct c2_net_transfer_mc *tm);`
- `void c2_net_tm_event_post(const struct c2_net_tm_event *ev);`
- `int c2_net_tm_stats_get(struct c2_net_transfer_mc *tm, enum c2_net_qtype qt, struct c2_net_qstats *qs, bool reset);`

Commented [5]: This replaces the previous c2_net_conn and c2_service_id. —carl_braganza

Commented [6]: Existing interfaces. —carl_braganza

Commented [7]: Modified 7/18/2011 to take an end point address and not a pointer. —carl_braganza

Commented [8]: All application end point references must be released before fini. —carl_braganza

- *int c2_net_end_point_create(struct c2_end_point **ep, struct c2_net_transfer_mc *tm, const char *addr);*
- *int c2_net_end_point_get(struct c2_end_point *ep);*
- *int c2_net_end_point_put(struct c2_end_point *ep);*
- *int c2_net_buffer_register(struct c2_net_buffer *buf, struct c2_net_domain *dom);*
- *int c2_net_buffer_deregister(struct c2_net_buffer *buf, struct c2_net_domain *dom);*
- *int c2_net_buffer_add(struct c2_net_buffer *buf, struct c2_net_transfer_mc *tm);*
- *void c2_net_buffer_del(truct c2_net_buffer *buf, struct c2_net_transfer_mc *tm);*
- *void c2_net_buffer_event_post(const struct c2_net_buffer_event *ev);*
- *int c2_net_desc_copy(struct c2_net_buf_desc *from, struct c2_net_buf_desc *to);*
- *void c2_net_desc_free(struct c2_net_buf_desc *desc);*

Commented [9]: Modified 7/18/2011 to be associated with a TM and not the domain. This reduces NUMA locking issues by not requiring the domain mutex during message receive. End points can only be created in started TMs. —carl_braganza

Event delivery

The operation of a transfer machine is asynchronous, and the module makes extensive use of event callbacks to notify the application of operation completion, change of state, etc. Events and callback function pointers definitions involve the following data structures:

```
struct c2_net_buffer_event {
    struct c2_net_buffer        *nbe_buffer;
    c2_time_t                    nbe_time;
    int32_t                      nbe_status;
    c2_bcount_t                  nbe_offset;
    c2_bcount_t                  nbe_length;
    struct c2_net_end_point     *nbe_ep;         /* C2_NET_QT_MSG_RECV only */
};

typedef void (*c2_net_buffer_cb_proc_t)(const struct c2_net_buffer_event *ev);
struct c2_net_buffer_callbacks {
    c2_net_buffer_cb_proc_t nbc_cb[C2_NET_QT_NR];
};

enum c2_net_tm_ev_type {
    C2_NET_EV_ERROR,
    C2_NET_EV_STATE_CHANGE,
    C2_NET_EV_DIAGNOSTIC,
    C2_NET_EV_NR
};

struct c2_net_tm_event {
    enum c2_net_tm_ev_type      nte_type;
    struct c2_net_transfer_mc  *nte_tm;
    c2_time_t                   nte_time;
    int32_t                     nte_status;
    enum c2_net_tm_state        nte_next_state;
    struct c2_net_end_point    *nte_ep;          /* on transition to C2_NET_TM_STARTED */
    void                       *nte_payload;
};

struct c2_net_tm_callbacks {
    void (*ntc_event_cb)(const struct c2_net_tm_event *ev);
};
```

There are two different types of events:

- Buffer completion events
- General transfer machine events

Buffer completion events are represented by the struct **c2_net_buffer_event** data structure. The transport invokes the **c2_net_buffer_event_post** subroutine to notify the application of the completion of a buffer operation, successfully or otherwise, or when a buffer operation has been cancelled by application use of the **c2_net_buffer_del** subroutine. A 0 value in the nbe_status field indicates that the buffer operation completed successfully; negative values provide the reason for failure - a value of **-ECANCELLED** is set for cancelled operations.

Separate callback subroutines are used for each type of buffer queue, as can be seen in the **c2_net_buffer_callbacks** structure, which is pointed to by the **c2_net_buffer** structure. The **c2_net_buffer_event** structure must never be freed by the application. It is typically allocated on the stack of the subroutine invoking the **c2_net_buffer_event_post** subroutine.

The vector of callback subroutines must be specified with the **nb_callbacks** pointer of a **c2_net_buffer** when the **c2_net_buffer_add** subroutine is invoked, or earlier if desired - the pointer is not modified by the messaging API. Multiple buffers can share the same set of callback subroutines.

There are three different categories of transfer machine events, distinguished by the value of the **nte_type** field set in the **c2_net_tm_event** structure. Other field usage varies by category, as follows:

Event Category	nte_type	Other valid fields	Notes

Transfer machine state change	`C2_NET_TEV_STATE_CHANGE`	`nte_tm`
		`nte_time`
		`nte_status`
		`nte_next_state`
		`nte_ep`
			A transfer machine enters the **C2_NET_TM_FAILED** state if a start operation fails, and must then be fini'd.
			The **nte_ep** field is set only on transition to the C2_NET_TM_STARTED state, and contains the pointer to the TM's end point to be set in the **ntm_ep** field.

Error	`C2_NET_TEV_ERROR`	`nte_tm`
		`nte_time`
		`nte_status`
			Used to report errors in a non-buffer context.
			For example, **-ENOBUFS** if messages were dropped due to a lack of receive message buffers.

Diagnostic	`C2_NET_TEV_TDIAGNOSTIC`	`nte_tm`
		`nte_time`
		`nte_status`
		`nte_payload`
			Transport specific diagnostic event. The **nte_payload** pointer may be set.

Transfer machine events are posted by the underlying transport invoking the **c2_net_tm_event_post** subroutine. The **c2_net_tm_event** structure must never be freed by a callback subroutine - it is typically allocated on the stack of the posting entity.

A callback function pointer is defined to a transfer machine during initialization (example shown in a following section). Multiple transfer machines can share this same structure.

Buffer management

Applications are totally responsible for network buffer management. The application must create **c2_net_buffer** structures for each desired purpose and register them with the transport using **c2_net_buffer_register**() - examples are show in sections below. Certain transport level optimizations may be possible with such pre-registration and declaration of intended use. Applications too can benefit by maintaining pools of buffers to handle memory in an optimal manner.

An application initializes the networking module as before (most applications initialize all of Colibri using **c2_init()** instead of initializing only the networking module), initializing the desired transport and a network domain. Then it registers buffers with the domain:

```
struct c2_net_buffer_callbacks app_buf_callbacks = {
    .nbc_cb = {
       [C2_NET_QT_MSG_RECV]          = app_m_recv_cb,
       [C2_NET_QT_MSG_SEND]          = app_m_send_cb,
       [C2_NET_QT_PASSIVE_BULK_RECV] = app_p_recv_cb,
       [C2_NET_QT_PASSIVE_BULK_SEND] = app_p_send_cb,
       [C2_NET_QT_ACTIVE_BULK_RECV]  = app_a_recv_cb,
       [C2_NET_QT_ACTIVE_BULK_SEND]  = app_a_send_cb
    },
};
struct c2_net_buffer *nb;
. . .
rc = c2_net_init();
rc = c2_net_xprt_init(&c2_net_lnet_xprt);
rc = c2_net_domain_init(&app_dom, &c2_net_lnet_xprt);

C2_ALLOC_PTR(nb);
/* allocate memory for nb in nb_buffer */
rc = c2_net_buffer_register(nb, &app_dom);
C2_ASSERT( nb->nb_flags & C2_NET_BUF_REGISTERED );
C2_ASSERT( nb->nb_dom == &app_dom );

nb->nb_callbacks = app_buf_callbacks;  /* prep for future use */
```

Commented [10]: It's convenient to place call-backs into a separate struct c2_net_tm_ops, which can be initialised once and shared by multiple machine instances (and make it a const pointer, see _ops fields in various other structures).—nikita_danilov

Good idea.
- Carl

Buffers need to be enqueued onto specific transfer machine queues to get utilized - operations are performed on the buffer as a side effect of being added to a queue. Buffers fields must be set in different ways, depending on their intended use, prior to adding them to a queue. When the operation involving a buffer gets completed, an event is posted on the buffer containing status and results. The following (logical) queues are defined:

C2_NET_QT_MSG_SEND
: This queue contains the buffers with messages to be transmitted.
The buffer is initialized with the message to send and the end point of the recipient.

C2_NET_QT_MSG_RECV
: This queue contains buffers in which messages will be received.
The buffer is initialized with memory in which to receive a messages. The

callback event contains the end point of the sender.

C2_NET_QT_PASSIVE_BULK_SEND
: This queue contains buffers with data to be transferred, in bulk, under control of a remote transfer operation. Data flows from this process to the remote process (i.e. the remote process “reads”).
The buffer is initialized with memory containing the data to be sent, and the end point of the remote end that will perform the operation. After adding the buffer to the queue, the network buffer descriptor is set.

C2_NET_QT_PASSIVE_BULK_RECV
: This queue contains buffers which are awaiting data, transferred in bulk, under control of a remote transfer operation. Data flows from the remote process to this process (i.e. the remote process “writes”).
The buffer is initialized with memory in which to receive the data and the end point of the remote end that will perform the operation. After adding the buffer to the queue, the network buffer descriptor is set.

C2_NET_QT_ACTIVE_BULK_SEND
: This queue contains buffers which contain data to be transferred, in bulk, under local process control.
The buffer is initialized with memory containing the data to be sent and the network buffer descriptor of the remote end.

C2_NET_QT_ACTIVE_BULK_RECV
: This queue contains buffers which are awaiting data, transferred in bulk, under local process control.
The buffer is initialized with memory in which to receive the data, and the network buffer descriptor of the remote end.

C2_NET_QT_NR
: Not strictly a queue *per se*, but the value is used as the size of queue related arrays.

Each type of buffer related event is mapped to a distinct callback subroutine. These callbacks are supplied in a separate **c2_net_buffer_callbacks** data structure which the application can use for multiple buffers on transfer machines if desired.

It is possible to request a cancellation of a buffer operation by calling the **c2_net_buffer_del**() subroutine - typically this would be done during the orderly shutdown of the application. Notification that the cancellation took place is through the buffer completion callback on that buffer, with the status set to **-ECANCELED**. However, since all operations are asynchronous, there is no guarantee that the operation will get cancelled - it is possible that it successfully completes. Either way, the application will get notified in a consistent manner. Note that while a a callback is active, the application owns the buffer, not the transfer machine. It is up to the application to ensure that it does not call **c2_net_buffer_del**() while in a callback. From the point of view of the transfer machine, the buffer is already removed from the queue and the operation is a no-op.

Initializing a transfer machine

A transfer machine is initialized as follows:

```
struct c2_net_tm_callbacks app_tm_callbacks = {
    .ntc_event_cb = app_tm_event_cb
};
struct c2_net_transfer_mc app_tm = {
    .ntm_state     = C2_NET_TM_UNDEFINED;
    .ntm_callbacks = app_tm_callbacks;
};
. . .
rc = c2_net_tm_init(&app_tm, &app_dom);
C2_ASSERT( app_tm.ntm_state == C2_NET_TM_INITIALIZED );
```

The application must start the transfer machine to commence operations. Servers and clients (assuming an application level RPC context) may start their transfer machines in different ways. Servers pass their well known end point address to the transfer machine during startup:

```
rc = c2_net_tm_start(&app_tm, "10.1.1.5:3492:3"); /* transport specific address */
```

Clients could create an end point with a dynamic end point address and then pass that to the start subroutine - this is transport specific.

> Current LNET usage in Lustre has almost fully defined end points for all clients; the process identifier and portal numbers are well defined, the networks are statically configured on each host, and only the IP addresses vary by host.

A client may choose to use dynamic end-point addressing, if supported by the chosen transport, as follows:

```
rc = c2_net_tm_start(&app_tm, NULL);
```

> Not planning on supporting dynamic end point addressing in the bulk emulation in-memory or sunrpc based transports.

Receiving messages

Applications must supply buffers in which to receive messages. Servers in particular must do this early on or clients won't be able to contact them. Clients only need to do this when they anticipate a response in the immediate future (such as prior to sending an RPC request message). There is a special error event callback defined to provide an indication of dropped messages (error code **-ENOBUFS**).

> The LNET API allows us to define *lazy portals*, that will buffer/retransmit messages if there is no available buffer. However, this is not a good idea for a system in steady state.

```
/* add a buffer to the receive queue */
C2_ASSERT( (nb->nb_flags & C2_NET_BUF_QUEUED) == 0 );
nb->nb_qtype = C2_NET_QT_MSG_RECV;
rc = c2_net_buffer_add(nb, &app_tm);
C2_ASSERT( nb->nb_flags & C2_NET_BUF_QUEUED );
C2_ASSERT( nb->nb_tm == &app_tm );
```

A message is delivered via a **C2_NET_QT_MSG_RECV** callback. The identity of the sender is in the **nbe_ep** field of the event. The **nbe_length** field of the event indicates the length of the

Commented [11]: A client might receive an unsolicited message (*e.g.*, a resource revocation call-back from a server) at almost any time. —nikita_danilov

Sure - but then it acts as a *server* in the traditional RPC role sense!
- Carl

message stored in the **c2_bufvec**, and the **nbe_offset** field the starting position of the message in the buffer. The application should process the message, and then add the buffer back to the receive queue. A client receiving a response message may choose not to add the buffer back to the receive queue, if it only adds such buffers just in time. The end point object's reference count will be decremented when the callback returns.

```
/* callback */
void app_m_recv_cb(const struct c2_net_buffer_event *ev)
{
  C2_ASSERT(ev->nbe_buffer != NULL && ev->nbe_buffer->nb_qtype == C2_NET_QT_MSG_RECV);
  /* process message - length is ev->nbe_length, starting at ev->nbe_offset in the buffer */
  ...
  /* server can "remember" client as follows ... */
  rc = c2_net_end_point_get(ev->nbe_ep);  /* bump ep reference count */
  safe_place->ep = ev->nbe_ep; /* stash pointer to client end point for response */
  …
  C2_ASSERT( (ev->nbe_buffer->nb_flags & C2_NET_BUF_QUEUED) == 0 );
  rc = c2_net_buffer_add(ev->nbe_buffer, ev->nbe_buffer->nb_tm); /* buffer back to receive queue */
  return;
}
```

Sending messages

To send a message, the application must know the end point of the recipient. Servers have well known end points, so clients can create the **c2_end_point** for the server using **c2_net_end_point_create**(), as shown earlier. Servers, when responding to messages from client, obtain the client end point from their message.

Sending a message requires that the application add a buffer onto the message send queue. The **nb_length** field of the **c2_net_buffer** indicates the length of the message stored in the **c2_bufvec**, and the destination is set by the end point in the **nb_ep** field. As a side effect, the buffer gets sent to the end point specified. An event will be generated when the message has been sent and the buffer may be freed - this is not necessarily a guarantee that the message has arrived.

```
void send_msg()
{
 . . .
 C2_ASSERT( (nb->nb_flags & C2_NET_BUF_QUEUED) == 0 );
 nb->nb_qtype = C2_NET_QT_MSG_SEND;
 nb->nb_ep = safe_place->ep; /* respond to client; ref count will get updated when buffer added */
 nb->nb_length = msg_length; /* actual message length */
 /* put message in nb->nb_buffer */
 rc = c2_net_buffer_add(nb, &app_tm); /* send the message */
 C2_ASSERT( nb->nb_flags & C2_NET_BUF_QUEUED );
 rc = c2_net_end_point_put(safe_place->ep); /* okay to release our copy of the end point now */
 safe_place->ep = NULL; /* must remember to zero pointer in buffer upon completion */
}

/* callback */
void app_m_send_cb(const struct c2_net_buffer_event *ev)
{
 C2_ASSERT(ev->nbe_buffer != NULL && ev->nbe_buffer->nb_qtype == C2_NET_QT_MSG_SEND);
 /* application now owns the buffer */
 C2_ASSERT( (ev->nbe_buffer->nb_flags & C2_NET_BUF_QUEUED) == 0 );
 ev->nbe_buffer->nb_ep = NULL; /* drop the pointer; count decremented after the callback returns */
 if (ev->nbe_status == 0) {
    /* success */
```

Commented [12]: Added to support transports that can deliver multiple messages in a single buffer. —carl_braganza

Commented [13]: implied, I believe is that application should not have references back to the buffer when it returns it to the queue —dave_cohrs

Applications manage the buffer so can keep pointers if they want; they shouldn't examine buffers that are in use by the net layer. —carl_braganza

Commented [14]: This isn't shown earlier, this is shown in `Bulk data transfer (passive side)' below. —valery_vorotyntsev

Commented [15]: LNET has ACK capabilities. This is a build or runtime choice. —carl_braganza

```
 }
 return;
}
```

Bulk data transfer (passive side)

Bulk data can be exchanged asynchronous to a message by sending a network buffer descriptor in the message, instead of including the data in the message. The process making a request to transfer (read or write) bulk data must add a **c2_net_buffer** initialized with a pointer to local memory and the end point of the remote process, on to the transfer machine's bulk data *passive* send or receive queue. The side effect of doing so is to create a network buffer descriptor for the local memory, which the application should then send to the remote process that will perform the bulk transfer. Note that the original process does not, itself, perform this operation - hence the use "passive". An event will be generated when the remote operation completes.

The following illustrates a read request which asks a remote server to write to a local buffer:

```
void read_req()
{
 struct c2_net_buf_desc nbd;
 struct c2_end_point *server_ep;
 . . .
 nb->nb_qtype = C2_NET_QT_PASSIVE_BULK_RECV; /* remote server will write to this buffer */
 /* get the server's end point */
 rc = c2_net_end_point_create(&server_ep, &app_dom, ...); /* supply server end point description */
 nb->nb_ep = server_ep;
 /* enqueue the data buffer in which to receive the bulk data; ep ref count incremented */
 rc = c2_net_buffer_add(nb, &app_tm);
 rc = c2_net_desc_copy(nb->nb_desc, &nbd);  /* recover descriptor from buffer */

 /* send descriptor in message to server */
}

void app_m_send_cb(const struct c2_net_buffer_event *ev)
{
 C2_ASSERT(ev->nbe_buffer != NULL && ev->nbe_buffer->nb_qtype == C2_NET_QT_MSG_SEND);
 c2_net_desc_free(&ev->nbe_buffer->nb_desc); /* free descriptor when message sent */
 /* application owns message buffer */
}

void app_p_recv_cb(const struct c2_net_buffer_event *ev)
{
 C2_ASSERT(ev->nbe_buffer != NULL && ev->nbe_buffer->nb_qtype == C2_NET_QT_PASSIVE_BULK_RECV);
 if (ev->nbe_status == 0) { /* success */
       c2_net_desc_free(&ev->nbe_buffer->nb_desc); /*free desc that was added by c2_net_buffer_add */
       /* put the data somewhere and release the buffer */
       /* data length is in ev->nbe_length, starting at ev->nbe_offset */
 } else { /* failure */
 }
 /* application now has control over the passive data buffer and end point object */
 c2_net_end_point_put(ev->nbe_buffer->nb_ep); /* release server ep */
}
```

Bulk data transfer (active side)

The recipient of the above message will extract the network buffer descriptor and initiate the

bulk data transfer operation (RDMA if supported by the transport) to send or receive data from its local memory to the remote process memory described by the descriptor. This is done by adding a **c2_net_buffer** initialized with the network buffer descriptor and the local memory pointer, to the transfer machine's bulk data *active* send or receive queue - the side effect of doing so is to initiate the asynchronous transfer operation. An event will be generated when the transfer completes.

```
void bulk_send() {
 ...
 nb->nb_qtype = C2_NET_QT_ACTIVE_BULK_SEND; /* send from this buffer to remote */
 c2_net_desc_copy(&nb->nb_desc, &nbd); /* copy descriptor from message into buffer */
 rc = c2_net_buffer_register(nb, &app_tm);
 rc = c2_net_buffer_add(nb); /* initiates the write */
}
/* callback */
void app_a_send_cb(const struct c2_net_buffer_event *ev)
{
 C2_ASSERT(ev->nbe_buffer != NULL && ev->nbe_buffer->nb_qtype == C2_NET_QT_ACTIVE_BULK_SEND);
 /* check status */
 /* app now in control of buffer and network descriptor */
 c2_net_desc_free(&ev->nbe_buffer->nb_desc);
}
```

Statistics

Statistical data is maintained individually for each queue. Timers are initialized when a buffer is added to a queue, and duration measured when the completion event is posted. No interpretation of the statistics is made by this layer - instead, the raw data is available for the upper protocol layers to retrieve and analyse.

The legacy statistics available through the domain object are deprecated.

Support is provided to atomically recover and reset statistical counters of a single queue, or of all queues together.

Capabilities and control

The **c2_net_domain_get_max_buffer_size**() subroutine is used to determine the maximum buffer size that the transport will support.

The **c2_net_domain_get_max_buffer_segments**() subroutine is used to determine the maximum number of segments that the transport will support in a buffer vector.

Diagnostics

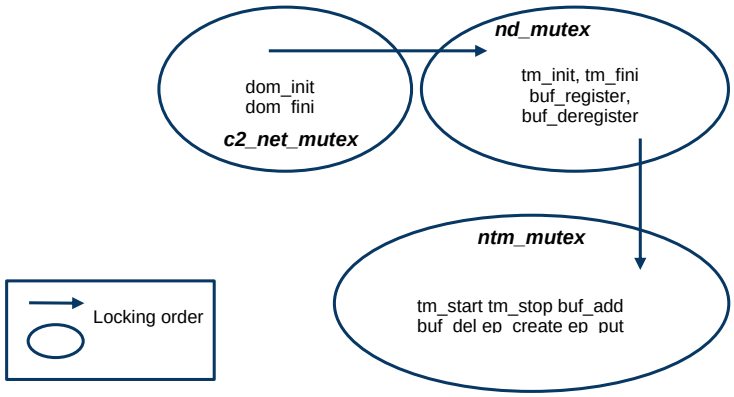
Support for diagnostic applications is provided by the Networking module:

- Diagnostic events can be posted by the transport. They are distinguished by the **nte_type** field value being set to **C2_NET_TEV_DIAGNOSTIC**. The transport can use the **nte_payload** field as desired, or even embed the entire **c2_net_tm_event** data structure in a container structure.

Logical specification

Mutex usage by the messaging layer subroutines and their locking order is illustrated in the figure below.

Mutex usage and locking order



Any of the mutexes can be directly obtained, but nested locking is only permitted in the order illustrated. When nested locking is used, it is permitted to drop the upper locks while holding a lower lock, but the upper locks cannot be re-obtained without first releasing the lower lock. Note that the transfer machine's existence is partially protected by its state - it cannot be fini'd once started, until it is first stopped, and all associated end point objects released. Similarly, buffer existence is protected by state flags.

Most internal serialization is performed using either the **nd_mutex** in the **c2_net_domain**, or the **ntm_mutex** in the transfer machine. The transfer machine mutex is used for all transfer machine operations involving a transfer machine whose state is greater than or equal to **C2_NET_TM_STARTING** and less than **C2_NET_TM_STOPPED**, and for end point management. The domain mutex is used otherwise.

This allows a transport to implement processor affinity for buffer operations. i.e. the transport could make buffer completion callback on the same processor that was used to initiate the operation. A higher level application could create one transfer machine per processor.

Note: prior to 7/18/2011 end point objects were anchored in the domain, and protected by the domain mutex. This worked fine for all end point usage except message receive, where the transfer machine had to obtain the domain mutex internally to find or create an end point object to represent the incoming message source. This requirement gets eliminated with the shift to end points being anchored in transfer machines, allowing transfer machines to operate totally independent of the domain mutex in steady state. The cost to applications is that end points are no longer usable across transfer machines in the domain; however, since the **nep_addr** string of

one end point object can easily be used to create an end point object in any other TM, this is not such a big problem. Applications also need to release all end points before fini'ing a transfer machine.

Higher level protocol software should not reference a buffer once it has been added to an operation queue. Instead, it should wait for the event callback on the buffer to determine when it is safe to de-reference. It is guaranteed that events delivered for a given buffer will be serialized. In general, however, multiple concurrent events can be delivered for a transfer machine.

Most of the services offered by the transfer machine are implemented by the underlying transport. The general pattern followed involves the use of registered **c2_net_buffer** data structures and the operational queues maintained in the **c2_net_transfer_mc** data structure, as follows:

- A registered buffer is initialized with the correct fields (described in the functional specification) for one of the six possible operations (message send/recv, remote data read/write, local data send/recv)

- The buffer is added to a logical queue in the transfer machine by invoking **c2_net_buffer_add**(). The buffer's **nb_dom** field must match the transfer machine's **tm_dom** field, and the buffer's **C2_NET_BUF_REGISTERED** flag must be set, and the **C2_NET_BUF_QUEUED** and **C2_NET_BUF_IN_USE** flags must not be set.

- The **nb_callbacks** field must be set and the callback for the value of **nb_qtype** must be defined.

- Buffer addition operations are only permitted when the transfer machine state is **C2_NET_TM_STARTED**. The exception is the addition of buffers to the **C2_NET_MSG_RECV** queue, which may be done in any of the states leading to the started state (i.e. any time after the transfer machine is initialized and before it is stopped).

- The **c2_net_buffer_add**() subroutine obtains the transfer machine lock, performs the above checks, and then adds the buffer to the appropriate list, clears the **C2_NET_BUF_IN_USE** flag, sets its **C2_NET_BUF_QUEUED** flag, and sets its **nb_add_time** to the current time. If a **c2_end_point** is associated with the buffer operation, then its reference count is incremented. Finally, the subroutine will invoke the transport's **xo_buf_add**() method on the buffer to initiate the desired operation, then release the lock and return.

  The transport's **xo_buf_add**() method is responsible for initiating the concerned operation; the method should not block. The transport method is required to translate the **c2_bufvec** scatter-gather buffer data structure into its own internal form (possibly a struct **iovec**) and do whatever is required to perform the desired operation. The transport is permitted to attach operation state on the private data pointer in registered

buffers. It is responsible for remembering the following pieces of information which will be required to post an event upon completion:

- the struct **c2_net_transfer_mc** pointer
- the struct **c2_net_buffer** pointer

Buffers involved in passive bulk data send or receive must return **c2_net_buf_desc** data structures; this data structure will be defined in a FOP format file (.ff suffix) as an opaque sequence of bytes to make XDR serialization layers available. Transports are required to encode their internal network buffer descriptors to a byte order independent opaque buffer, then serialize it into the **c2_net_buf_desc** data structure for return, using these XDR routines. Applications can copy this descriptor using the **c2_net_desc_copy**() subroutine, and send it to the remote process. The descriptor data is freed using **c2_net_desc_free**().

Buffers involved in active bulk data send or receive must be initialized with the **c2_net_buf_desc** data structure set in the passive **c2_net_buffer**. Transports should be capable of deserializing this data and decoding the content.

At least the following information should be encoded in a network buffer descriptor:

- Identity of the remote buffer
- End point of the remote buffer owner (if the connection is multiplexed between multiple transfer machines)
- End point of the intended target operation initiator to enforce security, if possible
- Direction of data transfer
- Buffer size

The **xo_buffer_add**() methods must also remain cognizant of the transfer machine state. While it is permitted to add receive buffers before the transfer machine enters the **C2_NET_TM_STARTED** state, no messages are to be received until this transition completes.

The **xo_buffer_add**() methods should arrange for the buffer's **C2_NET_BUF_IN_USE** bit to be set when use of the buffer commences, and to clear it when the buffer operation completes.

- When the operation involving the buffer completes, the transport is responsible for invoking the **c2_net_buffer_event_post**() subroutine to notify the transfer machine of the completion. Events involving buffers will identify the buffers. Events involving received messages should have **c2_end_point** data structures to describe the end point; the transport should use the **c2_net_end_point_create**() subroutine to fetch or create the end point.

  Operation completion is defined by the following circumstances:

  - Successful termination of the requested operation

- Failure of the requested operation
- Expiry of the timeout period specified in the **nb_timeout** field of the buffer before completion. ***Support for timeouts is transport specific.***

Transports should be aware that the application is permitted to make re-entrant calls into the transport from this callback.

- The **c2_net_buffer_event_post**() subroutine will obtain the transfer machine lock. It has to search for and unlink the buffer from its logical queue and unset its **C2_NET_BUF_QUEUED**, **C2_NET_BUF_IN_USE** and **C2_NET_BUF_CANCELLED** flags. Next the subroutine will update the statistics for the queue, and increment the transfer machine's callback counter. The subroutine then ***releases*** the transfer machine lock, and invokes the appropriate, buffer callback.

  Releasing the lock before making the callback is crucial as it allows the application callback to make further calls into the transfer machine. For example, a receive buffer may be re-queued in this manner. Note that the underlying transport should also be re-entrant in this manner as it is normally the entity that calls the **c2_net_buffer_event_post()** subroutine.

  When the call back returns, the subroutine will re-obtain the lock, decrement the reference count of any associated end point in the cases where **c2_net_buffer_add** incremented the count, and of the transfer machine's callback counter itself. A signal will be sent on the transfer machine's condition variable to wake up pending waiters. The subroutine will then broadcast to all waiters on the **tm_chan** channel (this makes it easier for applications to implement blocking semantics when desired). Then the subroutine will release the lock and return.

- The **c2_net_tm_event_post**() subroutine is similar to the **c2_net_buffer_event_post** subroutine, though simpler as it does not involve buffers. It is not combined with the buffer post mainly for performance optimization. The use of the wakeup on the tm_chan channel is particularly useful for state transition events - it is important, however to test the condition when awoken, as concurrent buffer completion also signals on the same channel.

  Note that as the transport is required to post events, and the lifetime of an event data structure is that of the post invocation only, transports can embed this data structure into a transport specific containing structure to provide transport specific data to the upper layers - this may be useful to diagnostic applications.

The **c2_net_buffer_del**() subroutine obtains the transfer machine lock if queued. It then checks to see if the **C2_NET_BUF_QUEUED** flag is set, and if so it will invoke the transport's **xo_buf_del**() method to cancel the operation, if possible. The subroutine releases the lock and returns. The buffer completion callback will be invoked when the buffer is released - return does not imply this. There is no guarantee that the callback will indicate cancellation as the operation

may have completed successfully during this time, or the transport does not support cancellation. Also, it is up to the application to ensure that it does not call **c2_net_buffer_del**() while the buffer is being processed by a buffer callback. In this case, because the buffer is not queued, no action is taken by **c2_net_buffer_del**().

The **c2_net_end_point_create**() subroutine can only be used on a transfer machine in the **C2_NET_TM_STARTED** state. It obtains the transfer machine lock and then invokes the transport specific **xo_end_point_create**() method to perform the operation. This method scans the list of defined end points maintained in the transfer machine's **ntm_end_points** list of **c2_net_end_point** data structures, looking for a match. If it finds one, it increments the end point reference count and returns the pointer to the data structure. If it does not find a match, it allocates a new data structure, initializes it with its own **release**() method, add it to the transfer macine's list, and then return its pointer. Upon return of the transport method, the **c2_net_end_point_create**() subroutine will unlock the transfer machine and return.

The **c2_net_end_point_get**() subroutine asserts that the count of the embedded **c2_ref** data structure is not zero, and then calls **c2_ref_get**() to increment it further. The transfer machine lock is not required. The **c2_net_end_point_put**() subroutine obtains the transfer machine lock, then calls **c2_ref_put**(), which will invoke the **release**() method when the count goes to zero. The release method, which is private to the transport, will unlink the end point from the transfer machine, free the memory associated with the end point, along with other transport specific resources. The transfer machine lock is required by **c2_net_end_point_put**() to serialize the actions of the **release**() subroutine which has to unlink the end point from the transfer machine; the **release**() subroutine itself could not obtain this lock, as it would be in a race condition with the concurrent creation of an identical end point through **c2_net_end_point_create**(). All end points must be released by the application before the transfer machine is fini'd.

> Note that as the transport methods are responsible for allocating and freeing the end point data structure, the transport should, therefore, embed the **c2_net_end_point** data structure in its own data structure to maintain private data for each connection. No private field will be allocated for transport use in the **c2_net_end_point** data structure.

The **c2_net_buffer_register**() subroutine will grab the network domain lock and invoke the underlying transport **xo_buffer_register**() method to do whatever it takes to register the buffer. Then the subroutine will set the **C2_NET_BUF_REGISTERED** flag and add the buffer to the list of registered buffers maintained in the domain, unlock the domain and return.

> Need to work out what optimizations the LNET transport can do here. Currently the kernel implementation has no support for this, nor has the GPL'd user space version (which we won't be using).
>
> It is possible to use an additional kernel module to pin these buffer pages. However great care should be taken to free them upon process termination.

The **c2_net_buffer_deregister**() subroutine will grab the domain lock, check if the **C2_NET_BUF_REGISTERED** flag is set and that the buffer is on its queue. If any other buffer flag is set, it will fail the operation. It invokes the transport **xo_buffer_deregister**() method to

do whatever it takes to deregister the buffer. Then the subroutine will clear the **C2_NET_BUF_REGISTERED** flag, unlink the buffer from the registered buffer list, unlock the domain and return.

The **c2_net_tm_init** subroutine grabs the domain lock, then initializes the **c2_net_transfer_mc** data structure, including its embedded condition variable, and then calls the transport specific **xo_tm_init**() method. Upon completion, it links the transfer machine into the list of transfer machines held by the domain, releases the lock and returns.

The **c2_net_tm_start**() subroutine grabs the transfer machine lock, and then validates that the transfer machine is in the **C2_NET_TM_INITIALIZED** state. The subroutine will internally create an end point for the supplied address, but not expose the end point via the **ntm_ep** field yet. Next, the subroutine sets the transfer machine state to **C2_NET_TM_STARTING**, and calls the **xo_tm_start**() method and then releases the lock and returns. The **xo_tm_start**() method should do what it takes to enable network communication and expose the specified end point on the network. It should not block the invoker while these operations take place, but instead should post a state transition event to the **C2_NET_TM_STARTED** state upon completion. If the transfer machine fails, it will release the end point reference, and if it succeeds, it will set **ntm_ep** to point to the end point, with a guaranteed reference count of 1.

The **c2_net_tm_stop**() subroutine grabs the transfer machine lock , and then validates that the transfer machine is in on of the **C2_NET_TM_INITIALIZED**, **C2_NET_TM_STARTING** or **C2_NET_TM_STARTED** states. Then, it then sets the state to **C2_NET_TM_STOPPING** and calls the **xo_tm_stop()** method. The method should cancel the transition to the starting state, if any, and initiate the shutdown of the transport interface for this transfer machine (only). Incoming messages are to be stopped and the method is provided an indicator of whether pending operations are to be cancelled or drained. The method should not block the invoker, but instead should a state transition event to the **C2_NET_TM_STOPPED** state upon completion.

At any time, if the transfer machine encounters a fatal internal error, it should transition itself to the **C2_NET_TM_FAILED** state. This could happen during startup, when running or even during shutdown.

The **c2_net_tm_fini**() subroutine grabs the network domain lock. If the transfer machine is in the **C2_NET_TM_FAILED** or **C2_NET_TM_STOPPED** states, the transfer machine callback count is 0, all transfer machine buffer lists are empty, and the end point list contains at most the transfer machine end point with a reference count of 1, then the transport **xo_tm_fini**() method is called. The transfer machine then grabs its transfer machine lock, decrements its **ntm_ep** refcount (if set), sets its pointer to NULL, and releases the transfer machine lock. The transfer machine is then removed from the list in the network domain. Note that the transfer machine's callback counter ensures that there are no callbacks in progress.

The existing **c2_net_dom_init**() subroutine initializes the **c2_net_domain** data structure and its embedded mutex, and then calls the transport specific **xo_dom_init**(). The existing **c2_net_dom_fini**() calls the **xo_dom_fini**() method then cleans up the **c2_net_domain** data

structure. Both subroutines will obtain the global **c2_net_mutex** prior to calling the transport methods, but the **nd_mutex** lock is not held at this time. The **c2_net_dom_init** subroutine should be extended to fail if there are transport machines, end points or registered buffers still defined.

The existing **c2_net_init**(), **c2_net_fini**(), **c2_net_xprt_init**() and **c2_net_xprt_fini**() subroutines are currently no-ops. The **c2_net_init**() and **c2_net_fini**() subroutines must be extended to register the FOP data structures used, and initialize and fini the global **c2_net_mutex**. There is no reason to add additional functionality to the other subroutines at this point.

The **c2_net_domain_get_max_buffer_size**(), **c2_net_domain_get_max_segment_size**() and the **c2_net_domain_get_max_buffer_segments**() subroutines are implemented over corresponding **xo**_ methods, an ioctl style interface. They are invoked while holding the domain mutex.

Conformance

[r.rpccore.flexible] supports synchronous and asynchronous communication
: The proposal directly supports asynchronous message passing. Synchonous message passing is not directly required but can be implemented above this layer.

[r.rpccore.flexible] supports sending various auxiliary information
: The proposal sends opaque data - it does not care about the content.

[r.rpccore.efficient] the network bandwidth is utilized fully by sending concurrent messages
: The proposal allows concurrent messages to be in flight, and allows for queues to be maintained so that the underlying transport can handle messages in an optimal manner.

[r.rpccore.efficient.bulk] 0-copy, if provided by the underlying network transport, is utilized
: Allows for bulk data transfer using RDMA if the transport supports it.

[r.messaging.oneway] support for one-way messaging
: One way messaging is supported.

[r.messaging.efficient.buffer-use] support for efficient management of buffers
: Buffer management is handled by the upper layer protocols. A registration mechanism is provided to potentially optimize the kernel's handling of buffers reserved for network data.

[r.messaging.processor_affinity] Provide support for optimal processor affinity allocation.
: Each transfer machine has its own mutex which provides the ability to localize all active transfer machine operation to a specific processor, in a suitably "smart" transport.

State

State, events, transitions

Transfer machines exist in one of the following states:

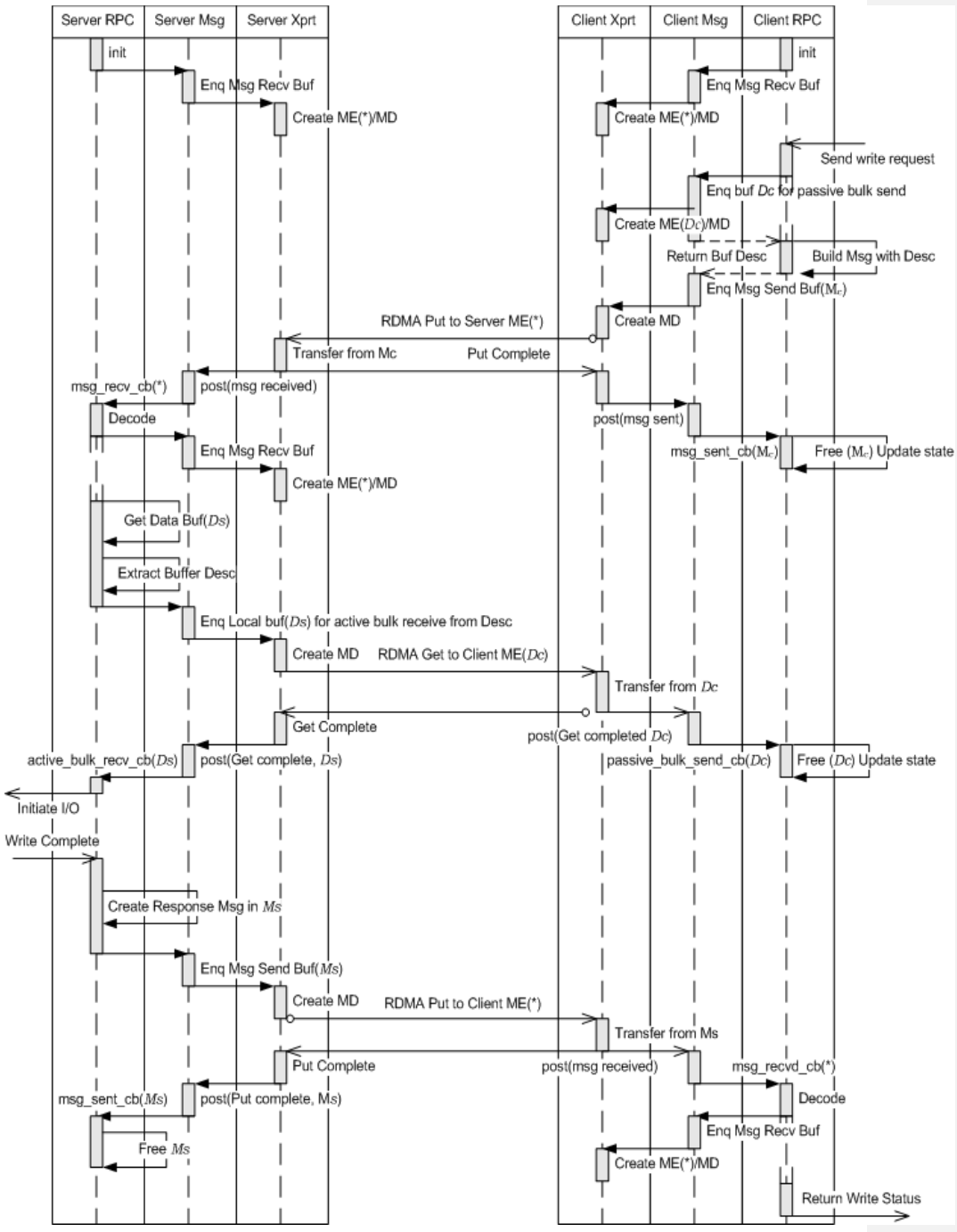
State	Description	Next State
C2_NET_TM_UNDEFINED	The state of an uninitialized transfer machine.	**C2_NET_TM_INITIALIZED**
C2_NET_TM_INITIALIZED	The state after initialization but before starting.	**C2_NET_TM_STARTING, C2_NET_TM_UNDEFINED**
C2_NET_TM_STARTING	The state after starting is initiated but before it completes. The TM mutex must be used for all TM related operations.	**C2_NET_TM_STARTED, C2_NET_TM_STOPPING**
C2_NET_TM_STARTED	The active, operational state. The TM mutex must be used for all TM related operations. End points can be created only when in this state.	**C2_NET_TM_STOPPING**
C2_NET_TM_STOPPING	The state after shutdown is initiated but before it completes. The TM mutex must be used for all TM related operations.	**C2_NET_TM_STOPPED**
C2_NET_TM_STOPPED	The state when activity is stopped.	**C2_NET_TM_UNDEFINED**

State transition events are posted using the **ntm_cb** callback. These events are distinguished from others by the **nte_type** value set to **C2_NET_TEV_STATE_CHANGE**. The **nte_tm** value identifies the transfer machine, and the **nte_next_state** value is set to the transition state.

Use cases

Scenarios

The typical message passing handshake required to implement an RPC write with asynchronous bulk data transfer is illustrated in the sequence diagram below.


Server RPC
Server Msg
Server Xprt
Client Xprt
Client Msg
Client RPC
init
Enq Msg Recv Buf
Create ME(*)/MD
Send write request
Enq buf Dc for passive bulk send
Create ME(Dc)/MD
Return Buf Desc
Build Msg with Desc
Enq Msg Send Buf(Mc)
RDMA Put to Server ME(*)
Create MD
Transfer from Mc
Put Complete
msg_recv_cb(*)
post(msg received)
Decode
post(msg sent)
msg_sent_cb(Mc)
Free (Mc) Update state
Get Data Buf(Ds)
Extract Buffer Desc
Enq Local buf(Ds) for active bulk receive from Desc
RDMA Get to Client ME(Dc)
Transfer from Dc
Get Complete
post(Get completed Dc)
active_bulk_recv_cb(Ds)
post(Get complete, Ds)
passive_bulk_send_cb(Dc)
Free (Dc) Update state
Initiate I/O
Write Complete
Create Response Msg in Ms
Enq Msg Send Buf(Ms)
RDMA Put to Client ME(*)
Transfer from Ms
Put Complete
post(msg received)
msg_recvd_cb(*)
msg_sent_cb(Ms)
post(Put complete, Ms)
Decode
Free Ms
Return Write Status

Initially, both the client and server establish receive buffers. A message is sent to an anonymous receive buffer on the other side. As soon as possible, the receiver should return the buffer back to the receive queue for reuse.

Notice how both client and server are event driven - they do not wait for network I/O to complete. Note too how one-way messaging is possible - there is no requirement of the network layer that a response message be sent.

Notice how the client encodes a descriptor of its data in its write request. The server will initiate a transfer of this data and then do something else until it arrives.

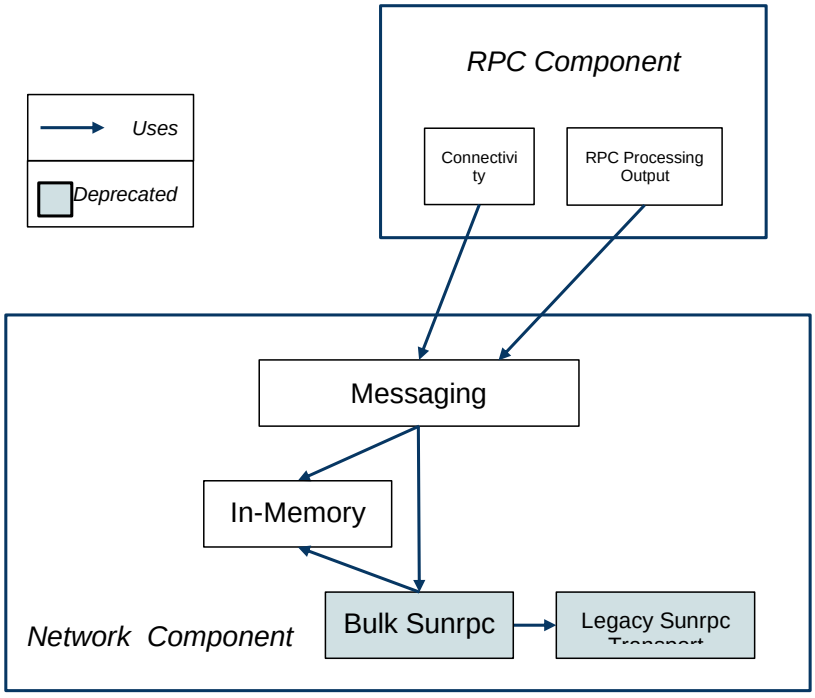
An LNET like transport is used for illustration purposes, hence the reference to MD/ME/Get/Put. It is assumed that the “anonymous” message buffers are hooked off an MD with a well defined set of match bits. Data buffers, on the other hand, would use session specific match bits that are encoded in the buffer descriptor.

References

[1] AR of RPC layer
[2] HLD of RPC layer core
[3] HLD of FOP:core wire formats
[4] LNET API Documentation (Lustre Wiki)

Sunrpc and in-memory bulk transport emulation

Two transports will be built. The first is an in-memory transport that will provide communication within a single process, facilitating early adoption and testing of the messaging layer. The second transport will provide communication across processes and hosts, by building a bulk emulation transport over the existing (and now deprecated) Sunrpc transport. This of course results in the new bulk transport being defined as deprecated right out of the box, but that’s the way it is!



The in-memory transport will provide support for message and bulk data transfer across network domains within the same process. The same code will also serve as the base code from which the sunrpc bulk emulation transport is derived, because the latter can share the same buffering and threading support.

The in-memory transport will not support the use of the **nb_offset** field in the **c2_net_buffer** structure. The value must be set to 0.

The existing sunrpc code has different kernel and user space implementation, and there is no server support or asynchronous client in the kernel today. This has to be rectified as it is necessary for message passing.

The new bulk sunrpc code will be written in a kernel/user space agnostic manner. The code will be referenced from the existing net/usunrpc and net/ksunrpc directories, and linked into those modules. We will not be using the RFC5666 support, but instead emulating the c2_net interfaces above over normal TCP/IP.

One constraint imposed by the existing usunrpc implementation (through its underlying SUN ancestry) is that only one server is permitted in a single process. The SUN rpc library uses globals to track the select system call fd set, so there is very little one can do to fix this issue.

The design then is to use a singleton server for the bulk sunrpc transport, shared by all the transfer machines. This singleton sits in its own private {uk}sunrpc network domain, and is only used to handle incoming RPC calls; each TM has its own private {uk}sunrpc network domain for client usage. The addressing used for end points is now a 3-tuple of (host, port, service id), where service id is an integer assigned to each TM during configuration, representing the service provided by the TM. (This is similar in nature to LNET portal numbering). All of the transfer machines within the process use the same (host, port) combination, regardless of the associated network domain.

Functional specification

The component must adhere to the transport specifications defined by the messaging component.

The following will be defined in **net/bulk_sunrpc.h**:

```
struct c2_net_xprt c2_net_bulk_sunrpc_xprt;
```

The following will be defined in **net/bulk_mem.h**:

```
struct c2_net_xprt c2_net_bulk_mem_xprt;
```

The actual data structures used to implement these transports are found primarily in a bulk emulation sub-directory, in the file **net/bulk_emulation/mem_xprt.h**. Headers specific to the sunrpc based emulation will be found in **net/bulk_emulation/sunrpc_xprt.h**. The data structures shown below are not organized by file location but instead by logical order.

The data structures use the field prefix notation of the c2_net data structures they are associated with, with the leading “n” replaced by “x”. For example, “ntm_” becomes “xtm_” for transfer machine private fields. This makes it easy to determine the context in spite of the embedded nature of the data structures.

Private domain data (set via **nd_xprt_private**) will contain at least the following:

```
typedef void (*c2_net_bulk_mem_work_fn_t)(struct c2_net_transfer_mc *tm,
                                          struct c2_net_bulk_mem_work_item *wi);
struct c2_net_bulk_mem_domain_pvt {
    struct c2_net_domain       *xd_dom;
    c2_net_bulk_mem_work_fn_t   xd_work_fn[C2_NET_XOP_NR]; /* work functions */
    size_t                      xd_sizeof_ep;
    size_t                      xd_sizeof_tm_pvt;
    size_t                      xd_sizeof_buf_pvt;
    size_t                      xd_addr_tuples;
    size_t                      xd_num_tm_threads;
    struct c2_list_link         xd_dom_linkage;
    struct c2_atomic64_t        xd_buf_id_counter;
};
struct c2_net_bulk_sunrpc_domain_pvt {
    struct c2_net_bulk_mem_domain_pvt  xd_base;
    struct c2_net_xprt_ops            *xd_base_ops;
    struct c2_net_domain               xd_rpc_dom;  /* the underlying {uk}sunrpc domain */
    c2_time_t                          xd_ep_release_delay;
    struct c2_list                     xd_conn_cache; /* struct c2_net_bulk_sunrpc_conn */
};
```

The bulk sunrpc domain private data embeds the base bulk memory private data, and keeps a copy of the original base transport op and work function pointers (not shown). It also keeps the domain data structure for the underlying sunrpc domain used for client rpc calls only.

The shared common service related data structures are defined by the following module-private globals:

```
struct c2_rwlock          sunrpc_server_lock;       /* for TM list serialization */
struct c2_mutex           sunrpc_server_mutex;      /* for service start/stop serialization */
struct c2_net_domain      sunrpc_server_dom;
struct c2_net_service_id  sunrpc_server_id;
struct c2_net_service     sunrpc_server_service;
struct c2_list            sunrpc_server_tms;        /* list of transfer machines */
uint32_t                  sunrpc_server_active_tms; /* count of active transfer machines */
struct c2_mutex           sunrpc_tm_start_mutex;    /* for TM start serialization */
```

Private transfer machine data (set via **ntm_xprt_private**) will contain at least the following:

```
enum c2_net_bulk_mem_tm_state {
   C2_NET_XTM_UNDEFINED=0,
   C2_NET_XTM_INITIALIZED,
   C2_NET_XTM_STARTING,
   C2_NET_XTM_STARTED,
   C2_NET_XTM_STOPPING,
   C2_NET_XTM_STOPPED,
   C2_NET_XTM_FAILED
};
struct c2_net_bulk_mem_tm_pvt {
    struct c2_net_transfer_mc         *xtm_tm;
    enum c2_net_bulk_mem_tm_state      xtm_state;
    struct c2_list                     xtm_work_list;
    struct c2_cond                     xtm_work_list_cv;
    uint32_t                           xtm_callback_counter;
    struct c2_thread                  *xtm_worker_threads;
    size_t                             xtm_num_workers;
};
struct c2_net_bulk_sunrpc_tm_pvt {
    struct c2_net_bulk_mem_tm_pvt      xtm_base;
    struct c2_list_link                xtm_tm_linkage; /* link on sunrpc_server_tms list */
};
```

The bulk sunrpc domain private transfer machine data embeds the base in-memory transport's transfer machine private data, and adds a **c2_service** structure to define a sunrpc server for the transfer machine.

The following structure defines work items handled by background threads:

```
enum c2_net_bulk_mem_work_opcode {
   C2_NET_XOP_STATE_CHANGE,     /* state change callback */
   C2_NET_XOP_CANCEL_CB,        /* buffer operation cancelled callback */
   C2_NET_XOP_MSG_RECV_CB,      /* message received callback */
   C2_NET_XOP_MSG_SEND,         /* message send processing */
   C2_NET_XOP_PASSIVE_BULK_CB,  /* passive bulk buffer completion callback */
   C2_NET_XOP_ACTIVE_BULK,      /* active bulk processing (send/recv) and callback */
   C2_NET_XOP_ERROR_CB,         /* generic error callback */
   C2_NET_XOP_NR
};
struct c2_net_bulk_mem_work_item {
    struct c2_list_link               xwi_link;
```

```
    enum c2_net_bulk_mem_work_opcode  xwi_op;
    enum c2_net_bulk_mem_tm_state     xwi_next_state;
    int32_t                           xwi_status;
    c2_bcount_t                       xwi_nbe_length;
    struct c2_net_end_point          *xwi_nbe_ep;
};
```

The following private data is attached to a buffer (via **nb_xprt_private**) during registration:

```
struct c2_net_bulk_mem_buffer_pvt {
    struct c2_net_buffer         *xb_buffer;
    struct bulk_mem_work_item     xb_wi;     /* work description and link */
    int64_t                       xb_buf_id;
    bool                          xb_cancel_requested;
    struct c2_net_buffer_event    xb_event;
};
struct c2_net_bulk_sunrpc_buffer_pvt {
    struct c2_net_bulk_mem_buffer_pvt xsb_base;
    struct sockaddr_in                xsb_peer_sa;
};
```

A work item is embedded in the buffer private data. The **xb_buf_id** is an identifier given to each passive buffer, and encoded into the network buffer descriptor.

The following defines an end point for these transports:

```
struct c2_net_bulk_mem_end_point {
   uint64_t                   xep_magic;
   struct sockaddr_in         xep_sa;
   uint32_t                   xep_service_id;
   struct c2_net_end_point    xep_ep;          /* embedded end point */
   char                       xep_addr[36];    /* printable address */
}

/* domain cache for sunrpc connections */
struct c2_net_bulk_sunrpc_conn {
    uint64_t                        xc_magic;
    struct c2_net_domain           *xc_dom;
    struct c2_ref                   xc_ref;
    struct c2_list_item             xc_dp_linkage; /* through domain pvt */
    struct sockaddr_in              xc_sa;
    uint32_t                        xc_service_id;
    struct c2_service_id            xc_sid;
    struct c2_atomic64              xc_last_use;
    bool                            xc_in_use;
    bool                            xc_sid_created;
    bool                            xc_conn_created;
};
/* per TM end point */
struct c2_net_bulk_sunrpc_end_point {
   uint64_t                         xep_magic;
   struct c2_net_bulk_mem_end_point  xep_base;
};
```

The in-memory transport uses standard TCP/IP address and port information to identify end points. This is stored in a struct **sockaddr_in**. The printable address is stored in the **xep_addr** array, allocated contiguously within this structure, and will be formatted as “dotted-ip-address-string:port-number”. (The largest dotted-ip (255.255.255.255) is 15 bytes, and the largest short is 5 bytes long, so the field fits within the 36 bytes allocated. This would have to be adjusted if IPv6 is supported.) The **xep_service_id** field is always set to 0 by the in-memory transport; it

is provided for use by derived classes, notably the sunrpc based bulk emulation transport. Adequate space is allocated in the **xep_addr** structure to include this number - the in-memory routines will add this number to the printable description only if not zero, resulting in the format "dotted-ip-address-string:port-number:service-id". The domain private field **xd_addr_tuples** field is set to 2 by default for the in-memory transport. Derived transports could set it to 3 if desired.

The bulk sunrpc transport uses the service identifier number in the end point, resulting in a 3-tuple address of (host, port, service id). The bulk sunrpc transport also adds a **c2_service_id** structure to the end point. This is used to finally map to a **c2_net_conn** data structure of the underlying sunrpc transport. The underlying sunrpc transport has its own reference counting mechanism for this data structure, which has to be honored: The connection is created with **c2_net_conn_create**. Each internal use should use the **c2_net_conn_find**() subroutine to obtain a pointer to the possibly shared connection, which should be released using **c2_net_conn_release**() when done. This does not free the object however; a call to **c2_net_conn_unlink**() must be invoked to do so.

There is a further wrinkle in the use of the underlying sunrpc transport connections. Unlinking a **c2_net_conn** connection immediately will result in the socket being closed. If the end point is accessed again very soon, a new socket will be created using a different local port, as the previously used local port remains in a TIMED_WAIT for approximately 4 minutes. It is possible to rapidly exhaust the available local ports in this manner. To alleviate this problem, we must introduce a delay in unlinking **c2_net_conn** data structures to allow for re-use within a few seconds. Thus, the lifecycle of a **c2_net_conn** cannot exactly match that of a **c2_net_end_point** object.

The API requires us to support end pointers per transfer machine as of 7/18/2011; prior to this, the end points were shared across the domain with a single sunrpc domain per bulk domain providing client connections, which is goodness as the existing sunrpc domain is heavy weight, with a number of threads and multiplicity of TCP sockets per connection. We were able to directly associate each end point with a **c2_net_conn** and handle the delayed release via tracking information in the end point itself. Unfortunately, this does not work with the new requirements.

With the TM constrained end points, a further level of indirection is introduced: End points track only the service id of the underlying sunrpc connection. The actual connection object is obtained via a domain global cache of **c2_net_bulk_sunrpc_conn** structures whose existence implies that a **c2_net_conn** data structure has been created for that service id. Convenience routines are provided to get and release the connections via this cache.

The **c2_net_bulk_sunrpc_conn** structure carries an **xep_last_use** field which will record the absolute time at which the connection was last used. This information is used to decide when to fini the network connection. The value is expressed with an atomic variable instead of a **c2_time_t** to avoid acquiring the domain mutex when updating. A skulker thread periodically scans the list of cached **c2_net_bulk_sunrpc_conn** structures and unlinks those that have not

been recently used. The **c2_net_end_point** release method does not operate on the global connection cache.

The in-memory transport defines the following network descriptor:

```
struct mem_desc {
   struct sockaddr_in     md_active;
   struct sockaddr_in     md_passive;
   c2_bcount_t            md_len;
   enum c2_net_queue_type md_qt;
   int64_t                md_buf_id;
};
```

This structure is copied without encoding into the buffer descriptor.

The in-memory transport maintains a private list to track the multiple network domains used for communication. The following is defined in **net/bulk_emulation/mem_xprt_pvt.h**:

```
extern struct c2_list    c2_net_bulk_mem_domains;
```

To protect this list we introduce a mutex in the messaging layer - this mutex must also be used to protect the domain init and fini (see the mutex scoping diagram in the Logical Specification of the messaging layer). The following will be defined in **net/net_internal.h**:

```
extern struct c2_mutex  c2_net_mutex;
```

The mutex will be initialized in the **c2_net_init**() subroutine, and fini'd in the **c2_net_fini**() subroutine, both of which are invoked by the Colibri initialization mechanism.

Logical specification

RPC procedures

The Bulk Sunrpc module will be an RPC client-server application. As such, it needs to define RPC procedure numbers for message posting, bulk data "put" and bulk data "get". These low level RPC calls will be used to emulate message passing and RDMA, and are not visible to higher levels. Define the format of the network buffer descriptor. As these subroutines require XDRs, represent them with FOPs and use the **fop2c** compiler will be used to construct them from the following.

The **s_handler**() callback, in both user and kernel space, is required to know the end point of the caller in order to create **c2_net_end_point** structure in a received message, and to validate the remote access of passive bulk buffers. This information cannot be obtained from the socket peer address, because a client's socket is not using the same port number as the transfer machine server in the client. The information has to be present in the message.

```
/* end point address */
DEF(sunrpc_ep, RECORD,
    _(sep_addr, U32),
    _(sep_port, U32),  /* really a short */
```

```
    _(sep_id,   U32));

/* appropriate network buffer descriptor */
DEF(sunrpc_buf_desc, RECORD,
    _(sbd_id,         U64),
    _(sbd_active_ep,  sunrpc_ep),
    _(sbd_passive_ep, sunrpc_ep),
    _(sbd_qtype,      U32),
    _(sbd_total,      U32));

/* buffer representation */
DEF(sunrpc_buffer, SEQUENCE,
    _(sb_len, U32),
    _(sb_buf, BYTE));

/* the message send call and response */
DEF(sunrpc_msg, RECORD,
    _(sm_sender,   sunrpc_ep),
    _(sm_receiver, sunrpc_ep),
    _(sm_buf,      sunrpc_buffer));
DEF(sunrpc_msg_resp, RECORD,
    _(smr_rc, U32));

/* the get call and response */
DEF(sunrpc_get, RECORD,
    _(sg_desc,    sunrpc_buf_desc),
    _(sg_offset,  U32));
DEF(sunrpc_get_resp, RECORD,
    _(sgr_rc,  U32),
    _(sgr_eof, U32),
    _(sgr_buf, sunrpc_buffer));

/* the put call and response */
DEF(sunrpc_put, RECORD,
    _(sp_desc,   sunrpc_buf_desc),
    _(sp_offset, U32),
    _(sp_buf,    sunrpc_buffer));
DEF(sunrpc_put_resp, RECORD,
    _(spr_rc,    U32));
```

The offset field in the **get** and **put** procedures is to support scatter-gather vectors by making multiple invocations for each buffer. There is no need to support scatter-gather for the message (as per [3] above).

These data types and procedures are private to this layer and there will be no clash with higher level FOP usage.

Extensions to legacy sunrpc transport

The legacy user space sunrpc client and server already provide the necessary functionality. There are a few things to look out for:

- **s_handler** return processing - there are potential errors in case no results are set

There is no need to use asynchronous calls - the design uses background threads to perform the work, so they can made synchronous calls.

The kernel space sunrpc server does not exist and has to be written. A single background thread can be used for this purpose.

Threading and serialization model

The in-memory and sunrpc implementation serialization models are essentially the same. One exception, which will be described later, is how the in-memory model handles cross-domain communication. Another relates to the mechanism of deferred creation of sunrpc network connections and use of the shared global server.

The transport will maintain a pool of worker threads, per transfer machine, to handle work items asynchronously. In the sunrpc case, the “network” interaction is through threads dispatched by the legacy sunrpc server support. In the in-memory case, the “network” interaction is performed on worker threads of a different transfer machine of a different domain. This is a little tricky, and will be described later after the general discussion of the transfer machine serialization.

The transport will use the transfer machine mutex for buffer list serialization and for internal serialization. Specific condition variables are defined as needed. The worker threads will obtain the transfer machine mutex and then block on the **xtm_work_list_cv** defined in the domain, waiting for work items to appear on the **xtm_work_list**. They will then increment the **xtm_callback_counter**. They will release the mutex while processing the work item. After processing of a work item completes, the **xtm_callback_counter** will be decremented. The **xtm_callback_counter** is required so that the transport state change logic can detect that a work item which has been removed from the **xtm_work_list** is still being processed, and wait for that item to complete during the state change to **C2_NET_XTM_STOPPED**.

The buffer queues are protected in the messaging layer by the transfer machine mutex. Buffers are used without holding the mutex - instead the appropriate flag is set to protect the buffer out of the mutex. They are marked with a **C2_NET_BUF_QUEUED** flag when they are on the queue. A worker thread should set the **C2_NET_BUF_IN_USE** flag only when it actually starts using a buffer and should normally not clear the flag once it dequeues the work item - it will be cleared by **c2_net_tm_event_post**. The cancellation flag, **C2_NET_BUF_CANCELLED** is set in a buffer by a the **xo_buf_del**() subroutine only if the **C2_NET_BUF_IN_USE** flag is not set. If the buffer is in use, then the subroutine sets the **xb_cancel_requested** boolean to true and will not attempt to schedule the cancellation callback on the buffer. The active worker function that is operating on the buffer *may* notice that this value has been set and abort the operation on the buffer, marking its status as cancelled by setting the **C2_NET_BUF_CANCELLED** flag, or it may complete the operation - either way it has to schedule the buffer completion callback.

The interpretation of the **C2_NET_BUF_IN_USE** flag is defined by the queue type. In general, the flag should be set as late as possible, to increase the cancellation window and reduce the complexity. In the case of message receive buffers, this should be done in the **s_handler**() before copying a message to the buffer; in the case of message send buffers only just before being processed for transmission. In the case of passive buffers, the flag need not be set until data transfer to the buffer completes - if a request comes for the buffer and it is not present, then the request fails. In the case of active buffers, set the flag only when the operation commences.

Now back to in-memory cross-domain communication. As mentioned earlier, it is a little tricky as the "network" thread initiating the action (message send and active bulk operations) has to do the following:

1. Safely locate the appropriate transfer machine in another network domain
2. Safely perform an operation on the transfer machine of the other domain (viz. copy the data)

The transport maintains a global list of in-memory domains, **c2_net_bulk_mem_domains**, protected by the **c2_net_mutex** provided by the messaging layer. The mutex is used by the messaging layer to serialize calls to **c2_net_dom_init**() and **c2_net_dom_fini**(). Thus the in-memory transport **xo_dom_init**() and **xo_dom_fini**() methods can directly access the list without explicit serialization.

The "network" thread operates without holding any lock on the originating transfer machine and domain; instead, the operation is protected by the **C2_NET_BUF_IN_USE** flag of the concerned buffer. It has to explicitly obtain the **c2_net_mutex** to safely walk the **c2_net_bulk_mem_domains** list to find the appropriate transfer machine in the destination domain. For each domain on the list, it has to obtain the domain's mutex (**nd_mutex**) to scan the list of transfer machines for the domain looking for the matching end point. This is safe because transfer machine fini and end point creation are both protected by the domain mutex. If a domain does not have a matching transfer machine, then its domain lock is released and the next domain searched. Once a matching transfer machine is found, then the "network" thread has to obtain the transfer machine mutex (**ntm_mutex**).

At this point it releases the **nd_mutex** and **c2_net_mutex**, in that order. Now, while holding **ntm_mutex** it can check the transfer machine state, and if in the **C2_NET_TM_STARTED** state can proceed with its operation. Since it is holding the other transfer machine's mutex, it can safely add a work item to that transfer machine's **xtm_work_list**. It releases the destination transfer machine's mutex when done.

State changes are accomplished while holding the transfer machine mutex. Care should be taken to examine the current state before executing a state change, to avoid making an illegal state transition. For example, the work item requesting a change to state to started should check the current state.

The sunrpc based bulk emulation transport supports the deferred creation of network connections from an end point. This is because the messaging layer's **c2_net_end_point** role includes both the **c2_service_id** as well as the **c2_net_conn** role in the underlying sunrpc transport. The end point object is provided to the **c2_net_tm_start** subroutine - as each transfer machine is implemented as a **c2_service**, one certainly cannot create the connection to the service before it exists. End point creation is protected by the domain mutex, both for the initialization of the **c2_service_id**, as well as during the (deferred) network connection creation. In the sunrpc based bulk emulation transport, end points are used by the worker threads at run time, typically not serialized in any manner, so obtaining the domain mutex in this fashion is

permitted.

The sunrpc based bulk emulation transport uses a shared server for all active transfer machines. The shared server related resources are protected by the module-private read-write lock, **sunrpc_server_lock**. The operations involved are the start and stop of the service, managing the list of transfer machines (**sunrpc_server_tms**), and the searching for a transfer machine from the service handler.

In normal operation, service handler functions determine the target transfer machine from their arguments, then obtain shared access (read) to the lock while searching for this transfer machine. This lock is released only ***after*** obtaining the mutex on the target transfer machine. At this time, the transfer machine's state should be examined and the incoming RPC processing performed only if the state is **C2_NET_XTM_STARTED**.

Transfer machines are added to the list during intialization (**xo_tm_init**) and removed from the list during termination (**xo_tm_fini**). The lock should be obtained in exclusive mode for these operations.

When starting a transfer machine (**xo_tm_start**), the end point has to be checked for uniqueness within the process, which implies serializing the concurrent invocation of the **xo_tm_start** subroutine across different sunrpc based bulk emulation network domains in the same process. The other transfer machines to check against were put on the **sunrpc_server_tms** list from an earlier call to **xo_tm_init**, protected by exclusive access to the **sunrpc_server_lock**. Of concern here is that the **xo_tm_start** subroutine is invoked holding the mutex of the transfer machine in question, and if it were to use the service handler search algorithm above (which obtains a transfer machine lock while holding the shared **sunrpc_server_lock**) it could result in a deadlock. To work around this problem without creating a circular locking dependency with either concurrent invocations of itself or the service handler search algorithm, the **xo_tm_start** subroutine will not attempt to obtain any other transfer machine mutex. Instead, it will first serialize its own invocation using the private **sunrpc_tm_start_mutex** mutex, and while holding this mutex, will obtain a shared access to the **sunrpc_server_lock** to ensure that the list does not change while being traversed (only **xo_tm_init** and **xo_tm_fini** modify the number of elements on the list, and hold exclusive access to **sunrpc_server_lock** while doing so, though note that the transfer machine list item's state can change while on the list). When traversing the list of transfer machines, it will only look at those which have end points defined; since an end point gets assigned and pinned in a transfer machine by the **c2_net_tm_start** call prior to the invocation of **xo_tm_start**, and the end point does not get cleared until after **xo_tm_fini** is called, this will suffice to ensure proper serialization without trying to obtain the mutex of the other transfer machines.

The common server is started when the first transfer machine is started, and stopped when the last transfer machine is stopped. The counter, **sunrpc_server_active_tms** is used to track this. The worker that sets the value to 1 during a start state change, will start the server, and the worker that sets the value to 0 during a stop state change will stop the server. Access to this counter and the action of starting and stopping the service should be serialized with the

sunrpc_server_mutex.

Each sunrpc bulk emulation domain maintains a skulker thread that periodically walks the list of end points anchored in the domain, while holding the domain mutex. It looks only at end points on the list whose reference count is 0, and whose **xep_last_use** value is older than the current time by some configurable margin. It directly calls the internal release subroutine for such end points. This action is safe as end point creation is synchronized with the domain mutex. The **xep_last_use** value was set atomically without holding the domain mutex.

The skulker thread is created in **xo_dom_init**() and terminated from **xo_dom_fini**(), and hence protected by **c2_net_mutex**. The domain mutex is not fini'd until the **xo_dom_fini**() subroutine returns, so it is safe for the skulker thread to hold this mutex. The **xo_dom_fini**() subroutine obtains the domain mutex mutex to signal the skulker thread to terminate.

> One change that had to be made in **c2_net_domain_fini**() was to move the assertion of all the end point list being empty to after **xo_dom_fini**() returns. This allows the cache to be flushed during termination. The empty list assertion after the call is still valid, because the application should have released all end points prior to trying to fini the domain.

The skulker thread also times out buffer operations. Periodically, while holding the domain mutex, it walks the list of transfer machines in turn, obtains its mutex and then walks each of its queues looking at buffers that have their **nb_timeout** value set, and which are not in use (as indicated by the **C2_NET_BUF_IN_USE** flag). If the timeout period is less than or equal to the current time, then the buffer is marked with the flag **C2_NET_BUF_TIMED_OUT**, and an error callback invoked. Each transfer machine mutex is released before advancing to the next transfer machine in the list. The **C2_NET_BUF_TIMED_OUT** flag was newly introduced and is handled in work functions in a manner similar to the **C2_NET_BUF_CANCELLED** flag.

> Three subtle semantic changes were made to the messaging layer to support this first implementation of a buffer timeout operation. The **c2_net_buffer_register**() subroutine always sets the **nb_timeout** value to **C2_TIME_NEVER** to ensure that buffers are initialized correctly - the API had always asked for that, but now the **c2_net_buffer_add**() subroutine was extended to enforce the fact that if **nb_timeout** is set, it is set with a value in the future. The third change made was to **c2_net_buffer_event_post**(); it always resets the **nb_timeout** value back to its initial value of **C2_TIME_NEVER**.

Worker thread processing for in-memory

Worker threads sleep until there are work items available, and process the first item at the head of the **xtm_work_list**. The worker threads will terminate if they see the internal state of the transfer machine is **C2_NET_XTM_STOPPED**.

The following work operations are defined via the value of the **xwi_op** field of the **c2_net_bulk_mem_work_item** structure:

C2_NET_XOP_STATE_CHANGE

This entails changing the state of the transfer machine. The desired state is in the **xwi_new_state** field.

Desired state	Worker action
C2_NET_XTM_STARTED	The internal state must be **C2_NET_XTM_STARTING**. If the **xwi_state_change_status** is not zero, it will fail the TM and set the status in the event to the specified status value - this is provided for derived transports. Otherwise, it will set the internal state to **C2_NET_XTM_STARTED**. Post a state change event changing the transfer machine state to **C2_NET_TM_STARTED**.
C2_NET_XTM_STOPPED	The internal state must be **C2_NET_XTM_STOPPING**. Set the internal state to **C2_NET_XTM_STOPPED** and broadcast on the condition variable to awaken and terminate other threads. Wait for the work item queue to empty. Post a state change event changing the transfer machine state to **C2_NET_TM_STOPPED**.
*****	No other value is valid

The work item structure will be freed.

C2_NET_XOP_CANCEL_CB

This involves posting a cancellation event for the concerned buffer.

C2_NET_XOP_MSG_RECV_CB

This entails delivering the message received event. The item is added to the queue by the the send worker function of the sending transfer machine.

C2_NET_XOP_MSG_SEND

This item involves copying the buffer to the appropriate receive buffer (if available) of the target transfer machine in the destination domain, adding a **C2_NET_XOP_MSG_RECV_CB** item on that queue, and then invoking the completion callback on the send buffer. The sender's end point reference will be incremented for the additional reference from the receiving buffer.

C2_NET_XOP_PASSIVE_BULK_CB

This item entails posting the passive buffer release event. The item is added to the queue by the active buffer worker function.

C2_NET_XOP_ACTIVE_BULK

This item involves performing the desired bulk operation using synchronous GET or PUT RPC calls, followed by the active buffer release event. It is done by copying data from the appropriate passive buffers. Once the copy is complete, a work item is added to the queue to mark completion of the passive buffer, and then the completion callback is invoked on the active buffer.

C2_NET_XOP_ERROR_CB

This entails the delivery of an error. The work item will be freed.

The worker threads will terminate if they see the internal state of the transfer machine is **C2_NET_XTM_STOPPED**. Work item structures associated with buffers are not freed, because they are embedded as part of the buffer private data.

Interface subroutines for in-memory

xo_dom_init	This routine will allocate and initialize the private domain data and attach it to the domain. It will assume that the domain's private pointer is allocated if not NULL. This allows for a derived transport to pre-allocate this structure before invoking the base method. The method will initialize the size and count fields as per the requirements of the in-memory module. If the private domain pointer was not allocated, the routine will assume that the domain is not derived, and will then link the domain in a private list to facilitate in-memory data transfers between transfer machines.
xo_dom_fini	This routine will search for the domain on the list of in-memory domains, and if found will unlink the domain from the list of in-memory domains. Then it will release the private domain data.
xo_end_point_create	The routine will accept a single argument invocation of "hostIP:port" or a two argument invocation of "host" (string) and port number (numeric). The routine will search for an existing end point in the domain, and if not found, will allocate and zero out space for a new end point using the **xd_sizeof_ep** field to determine the size. It will fill in the **xep_sa** field with the IP and port number, and will link the end point to the domain link list. It will write the printable form in the **xep_addr** array. Both the user and kernel space parameters are in the same order.
xo_tm_init	The routine will allocate the transfer machine private data structure and set the state to **C2_NET_XTM_INITIALIZED**. It will use the **xd_sizeof_tm_pvt** field to allocate and zero space.
xo_tm_fini	The routine will release the transfer machine private data.
xo_tm_start	The routine will set the state to **C2_NET_XTM_STARTING**, then create the worker thread pool, and then add a **C2_NET_XOP_STATE_CHANGE** work item to change the transfer machine state to **C2_NET_XTM_STARTED**.
xo_tm_stop	The routine will set the state to **C2_NET_XTM_STOPPING**. It will attempt to set the **C2_NET_BUF_CANCELLED** flag in all queued buffers that do not have the **C2_NET_BUF_IN_USE** flag set. It will change the opcode in the buffer private area to **C2_NET_XOP_CANCEL_CB** for each such cancelled buffer so that

	its cancellation callback is delivered. Receive message buffer and passive buffer work items must be added to the work list. Lastly, it will add a **C2_NET_XOP_STATE_CHANGE** work item to change the state of the transfer machine to **C2_NET_XTM_STOPPED**.
xo_buf_register	The routine will allocate the private buffer data for the buffer, and initialize it. It will also validate the size of the buffer. It will use the **xd_sizeof_buf_pvt** to allocate and zero space.
xo_buf_deregister	The routine will free the private buffer data.
xo_buf_add	Initiates buffer operation. Will clear any previous cancellation state and initialize the status value to -1. Will add the buffer's embedded work item to the work list for messages to be sent, and for active buffers. The routine will create return buffer descriptors for passive buffers.
xo_buf_del	The routine will set the **C2_NET_BUF_CANCELLED** flag in the buffer if the **C2_NET_BUF_IN_USE** flag is not set. If it does so, it changes the work item to **C2_NET_XOP_CANCEL_CB**. Receive message and passive buffer work items must be added to the work list. If it cannot cancel, then it will return an error.
xo_get_*	The routines will return the statically defined buffer size limits.

Worker thread processing for sunrpc

Worker threads sleep until there are work items available, and process the first item at the head of the **xtm_work_list**. The worker threads will terminate if they see the internal state of the transfer machine is **C2_NET_XTM_STOPPED**.

The following work operations are defined via the value of the **xwi_op** field of the **c2_net_bulk_mem_work_item** structure:

C2_NET_XOP_STATE_CHANGE

This entails changing the state of the transfer machine. The desired state is in the **xwi_new_state** field.

Desired state	Worker action
C2_NET_XTM_STARTED	The internal state must be **C2_NET_XTM_STARTING**. While holding the **sunrpc_server_mutex**, increment the **sunrpc_server_active_tms** counter, and initialize the **sunrpc_server_domain** and start the **sunrpc_server_service** if the count gets set to 1. Otherwise, validate that the address of the new TM matches that of the previously started service. It will then call the in-memory work function to complete the

	operation. It will set the xwi_state_change_status to non-zero if the service failed to start.
C2_NET_XTM_STOPPED	The internal state must be **C2_NET_XTM_STOPPING**. Set the internal state to **C2_NET_XTM_STOPPED** and broadcast on the condition variable to awaken and terminate other threads. Wait for the work item queue to empty. While holding the **sunrpc_server_mutex**, decrement the **sunrpc_server_active_tms** counter and stop the **sunrpc_server_service** and fini the sunrpc_server_domain if the count goes to zero. Post a state change event changing the transfer machine state to **C2_NET_TM_STOPPED**.
*****	No other value is valid

The work item structure will be freed.

C2_NET_XOP_CANCEL_CB
No change from the base worker function.

C2_NET_XOP_MSG_RECV_CB
No change from the base worker function. The item is added to the queue by **s_handler**. The sender's end point must be present.

C2_NET_XOP_MSG_SEND
This item involves making a synchronous MSG RPC call to send the message. The use of a synchronous call means that the kernel implementation need not support asynchronous calls. After the call, the message sent event must be posted.

C2_NET_XOP_PASSIVE_BULK_CB
No change from the base worker function. The item is added to the queue by **s_handler** when an active get or set RPC completes.

C2_NET_XOP_ACTIVE_BULK
This item involves performing the desired bulk operation using synchronous GET or PUT RPC calls, followed by the active buffer release event.

C2_NET_XOP_ERROR_CB
No change from the base worker function.

The worker threads will terminate if they see the internal state of the transfer machine is **C2_NET_XTM_STOPPED**. Work item structures associated with buffers are not freed, because they are embedded as part of the buffer private data.

Interface subroutines for sunrpc

In general these directly or indirectly invoke the base in-memory methods, but there are some additions or changes.

xo_dom_init	This routine will allocate the private domain data and attach it to the domain. It will initialize the in-memory transport domain by directly calling its **xo_dom_init**() method, but provide it the sunrpc transport pointer. This sets up this transport as a derived transport from the in-memory transport. On return, it will set the **xd_sizeof_ep**, **xd_sizeof_tm_pvt**, **xd_sizeof_buf_pvt** fields of the in-memory transport private data to the sizes needed by this transport, and set the **xd_num_threads** value to the desired number of threads, and override the base transport subroutines that it needs to customize. Lastly, it will initialize the underlying RPC user or kernel space transport with **c2_net_xprt_init**() (a known no-op) and initialize **xd_rpc_dom**. The call to initialize the underlying RPC user or kernel transport domain will utilize the **c2_net__domain_init**() call that assumes protection within the **c2_net_mutex**.
xo_dom_fini	This routine will fini the **xtm_rpc_dom** with **c2_net__domain_fini**(). It will then clean up and release the private domain data with the base method.
xo_end_point_create	This routine will use the base method to search for an existing end point in the domain, and if not found, will allocate space for a new end point. Upon return from the base method, if the reference count is 1 and the service id not initialized, it will replace the release method with its own release method, and will use **c2_net_service_id_init** to initialize the end point. The initial zero'ing of the data structure marks the sid and connection as not valid.
xo_tm_init	The routine will obtain exclusive access to the **sunrpc_server_lock**, then invoke the base method, and upon successful completion link the transfer machine to the **sunrpc_server_tms** list.
xo_tm_fini	The routine will obtain exclusive access to the **sunrpc_server_lock**, unlink the transfer machine from the **sunrpc_server_tms** list, then invoke the base method.
xo_tm_start	The routine will obtain exclusive access to the **sunrpc_server_lock**, and then check that the end point of the TM is unique. It will release the lock, and then invoke the base method to start the TM.
xo_tm_stop	The routine will invoke the base method directly.
xo_buf_register	The routine will invoke the base method directly
xo_buf_deregister	The routine will invoke the base method directly.

xo_buf_add	The routine will validate the buffer size. Then it will check that if the buffer supplies an end point, the end point is valid. If not valid it will call **c2_net_conn_create**() to create the end point and fail if it cannot. Return buffer descriptors must be created for passive buffers. The buffer will be added to a work list if required.
xo_buf_del	The routine will invoke the base method directly.
xo_get_*	The routines will invoke the base method directly.

Note: In the implementation we chose to change the base in-memory transport to use function pointers in the domain private structure for internal subroutines where the semantics differ between sunrpc and the in-memory transports. The sunrpc implementation replaces these function pointers with its own subroutine pointers, sometimes implementing the new subroutine as an extension of the original base subroutine. This mechanism enabled considerable direct reuse of the **xo**_ subroutines.

Kernel Porting Plan

Both the in-mem and sunrpc transports will be ported to the kernel. This section delineates the changes and additions to the user-space implementation required to make the transports work in a kernel module.

XDR considerations

The C2 wrappers around kernel Sunrpc only expose interfaces for performing RPC itself. To support encoding and decoding the **c2_net_buf_desc** requires exposing additional interfaces. The bulk **sunrpc_desc_create()** and **sunrpc_desc_decode()** must be ported to support kernel XDR interfaces.

Kernel portlib limitations

The kernel portlib provides dummy implementations of **c2_mutex_is_locked** and **c2_mutex_is_not_locked** that always return true. The bulk transport code must be audited for uses of these and ensure it uses them without negation.

Sunrpc kernel server

- may need a ksunrpc_server_init and ksunrpc_server_fini like usunrpc library. usunrpc library uses this to set up pthread key for thread-specific data.
- extend ksunrpc_xprt_ops with xo_service_init and possibly xo_net_bulk_size
- analogs for all of usunrpc/client.c needed in ksunrpc
- linux kernel fs/lockd/svc.c is a simple example that can be followed to initialize and start actual sunrpc service. fs/nfsd/nfssvc.c is more complex, probably not required as an example for ksunrpc since that is already deprecated.
- lockd/svc.c uses BKL (lock_kernel). Need to try to see if we can avoid that. nfsd does not use lock_kernel. Seems a “service mutex” can be used instead. This mutex is used around various sunrpc svc calls and any time the “service invariant” (my term) cannot be guaranteed.
- consider adding sysfs mechanism for getting ksunrpc info and setting behavior

Sunrpc kernel ut

Investigate use of the existing net/ut/client.c test for the kernel server.
There is some UT code in ksunrpc/client.c. This should be refactored out so it can be built into UT kernel module.

Kernel net library

The kernel net library is combined with the kernel sunrpc wrappers in the **net/ksunrpc** directory.
- linux_kernel
- module init which initializes all state of the messaging and bulk transports
- net/ksunrpc/client.c contains existing module init/fini et al, must be incorporated into new linux_kernel/main.c
- bulk sunrpc xprt has c2_sunrpc_fop_init/fini which must be incorporated into new linux_kernel/main.c
- bulk mem xprt has c2_mem_xprt_init/fini which must be incorporated into new linux_kernel/main.c

---

Take 1 (obsolete)

My initial attempt at this task assumed that RPC like semantics were to be maintained.

Pseudo-code for ULPs

Thought exercise on how this is to be used by upper layer protocols. It does not illustrate asynchronous send of an RPC at the transport layer - that is discussed in the next section.

Pre-requisities

As a pre-requisite, I'm going to postulate the existence of different FOPs for read and write with memory descriptors. This is to capture the fact that the upper protocol layers are not sending buffers of data, but instead explicitly obtain and stuff memory descriptors in the FOPs. I'm also allowing for the fact that there such FOPs could be used both synchronously and asynchronously.

But before I define the FOP, I need to define how a memory descriptor will look like, both the over the wire form that will be used in a FOP, and the in memory form. To do so, I'm going to assume the following:

1. A transport buffer descriptor is transport specific, but must also include information on the direction of transfer.
    e.g. for an LNET Put operation: target process id(64+32), portal index(32). match bits(64); header data (64) is optional. A Get operation uses the same values, other than the header.
2. The transport supports scatter-gather memory operations with a single memory

Commented [16]: fop types, technically speaking — nikita_danilov

Commented [17]: Perhaps a better name would be "transfer descriptor" or something? Process identifiers and portals do not logically belong to a memory descriptor. Is this modelled after portals' matching entry?—nikita_danilov

You're right in that a memory descriptor is a portal data structure. How about "buffer descriptor"?
I've now used c2_net_buf_desc for the descriptor, and c2_net_buffer for the in-memory structure.
- carl

Commented [18]: "match bits" are composed of "match bits proper" (64 bits) and "ignore bits" (another 64 bits) —nikita_danilov

True. But the ignore bits are part of the ME, and are not provided to Get/Put. However, defining the data structure as an opaque array will allow us to accommodate this when we use an LNET transport.
- Carl

descriptor.

LNET supports this; may not be true in general.

In the example below, I define an over-the-wire data structure called **c2_net_buf_desc** to contain the transport memory descriptor data, and an in-memory data structure called **c2_net_buffer** to manage memory descriptors.

Assume the following in "core/net/net.h":

```
/* A network buffer descriptor that can be sent to another process.
 * NOTE: FORMAT UNDER REVIEW - WILL MOVE TO A .ff FILE AS A U64 SEQUENCE
 */
struct c2_net_buf_desc {
      uint32_t  nbd_len;        /* length of the values array */
      uint64_t  *nbd_data;  /* transport specific values */
};

/* network buffer access modes */
enum c2_net_buffer_mode {
  C2_NET_NB_REMOTE_READ = 0,
  C2_NET_NB_REMOTE_WRITE = 1
};

/* An in-memory description of a network buffer.
 * Used by c2_net_nb_stage(), c2_net_nb_transfer(), c2_net_nb_fini() and also passed to
 * c2_net_event_wait().
 */
struct c2_net_buffer {
      /* these fields may have to be set before use */
      struct c2_net_domain   *nb_domain;        /* the network domain */
      struct c2_net_conn     *nb_conn;          /* connection to initiator */
      struct c2_diovec        nb_iov;           /* the local buffer */
      enum c2_net_buffer_mode nb_access_mode;   /* the mode of access */
      void                   *nb_app_private;   /* private to higher layers */
      struct c2_net_buf_desc *nb_desc;          /* the OTW buffer descriptor. */
      bool                    nb_free_iov;      /* true if fini should free nb_iov */
      bool                    nb_free_desc;     /* true if fini should free nb_dec */
      /* the rest of the fields should be treated as read-only by upper layers */
      bool                    nb_got_event;     /* true if event delivered */
      int                     nb_event_status;  /* status of delivered event */
      bool                    nb_is_initiator;  /* true if process is RDMA initiator */
      uint64_t                nb_event_id;      /* event id assigned to this descriptor */
      void                   *nb_xprt_private;  /* the tranport specific data */
      struct c2_list_link     nb_linkage;       /* link in domain staged or transfer lists */
};
```

~~*Following under review and may be changed:*~~

~~An XDR routine for c2_net_buf_desc will be defined as a FOP primitive so it can be included in FOP definitions. This involves:~~

- ~~The **c2_xdr_net_buf_desc()** function will be defined in core/net/net_xdr.c, and declared in core/net/xdr.h.~~
- ~~**FPF_NBD** in **enum c2_fop_field_primitive_type** in "core/fop.h".~~
- ~~Memory layout and field type definition in core/fop.c. The XDR will be declared by including "core/net/xdr.h". The init/fini functions will be extended to initialize the new primtive.~~
- ~~Format definitions in fop/fop_format.[ch]~~

In the FOPs below, I use 32 bit numbers for data lengths, because the existing FOPs define it

Commented [19]: struct c2_net_buffer contains much more state than a simple buffer and looks more like a 0-copy transfer state machine. Perhaps it shold be renamed? —nikita_danilov

Commented [20]: Perhaps this should have been uint64_t mdesc_values[0]? It's not clear what a pointer means in an on-wire structure. —nikita_danilov

Deferred until I can resolve circular dependencies in use of fop2c. This is what it looks like if it is defined as a SEQUENCE of U64.
- Carl

Commented [21]: Returned in a stage() and should be freed by fini. Input in a transport() and should not be freed by fini. —carl_braganza

Commented [22]: I don't think memory descriptor should be a new c2_fop_field_primitive_type along with VOID, BYTE, U32 and U64. It can easily be defined via a usual .ff RECORD definition. —nikita_danilov

I had started that way, but then the reason I asked for this was to refer to struct mem_desc in other data structures (like in struct c2_net_buffer). I suppose I could just declare its existence and put a pointer and that would work, but, that would also mean that the transport logic is dependent on a .ff file with the appropriate definition. Do you see an issue here?
- carl

I think it's fine to depend on some transport specific .ff file. What _is_ a potential problem though is that a format of descriptor might depend on a transport type. E.g., LNet would have a different descriptor than sunrpc. This means that descriptor should be self-describing, and its .ff definition, will, most likely, be

```
DEF(c2_mem_desc, SEQUENCE,
      _(md_nob, U32),
      _(md_opaque, BYTE));
```

or something similar.

Exactly. I essentially wanted to use a ***transport independent*** *variable length array of uint64* - (identical to c2_vec, for that matter). I see a need for it to be passed around in transport independent code. We don't have "standard" .ff files which would generate C data structures for our standard types that any other .ff could #include. For example, I can't use c2_vec in a .ff file though it should be perfectly valid to send it OTW! I wonder whether we should support such a thing? I'll defer changing this until we can discuss further.
- Carl

Commented [23]: Subhash, please research the feasibility of this part.
—carl_braganza

so.

So, to illustrate with pseudo-code, lets pretend the following will be added to stob/ut/io.ff:

```
/* Read with buffer descriptors.
 * sirb_object - the FID
 * sirb_offset - the offset in the file
 * sirb_count - the number of byte to read
 * sirb_bd    - the BD of the buffer in which to store the file data. Writable by target.
 */
DEF(c2_io_read_bd, RECORD,
        _(sirb_object, c2_fop_fid),
        _(sirb_offset, U64),
        _(sirb_count, U32),
        _(sirb_bd, c2_net_buf_desc));

/* Read with BDs response.
 * Sent on failure or after data has been transferred.
 * The memory descriptors can be released on receipt.
 * sirbr_rc - the return code. 0 is success.
 * sirbr_count - the number of bytes read
 */
DEF(c2_io_read_nb_rep, RECORD,
        _(sirbr_rc, U32),
        _(sirbr_count, U32));

/* Write with Buffer Descriptors
 * siwb_object - the FID
 * siwb_offset - the offset into the file
 * siwb_count - the number of bytes to write
 * siwb_bd    - the BD of the data to write. Readable by target.
 */
DEF(c2_io_write_nb, RECORD,
        _(siwb_object, c2_fop_fid),
        _(siwb_offset, U64),
        _(siwb_count, U32),
        _(siwb_nb, c2_net_buf_desc));

/* Write with MDs response.
 * Sent on failure or after data transfer completes.
 * The memory descriptors can be released on receipt.
 * siwbr_rc - the return code. 0 is success.
 * siwbr_count - the number of bytes written.
 */
DEF(c2_io_write_nb_rep, RECORD,
        _(siwbr_rc, U32),
        _(siwbr_count, U32));
```

Client side RDMA processing

Pseudo-code showing a client reading data from a server, described by a **c2_net_conn** (nid). The example ignores the possibility of a batching layer. The code is losely based on the ksunrpc_read_write subroutine in c2t1fs/main.c. The remote invocation is made with the existing **c2_net_cli_send**() interface. The pseudo-code below illustrates a thread blocking implementation strategy - more asynchronous approaches are also possible.

```
int
read_nb_async(c2_net_conn *nid, c2_fop_fid fid, struct c2_diovec *iov, loff_t pos)
{
    struct c2_net_buffer nb;
    struct c2_fop *f;
    struct c2_fop *r;
    struct c2_io_read_nb *arg;
    struct c2_io_read_nb_rep *ret;
    struct c2_net_call call;
    int rc = 0;

    /* create network buffers for memory that the client exposes to the server (pins it too) */
    nb.nb_conn = nid;
    nb.nb_iov = iov;
    nb.nb_free_iov = false;
    nb.nb_access_mode = C2_NET_NB_REMOTE_WRITE;
    c2_net_nb_stage(nid, &nb);

    /* allocate the fop request/response structures */
    f = c2_fop_alloc(&c2_io_read_nb_fopt, NULL);
    r = c2_fop_alloc(&c2_io_read_nb_rep_fopt, NULL);
    call.ac_arg = f;
    call.ac_ret = r;

    /* fill in the read request */
    arg = c2_fop_data(f);
    arg->sirb_object = fid;
    arg->sirb_offset = pos;
    arg->sirb_count = (unsigned int)c2_vec_count(&iov->div_vec.ov_vec);
    arg->sirb_nb = nb.nb_desc;

    /* make the call asynchrously */
    c2_net_cli_send(nid, call);

    /* potentially do something else ... */

    /* wait for the response */
    c2_net_cli_wait(call);

    /* get the result */
    rc = ret->sirbr_rc ? : ret->sirbr_count;

    /* wait for the data to arrive */
    if ( rc == 0 ) {
      c2_net_event_wait(nid->nc_domain, C2_NET_EV_REMOTE_OP_COMPLETED, &nb);
    }

    /* free the fop request/response structures */
    ...
    /* release the network buffer */
    c2_net_nb_fini(&nb); /* gets unpinned */

    return rc;
}
```

The **c2_net_nb_stage()** subroutine uses its first argument to call a transport specific method to expose the caller's buffers to the specified remote target initiator that will perform the RDMA required to access the data in the specified buffer. The **nb_access_mode** (**C2_NET_NB_REMOTE_WRITE**) is required to establish the desired access control for the specified remote target and the direction of transfer. It is implicitly assumed here that the returned memory descriptor is not usable with other servers. The subroutine will assign an event identifier in the **nb_event_id** field, then enqueue the **c2_net_buffer** onto the domain's

Commented [24]: What is this supposed to wait for? Is the idea that RDMA completion is always signalled by a separate transport level message, separate from the reply fop (This indeed is the LNet behaviour, but do we want to enforce it?)? The description of reply fop above seems to indicate that it is the reply fop that should be used as a completion indicator. —nikita_danilov

nd_staged_buffers list using the **nb_linkage** field, and then invoke a new transport specific method, **xo_nb_stage**, to perform the operation:

```
struct c2_net_domain {
   ...
   struct c2_list  nd_staged_buffers;
};

struct c2_net_xrpt_ops {
   ...
   int (*xo_nb_stage)(c2_net_conn *, c2_net_buffer *);
};
```

The transport can pick up other things needed to perform this operation, like the process id and the match bits, from the invocation environment or the **c2_net_domain** data structure. The match bits, in particular, should be used or derived from the **nb_event_id** field that the transport should set in the **c2_net_buffer** structure. The transport will also pin the memory pages.

What about the alignment of memory buffers? May not be any issue here - Nikita clarified in email.

An LNET transport would do an LNetMEAttach, using uniquely generated match bits, and then an LNetMDAttach. The returned descriptor would have sufficient data for the remote initator to do an LNetPut operation on the ME (target, portal and match bits).

The maximum buffer size for bulk transfer can be queried with the **c2_net_nb_max_size**() subroutine. This calls an existing (unused) method **xo_net_bulk_size**() in **c2_net_xprt_ops**.

The **c2_net_nb_fini()** subroutine unlinks the buffer from the domain nd_net_buffers list, then subroutine will invoke a new transport specific method to cancel any transport specific operation in progress:

```
struct c2_net_xrpt_ops {
   ...
   void (*xo_nb_fini)(c2_net_buffer *);
   void (*xo_nb_desc_free)(struct c2_net_buf_desc *);
   int (*xo_nb_desc_copy)(struct c2_net_buf_desc *from,  struct c2_net_buf_desc **to);
};
void c2_net_nb_desc_free(struct c2_net_domain *dom, struct c2_net_buf_desc *nbd);
int c2_net_nb_desc_copy(struct c2_net_domain *dom,
                        struct c2_net_buf_desc *from, struct c2_net_buf_desc **to);
```

It frees then data in the **c2_net_buffer** structure as needed. The **xo_nb_desc_free**() operation is provided to implement a **c2_net_nb_desc_free**() subroutine that can optionally be called to free a descriptor. Likewise, the **c2_nb_desc_copy**() operation is provided to support the **c2_net_nb_desc_copy**() subroutine.

The **c2_net_cli_send**() subroutine is already defined is not yet in use. It should be used to make an asynchronous remote call. Control returns potentially before the message is delivered, and the client may do other processing before attempting to collect the response. The subroutine will obtain a unique event identifier using the **c2_net_event_get_next_id**()

Commented [25]: Just to clarify what I mentioned elsewhere, it would be beneficial to separate data-structures and interfaces for (0) registering memory regions with rdma engine, (1) actual rdma transfer state-transitions. Specifically, match-bits are part of the latter, but not the former.—nikita_danilov

Commented [26]: I would imagine, a network transport would have ->xo_0copy_aligment() method that returns required alignment. —nikita_danilov

Commented [27]: Hrm...

```
core/net/net_cli.c:
int c2_net_cli_send(struct c2_net_conn
*conn, struct c2_net_call *call)
{
 ADDB_ADD(conn, net_addb_conn_send);
 return conn->nc_ops->sio_send(conn, call);
}
C2_EXPORTED(c2_net_cli_send);
```

—nikita_danilov

Sorry - i meant not used.
- Carl

subroutine and save it in the **ac_even_id** field. It will chain the **c2_net_call** structure into the **nd_active_call** list maintained in the **c2_net_domain** structure. These are described further in the transport event queue mechanism section. The subroutine will then call the transport specific **sio_send**() method to perform the asynchronous call.

It is critical that the memory descriptors get freed, but only after the data transfer completes. It is entirely possible that the client could receive the response FOP indicating a successful write before the server completes the RDMA to the data buffer.

> Can we force reliance on the response message coming back to know when the data transfer completes? If we could then that would make the client coding a lot simpler. However, this puts more of a burden on the server in that it then has to track the completion of its RDMA write before sending the RPC response message. This potentially could reduce the throughput and performance of the server and increase its complexity. It is better to synchronize on the client side by allowing the client to say that it wants to wait for the MD as well as the response. Accordingly...

The client therefore, first waits for the server's data transfer to complete by calling **c2_net_nb_wait()**, before looking for the response FOP. See transport event queue section below.

Commented [28]: How is this possible? write reply carries a "number of bytes written" field, which implies that by the time the reply was sent (and even more so by the time it was received) the write operation completed, which in turn implies that rdma completed?

That said, the server might send a number of "intermediary replies" for whatever purposes, but there is a "logical completion reply" that won't introduce any otherwise avoidable complexity at the server.

What, on the other hand, a transport level should do (if possible) is to check in ->o_nb_fini() that no rdma is outstanding.—nikita_danilov

The server must send its "logical completion reply" only after the rdma write has completed, which is why I provided the c2_net_nb_wait() call. The transport could make a check like you suggest, but only if there is an association between the call and the rdma buffers involved.
- Carl

In the contrived example above, the **c2_net_cli_wait()** subroutine blocks the invoking thread until the specified call completes or fails. This call is implemented over the transport event queue **c2_net_event_wait** subroutine. It will dequeue the **c2_net_call** structure from the domain's **nd_active_sends** list prior to returning.

Instead of the **c2_net_cli_send**() / **c2_net_cli_wait**() combination, the client could as well use the blocking **c2_net_cli_call**(). However, as the semantics of network buffer usage in the FOP is not known to this subroutine, the client should still explicitly wait for the network buffers after the **c2_net_cli_call**() completes successfully. Note that no call completion events are posted for synchronous calls.

It should be possible for a client to mix **c2_net_cli_send**() and **c2_net_cli_call**() invocations in the same process using different connections. Multiple send **c2_net_cli_send**() calls can be made on the same connection, but a **c2_net_cli_call**() may fail if there is an asynchronous call in flight and vice-versa - it is transport specific.

> ***Any issues/constraints about connection usage vis-a-vis threading?***

A client write using RDMA would be almost identical to the read code. The pseudo-code below provides an illustration of a simplistic client.

```
int
write_nb_async(c2_net_conn *nid, c2_fop_fid fid, const struct c2_diovec *iov, loff_t pos)
{
    struct c2_net_buffer nb;
    struct c2_fop *f;
    struct c2_fop *r;
    struct c2_io_write_nb *arg;
    struct c2_io_write_nb_rep *ret;
    struct c2_net_call call;
```

```
    int rc = 0;

    /* get network buffers for memory that the client exposes to the server (pin it too) */
    nb.nb_conn = nid;
    nb.nb_iov = iov;
    nb.nb_free_iov = false;
    nb.nb_access_mode = C2_NET_NB_REMOTE_READ;
    c2_net_nb_stage(&nb);

    /* allocate the fop request/response structures */
    f = c2_fop_alloc(&c2_io_write_nb_fopt, NULL);
    r = c2_fop_alloc(&c2_io_write_nb_rep_fopt, NULL);
    call.ac_arg = f;
    call.ac_ret = r;

    /* fill in the write request */
    arg = c2_fop_data(f);
    arg->siwb_object = fid;
    arg->siwb_offset = pos;
    arg->siwb_count = c2_vec_count(&iov->div_vec.ov_vec);
    arg->sirw_nb = nb.nb_desc;

    /* make the call asynchrously */
    c2_net_cli_send(nid, call);

    /* potentially do something else ... */

    /* wait for the response */
    c2_net_cli_wait(call);

    /* get the result */
    rc = ret->siwbr_rc ? : ret->siwbr_count;

    /* free the fop request/response structures */
    ...
    /* release the network buffer */
    c2_net_nb_fini(&nb); /* gets unpinned */

    return rc;
}
```

In this case the mode passed to **c2_net_nb_stage**() is **C2_NET_NB_REMOTE_READ**.

> An LNET transport would do an LNetMEAttach, using uniquely generated match bits, and then an LNetMDAttach. The returned descriptor would have sufficient data for the remote initator to do an LNetGet operation on the ME (target, portal and match bits).

Unlike the read case, there is no need for the client to explicitly wait on its RDMA buffers, as the receipt of the write response FOP is sufficient acknowledgment that the server has fetched the data.

Server side RDMA processing

The server handler for a FOP will decode the memory descriptor associated with the client buffer. The length of the buffer is described in the FOP itself, and will be used by the server to internally allocate (properly aligned) memory required for the FOP operation.

The following subroutine is provided to perform the RDMA:

```
int c2_net_nb_transfer(struct c2_net_buffer *);
```

The subroutine fills in a **c2_net_buffer** structure with information necessary to initiate an RDMA between the specified network buffer descriptor (**c2_net_buf_desc**) and the specified local buffer (**c2_diovec**), whose pages will get pinned. The direction of transfer is described in the network buffer descriptor. The return structure is also needed for event processing. It must be cleaned up finally with **c2_net_nb_fini**(), which will also unpin the local buffer pages.

If the operation is a form of "read", then the server must first initiate I/O to fetch the data from the underlying storage into its memory buffer. Then, following whatever cooking of the data necessary, the server arranges for the data to be pushed to client memory by invoking **c2_net_nb_transfer().** For example, pseduo-code for a read handler could look like this:

```
int read_nb_fop_execute(struct c2_fop *fop, struct c2_fop_ctx *ctx)
{
    struct c2_io_read_nb  *in = c2_fop_data(fop);
    struct c2_io_read_nb_rep *out;
    struct c2_diovec iov;
    struct c2_net_buffer nb;

    /* allocate buffer space */
    iov.div_vec.ov_vec.v_nr = 1;
    C2_ALLOC_ARR(iov.div_vec.ov_vec.v_count, 1);
    C2_ALLOC_ARR(iov.div_vec.ov_buf, 1);
    iov.div_vec.ov_buf[0] = c2_alloc_aligned(in->sirb_count, C2_DIOVEC_SHIFT);
    iov.div_vec.ov_vec.v_count[0] = in->sirb_count;

    /* read data into this buffer */

    /* write the data to the client network buffer using RDMA (non-blocking). Pins local buffer. */
    nb.nb_domain = nid->nc_domain;
    nb.nb_iov = iov;
    nb.nb_free_iov = false;
    nb.nb_desc = in->sirb_desc;
    c2_net_nb_transfer(&nb);

    /* construct the response FOP */
    /* post the response */
    c2_net_reply_post(ctx->ft_service, out, ctx->fc_cookie);

    /* wait for completion */
    c2_net_event_wait(ctx->ft_service->s_domain, C2_NET_EV_LOCAL_OP_COMPLETED, &nb);

    /* release the network buffer */
    c2_net_nb_fini(&nb); /* unpins local buffer */

    /* free local buffer */
    c2_free(iov.div_vec.ov_buf[0]);
    free(iov.div_vec.ov_buf);
    free(iov.div_vec.ov_vec.v_count);

    return 0;
}
```

Commented [29]: c2_diovec_alloc(&vec, size); — carl_braganza

Commented [30]: c2_diviovec_free() ? — carl_braganza

Note that the **c2_net_nb_transfer** call requires the **c2_net_domain** data structure pointer. This allows this subroutine to be used out of context of an RPC dispatch handler if desired. The subroutine will assign an event identifier in the **nb_event_id** field, then enqueue the **c2_net_buffer** onto the domain's **nd_transfer_buffers** list, using the **nb_linkage** field. It will

then invoke a new transport specific method, **xo_nb_transfer()**, to perform the operation:

```
struct c2_net_domain {
   ...
   struct c2_list  nd_transfer_buffers;
};

struct c2_net_xrpt_ops {
   ...
   int (*xo_nb_transfer)(struct c2_net_buffer *);
};
```

The transport must use the specified event identifier in the event reporting completion of this RDMA transfer.

If the direction of transfer in the network buffer descriptor is C2_NET_NB_REMOTE_WRITE, an LNET transport would create an MD for the buffer using LNetMDBind, then perform an LNetPut. It sets nb_event_id to match that in the event generated when the Put completes.

After that, the server returns the response FOP using the normal **c2_net_reply_post**() call. **It is entirely possible that the response message gets delivered to the client before the RDMA data is written to the client. It is the responsibility of the client to handle such race conditions.**

The server still has to ensure that the memory buffers do not get reused until the network read completes, so has to wait for the transfer to complete with **c2_net_event_wait**(), before releasing the network buffer resource and freeing the local buffer. The server could potentially use **c2_net_event_timed_wait**() if it did not want to wait forever.

As in the client case, **c2_net_nb_fini**() must be called to dequeue the **c2_net_buffer** and release associated transport specific resources.

If the operation is a form of “write”, then the server must first fetch the remote data from the client using **c2_net_nb_get()** and *wait for completion*. Upon completion of this transfer it can initiate the write to its underlying storage, and then send the response message using **c2_net_reply_post**(). For example, pseduo-code for a write handler could look like this:

```
int write_nb_fop_execute(struct c2_fop *fop, struct c2_fop_ctx *ctx)
{
    struct c2_io_write_nb  *in = c2_fop_data(fop);
    struct c2_io_write_nb_rep *out;
    struct c2_diovec iov;
    struct c2_net_buffer nb;

    /* allocate buffer space */
    c2_diovec_alloc(&iov, in->siwb_count);

    /* get client network buffer using RDMA (non-blocking). Pins local buffer. */
    nb.nb_domain = nid->nc_domain;
    nb.nb_iov = iov;
    nb.nb_free_iov = true;
    nb.nb_desc = in->siwb_desc;
    c2_net_nb_transfer(&nb);

    /* wait for completion */
```

Commented [31]: I don't see what advantages this increase in message concurrency gives us. A server has to wait for the bulk completion anyway (if only to release the memory buffers). It can send the reply afterwards, avoiding the race. —nikita_danilov

Yes. I phrased this badly.
- Carl

```
    c2_net_event_wait(ctx->ft_service->s_domain, C2_NET_EV_LOCAL_OP_COMPLETED, &nb);

    /* write the data */
    /* construct the response FOP */
    /* post the response */
    c2_net_reply_post(ctx->ft_service, out, ctx->fc_cookie);

    /* release the network buffer */
    c2_net_nb_fini(&nb); /* unpins local buffer and frees */

    return 0;
}
```

The **c2_net_nb_transfer** subroutine perform the desired RDMA operation using a transport specific method.

If the direction of transfer in the network buffer descriptor is C2_NET_MD_REMOTE_READ, an LNET transport would create an MD for the buffer using LNetMDBind, then perform an LNetGet. It sets nb_event_id to a value that will be returned in the event generated when the Get completes.

Transport event queue mechanism

The event queue mechanism should support the following requirements:

- Provide functionality at the **c2_net_domain** data structure level, as RDMA support would be needed by higher levels such as the rpc batching layer, out of context of the client/server calls.
- Should support blocking calls to wait for specific events. Optional timeout variants too
- Should support a general event dispatch mechanism - a registered callback handler would suffice. (should co-exist with the specific mechanism)
- Provide notification of buffer transfer completion on the client side to handle local buffer release (a remote server get or put operation completed). Event should have sufficient information to identify the **c2_net_buffer** data structure.
- Provide notification of call completion for asynchronous clients. The event should have sufficient information to identify the **c2_net_call** data structure.
- Provide notification of local server get or put operation completion. The event should have sufficient information to identify the **c2_net_buffer**.

First define the data structure to describe an event of interest at this level:

```
enum c2_net_event_type {
   C2_NET_EV_CALL_COMPLETED,      /* a client call completed (arg: c2_net_call) */
   C2_NET_EV_REMOTE_OP_COMPLETED, /* a remote Get/Put completed (arg: c2_net_buffer) */
   C2_NET_EV_LOCAL_OP_COMPLETED,  /* a local Get/Put completed (arg: c2_net_buffer) */
   C2_NET_EV_TRANSPORT,           /* transport specific event */
   C2_NET_EV_NR
};
struct c2_net_event {
   struct c2_net_domain    *ne_dom; /* the domain */
   enum c2_net_event_type   ne_type; /* the type of event */
   uint64_t                 ne_event_id; /* id of event (from c2_net_call or c2_net_buffer) */
   int                      ne_status; /* 0 on success, error code on failure */
   void                    *ne_arg; /* event specific payload */
};
```

To support a unique event identifier for all events associated with the network domain, we will provide the **c2_net_event_get_next_id**() subroutine to return the next available event identifier. It maintains a counter in the **c2_net_domain** structure:

```
struct c2_net_domain {
    ...
    struct c2_atomic64  nd_event_id_counter;
};

uint64_t c2_net_event_get_next_id(struct c2_net_domain *);
```

It is possible to have events unrelated to **c2_net_call** and **c2_net_buffer** data structures. The modules that post such events should are expected to use this subroutine to get a unique event identifier for the events.

To support events for asynchronous call completion, we add an event id field to **c2_net_call** and linkage fields so they can be tracked via the **c2_net_domain** structure and avoid race conditions between submission and the event signifying completion:

```
struct c2_net_call {
    ...
    uint64_t               ac_event_id;
    struct c2_list_linkage ac_active;
};

struct c2_net_domain {
    ...
    struct c2_list   nd_active_sends;
};
```

The **ac_event_id** field will be set by the **c2_net_cli_send**() subroutine and should be used by the transport when posting the event signifying call completion. No events are delivered for synchronous calls.

The following blocking calls are defined to wait for specific events:

```
int c2_net_event_wait(struct c2_net_domain *dom, struct c2_net_event_type et, ...);
int c2_net_event_timed_wait(struct c2_net_domain *dom, struct c2_net_event_type et,
                            struct c2_time *abs, ...);
```

We define a list for event waiters in the **c2_net_domain** data structure:

```
struct c2_net_event_waiter {
   uint64_t                new_event_id; /* id to wait on */
   struct c2_chan          new_chan; /* for blocking waits */
   struct c2_linkage       new_wait_link; /* wait list linkage */
   int                     new_event_status; /* status from event */
};

struct c2_net_domain {
    ...
    struct c2_list   nd_event_waiters;
};
```

The existing **nd_rwlock** will be used to serialize all lists anchored in the data structure.

Commented [32]: To clarify what is probably well-understood at the moment, the current c2_net_call with its fops does not belong to the transport layer interface, which operates on unformatted buffers. Matching of request and reply is done by the rpc layer by session mechanisms. —nikita_danilov

I get it (now).
- Carl

Commented [33]: It looks that the intent is to have (effectively) a single event queue per c2_net_domain? Could we instead associate an event queue with a "transfer" data structure? This would improve scalability (finer grained locking, plus no need to scan the list of registered waiters) and eliminate the need for global event identification.

In addition, event call-backs can be made type-safe by something like

```
struct c2_net_ev_cb {
        int (*nec_call)(...);
        int (*nec_remote)(...);
        ...
};
```

&c. for each event type.
—nikita_danilov

Okay. - Carl

The only dubious use of “read” instead of “write” is in the **c2_service_start** and **c2_service_stop** subroutines.

Commented [34]: This may be a bug. —carl_braganza

It looks very much like one. Please add a logD record (and a fix :-)).

The arguments to the subroutines depend on the event type, as indicated in the comments in the event type definition. These subroutines are fully implemented at the c2_net layer above the transport. They use a **c2_net_event_waiter** structure linked into the domain’s **nd_event_waiters** list. The invoking subroutine would define the event fields to match in the **c2_net_event_waiter** structure allocated on its stack, add the **c2_link** structure to the channel. Then it would enqueue the **c2_net_event_waiter** structure onto the net domain’s **nd_event_waiters** list with the **new_wait_link** field, and block waiting on the channel until signalled. Channels also provide timed wait functionality. The matched event is not returned to the blocking subroutine, but its status is returned back in the **new_event_status** field.

The event wait calls that wait for specific **c2_net_buffer** related events will return a failure if the data structure is not enqueued on the domain’s **nd_staged_buffers** or **nd_transfer_buffers** lists. They would have been put there by the calls to **c2_net_nb_stage**() or **c2_net_nb_transfer**(). If found on the lists, the structure’s **nb_got_event** value will indicate if the event associated with the buffer has already been delivered, in which case the subroutine will return immediately. In all cases, the **nb_event_status** value will indicate the success or failure of the operation.

Likewise, event wait calls that wait for specific **c2_net_call** related events will fail if the data structure is not enqueued on the domain’s **nd_active_sends** list. It would have been put there by the call to **c2_net_cli_send**().

We provide the following subroutine to post events:

```
void c2_net_event_post(struct c2_net_domain *dom, struct c2_net_event *ev);
```

When **c2_net_domain_init**() is invoked, the net domain data structure pointer is made available to the transport’s **xo_dom_init**() method, which makes it possible for the transport layer to post events.

The **c2_net_event_post** subroutine will wake up blocked waiters for specific events, and will also dispatch events to an application registered event callback upon the caller’s thread. It attempts to find matching **c2_net_buffer** and **c2_net_call** structures on its “active” queues, and if so, sets the appropriate pointers in the event before delivery. The invoker can free the event upon return from the subroutine.

Applications can register an event callback with the **c2_net_event_register_callback()** subroutine.

```
typedef void (*c2_net_event_cb_t)(const struct c2_net_event *, void *);
int c2_net_event_register_callback(struct c2_net_domain *domain, c2_net_event_cb_t cb, void *ctx);
```

The callback should process but not free the event. No reference to the event should be made after the callback returns. The callbacks are stored on a list in the **c2_net_domain** structure:

```
struct c2_net_event_callback {
   struct c2_list_linkage  nec_link;
   c2_net_event_cb_t       nec_cb;
   void                   *nec_ctx;
};
struct c2_net_domain {
    ...
    struct c2_list         nd_event_callbacks;
};
```

Fully asynchronous clients

Truly asynchronous clients would not block after the **c2_net_cli_send**() call but use the underlying transport event callback support to alert them when calls complete, whereupon they can recover the **c2_net_call** pointer from the event, and then use the **c2_net_cli_wait**() call to get the response.

Fully asynchronous servers

Truly asynchronous servers will not wish to block waiting for a Get operation to complete. Instead they would wish to initiate the data transfer, and then relinquish control, and then resume when the transport level queuing mechanism notifies them of the completion of the Get operation.

Is the currently defined c2_net interface sufficient for this purpose? Essentially, the question boils down to how the **c2_service.s_handler**() method could return without completing the RPC operation, and then be re-invoked later to resume operations in response to some stimuli. Unfortunately I think it is only partially supported with the existing definitions, and will need some tweaks to fully support this functionality. Consider:

- The **c2_net_reply_post()** subroutine is the mechanism by which a server's dispatch handler notifies the networking subsystem of completion, and provides the FOP to be returned to the caller. The function is implemented by an indirect call to the transport supplied **c2_service_ops.so_reply_post**() method, so can be customized by the transport. The subroutine is adequate for this purpose, but as it is not documented, its invocation context is ambiguous.

- The **c2_service.s_handler()** method is not documented. The method is invoked by the transport dispatch logic, which, thus, controls its invocation arguments - in particular, it provides an opaque "cookie" to be passed to **c2_net_reply_post**(). It defines the return argument to be an integer, but the meaning of the return value is not defined.

  Current implementation in the usunrpc server code assumes that a return from the handler that is preceded by a call to the **c2_net_reply_post**() subroutine implies RPC completion. There is no provision for re-invocation of the handler were **c2_net_reply_post**() not invoked, but instead, cleanup is immediately performed. The return value of the handler is ignored.

- Another issue, which, strictly speaking is not a c2_net layer problem, is that the

Commented [35]: This is exactly what we are planning to do, see HLD of fom. —nikita_danilov

Commented [36]: Generally, any interface that signals completion via c2_chan is OK. —nikita_danilov

definition of the **c2_fop_type_ops.fto_execute()** operation, which an RPC handler would call to execute a FOP, as currently illustrated by the c2t1fs server. The method is documented as “XXX temporary ...” and does not define how asynchronous behavior is to be implemented. However, it has a redeeming feature in that there is a **c2_fop_ctx** data structure passed, which is already defined to contain the opaque “cookie” to be passed to **c2_net_reply_post**(). Interpretation of this context, anyway, belongs to a server implementation, and not the FOP module.

We need to do the following to provide support for asynchronous execution of an RPC. Essentially, we must dissociate the receipt of the request FOP and the posting of the response FOP from the execution context. This will allow an implementation to perform non-busy waits during the construction of the response FOP:

- The **s_handler** return code will be documented to be ignored.
  The current usunrpc transport ignores this value anyway.

- The **s_handler** will be documented to be responsible only for the delivery of the request FOP that initiates an RPC. Return from the handler has no significance - i.e. it should not automatically free any resources related to its invocation, or cause any message to be sent to the caller. Instead, the handler will provide an opaque “cookie” that must be passed to **c2_net_reply_post** along with a response FOP. This cookie will identify the RPC context and resources that need to be freed.
  The current usunrpc transport frees resources immediately after the handler returns.

- Document that the **c2_net_reply_post** subroutine is responsible for the delivery of the response FOP. It can be invoked out of the context of the **s_handler** thread that provided its arguments. This allows an RPC to be completed from another thread and would require that a transport implementation store sufficient contextual information in the opaque cookie to allow it to compete the RPC handshake.
  In the current usunrpc transport, the cookie is allocated on the stack of the subroutine that invokes the handler.

Fully asynchronous c2t1fs server

To illustrate, this is how parts of the current c2t1fs server could be modified to be fully asynchronous using the new defintions from above:

```
struct call_ctx {
   struct c2_fop_ctx     cc_fop_ctx;
   int                   cc_state;
   struct c2_net_buffer  cc_nb;
   int                (*cc_handler)(struct c2_fop *fop, struct c2_fop_ctx *ctx);
};

static void io_handler(struct c2_service *service, struct c2_fop *fop, void *cookie)
{
    struct call_ctx *cc;
    C2_ALLOC_PTR(cc);
    cc->fop_ctx.ft_service = service;
    cc->fop_ctx.fc_cookie  = cookie;
    cc->cc_handler = fop->f_type->ft_ops->fto_execute;
```

```
    cc->cc_handler(fop, &cc->fop_ctx);
    return;
}

int write_handler(struct c2_fop *fop, struct c2_fop_ctx *fop_ctx)
{
    struct call_ctx *cc;
    cc = container_of(fop_ctx, struct call_ctx, cc_fop_ctx);
    if ( cc->cc_state == 0 ) {
        struct c2_io_write_nb *in = c2_fop_data(fop);
        struct c2_net_buffer *nb = &cc->cc_nb;
        /* save details of the request */
        /* initiate the RDMA */
        C2_ALLOC_PTR(nb);
        cc->fc_app_private = nb;
        c2_diovec_alloc(&nb->nb_iov, in->siwb_count);
        nb->nb_domain = cc->cc_fop_ctx.ft_service->s_domain;
        nb->nb_desc = in->siwb_desc;
        nb->nb_free_iov = true;
        nb->nb_app_private = cc; /* save call context */
        /* initiate RDMA and return to wait */
        c2_net_nb_transfer(nb);
        cc->cc_state = 1; /* next state on successful resume */
        return 1; /* ? */
    }
    if ( cc->cc_state == 1 ) {
        struct c2_net_buffer *nb = &cc->cc_nb;
        struct c2_io_write_rep *ex;
        struct c2_fop *reply;
        reply = c2_fop_alloc(&c2_io_write_rep_fopt, NULL);
        ex = c2_fop_data(reply);
        . . . /* as before, but buffer in nb->nb_iov */
        c2_net_nb_fini(nb); /* release network buffer resources */
        c2_net_reply_post(cc->cc_fop_ctx.ft_service, reply, cc->cc_fop_ctx.fc_cookie);
        c2_free(cc);
        return 1; /* ? */
    }
    /* error cases... */
    return 1; /* ignored */
}
static void event_handler(const struct c2_net_event *e, void *handler_ctx)
{
  struct call_ctx *cc;
  if ( ev->ne_type == C2_NET_EV_LOCAL_OP_COMPLETED ) {
      struct c2_net_buffer *nb = (struct c2_net_buffer *)e->ne_arg;
      if ( nb == NULL ) return;
      cc = nb->nb_app_private; /* recover call context */
      C2_ASSERT( cc->cc_nb == nb );
      if ( ev->ne_status != 0 ) {
          cc->cc_state = -1; /* error */
      }
      /* should really use another thread, but for illustration invoke here */
      cc->cc_handler(NULL, &cc->cc_fop_ctx);
  }
}

int main(int argc, char **argv)
{
        ...
        result = c2_net_domain_init(&ndom, &c2_net_usunrpc_xprt);
        result = c2_net_event_register_callback(&ndom, event_handler, 0);
        ...
}
```

Support for a batching RPC upper layer

A particular example of a Colibri upper layer protocol that would call the c2_net layer would be the “item” batching module below the “rpc” layer (as per the AR of the rpc layer). This layer is responsible for collating numerous “items” targeted to be sent to a single server and then sending them efficiently together in a single RPC call. Such items could be FOPs, ADDB records, etc. Each item would be expressed as a marshalled (XDR) bytestream, ready for transmission. Note that this is independent of the use of memory descriptors for data transfer within the FOPs - they continue to encode references to buffers using memory descriptors as desired, or not.

The batching layer would use some policies to decide when to transmit a batch, such as the max batch size, the max time an item may be delayed, the max items in flight, etc.. At this point, the module can utilize the scatter-gather support like that illustrated above when constructing the low level “rpc” to send (also expressed as a FOP). It can use the I/O vector to collect the disjoint items, and then send the batch of pre-marshalled items as *data* within in a single RPC without having to re-copy the items.

Such a batch message would be reasonably small, containing just a header describing the transmission (# items, epoch, array of item sizes, etc) and the memory descriptor of the batch. The receiving end would have pre-allocated memory for the message - the size of this pre-allocated memory buffer is one of the constraints on the overall batch size.

Commented [37]: This is one possible scenario. Another possibility is to send a batch "inline", i.e., as a single message without separate bulk. This method is suitable for meta-data operations (which are typically small) and avoids the overhead of additional bulk messages. —nikita_danilov

The receiving end would allocate a single contiguous memory segment to contain the item list. The items would be RDMA’d into this memory, then individually de-marshalled, based upon the size data in the RPC. Each de-marshalled item would occupy separately allocated non-transport managed memory, and would be enqueued for processing and eventually freed by an upper layer. After enqueing the de-marshalled items, the server handler could release the RPC memory buffers and send the RPC response to the client.

The upper layer would dequeue and process each item - if it is a FOP, the upper layer may have to extract memory descriptors and perform RDMA operations to handle the data transfer. Write buffers would have to be fetched (Get) from the initiating Colibri client in a non-blocking manner; the server only has to initiate the Put of read buffers. Eventually, the upper layer would have to submit a response FOP, for which it should utilize the local batching layer to send back the responses.

Finally, when processing FOP responses back on the initial Colibri client, care should be taken to explicitly wait for pending read buffers to be completed, as it is always possible that the FOP response preceded the buffer transfer.

Providing bulk transfer support in the Sunrpc transport

We will not be using the RFC5666 support, but instead emulating the c2_net interfaces above over normal TCP/IP.

Non-blocking server operation

The server is already well structured to support asynchronous operations without busy-wait. It uses low level sunrpc primitives to operate at a message passing level. In-coming messages are saved to a **struct work_item** and enqueued for a pool of worker threads to find and process. The worker thread body is the one that invokes the **s_handler** method.

It should not be very difficult to support asynchronous handler operation by saving the response fop in work_item, using the address of this structure as the cookie, and moving the cleanup logic from the worker thread body to the **so_reply_post** method.

Non-blocking client operation

Asynchronous client send is already implemented, though perhaps not stringently tested. The client worker code has to be extended to post an event when the call completes.

RDMA emulation

In the c2_net specification above, the initiator of the RDMA transfer operation is the server. In the usunrpc transport, if interpreted literally, that would translate into switching the client and server roles for the purpose of RDMA! However, this is not something I want to do, especially as we have to support a kernel client. Instead, we should emulate the desired effect without such a role switch.

Proposed solution:

- **c2_net_nb_stage(C2_NET_NB_READ_REMOTE)**
  Client will push buffer to server then post event indicating that the buffer is free.
- **c2_net_nb_stage(C2_NET_NB_WRITE_REMOTE)**
  In user space, a client will initiate a periodic pull on a background thread, until the server has the data ready for download. Post an event once the data arrives.
  The background thread should poll only when there are pending unsatisfied stage requests.
  The kernel space implementation plan is the same as the client, and as such, **it requires Colibri thread support in the kernel**.

- **c2_net_nb_transfer(C2_NET_NB_READ_REMOTE)**
  Server will rendezvous with buffer already pushed from client. If not yet arrived, enqueue the pending expectation. When it arrives, dequeue it and post event.
- **c2_net_nb_transfer(C2_NET_NB_WRITE_REMOTE)**
  Server will enqueue buffer for client to find. Post an event once the client retreives the data, and then “dequeue” it.

Procedures

We need to reserve a range of RPC procedure numbers for these special calls, beyond the range of the highest FOP code. This can be asserted during service start.

The special calls will be redirected by usunrpc_service_worker to an rdma dispatch procedure. Care should be taken when collecting statistics as these won't be FOPs.

We need the equivalent of the following procedures that can be called from the client side:

```
int push_buffer(uint64_t event_id, U32 total, U32 offset, U32 length, char *buffer, U32 eof);
int pull_buffer(uint64_t event_id, U32 total, U32 offset, U32 *length, char **buffer, U32 *eof);
```

The offset and total fields are required to support scatter gather using c2_diovec. There may be multiple calls required to upload or download a single network buffer; in no circumstance should the client or server attempt to collate the data for a single transmission. In both push and pull, the client tells the server the total length of the network buffer and the offset which it expects. In the pull case, the server sets the eof indicator when the last block of data has been read.

As these subroutines require XDRs, they will be represented with FOPs and the **fop2c** compiler will be used to construct them from the following:

```
/* appropriate network buffer descriptor */
DEFINE(c2_net_buf_desc, RECORD,
      _(nbd_event_id,    U64),
      _(nbd_access_mode, U32),
      _(nbd_total,       U32));

/* buffer representation */
DEFINE(sunrpc_buffer, SEQUENCE,
       _(sb_len, U32),
       _(sb_buf, BYTE));

/* the push call and response */
DEFINE(sunrpc_push_buffer, RECORD,
       _(sph_nbd,    c2_net_buf_desc),
       _(sph_offset, U32),
       _(sph_eof,    U32),
       _(sph_buf,    sunrpc_buffer));
DEFINE(sunrpc_push_buffer_resp, RECORD,
       _(sphr_rc, U32));

/* the pull call and response */
DEFINE(sunrpc_pull_buffer, RECORD,
       _(spl_nbd,    c2_net_buf_desc),
       _(spl_offset, U32));
DEFINE(sunrpc_pull_buffer_resp, RECORD,
       _(splr_rc,  U32),
       _(splr_eof, U32),
       _(splr_buf, sunrpc_buffer));
```

It is essential that the kernel and user space implementations share the same definitions, and that these remain private to these modules and not exposed to any user of the module.

User space implementation

In user space, any additionally required state will be maintained in **struct usunrpc_dom**, the transport domain structure in net/usunrpc/usunrpc_internal.h. The code will also take advantage

of state maintained in the **struct c2_net_domain** data structure where possible, to avoid duplication.

Each of the new methods mentioned in **c2_net_xprt_ops** would have a corresponding subroutine implemented in either client.c or server.c as appropriate, with the "**xo**_" prefix replaced by "**usunrpc**_". In addition, support for the special procedures required for bulk transfer will come from a new subroutine, **usunrpc_nb_handler**() in the same file. It will have similar semantics as the **s_handler**(), but will handle the special FOPs private to the transport. This new handler will need to use the new handler semantics, which means that the non-blocking server handler support must be in place.

The client does not need additional threads to perform the background I/O. There is an existing pool of threads already defined. The **usunrpc_client_worker** subroutine will be extended to perform this work. However, since the current implementation only expects a **struct c2_net_call** data structure on the queue, this will have to be replaced by something along the lines of:

```
struct usunrpc_client_work_item {
    enum { CWI_CALL, CWI_BUFFER } cwi_item_type;
    struct c2_net_call           *cwi_call;
    struct c2_net_buffer         *cwi_buffer;
};
```

The thread would own the buffer until its upload or download completes - it may have to busy/sleep wait in case of a download. A stage or transfer request would create such a work item and enqueue for the worker threads to find. There have to be at least 2 threads available to support asynchronous client send with RDMA.

The server side needs to maintain a list of buffers to be transferred. The **nd_transfer_buffers** list maintained in the **c2_net_domain** structure is adequate for this purpose. A **push_buffer**() call from a client can rendezvous with an existing buffer on the server transfer queue (unlikely, but possible), or be added to the queue with the expectation that it will be asked for soon (its **nb_event_delivered** field should be set to false to indicate that it arrived before being asked requested).

Kernel space implementation

In kernel space, any additionally required state will be maintained in **struct ksunrpc_dom**, the transport domain structure in net/ksunrpc/ksunrpc.h. Client specific code for rdma will be in core/net/ksunrpc/rdma_client.c.

The kernel implementation can use a similar model to the user space client implementation.

> The kernel client does not yet support the asynchronous **sio_send**() method. It is not required that this be implemented at the same time.

The background thread should be tracked through the **ksunrpc_dom** data structure. The thread can be started from **ksunrpc_dom_init**() and terminated from **ksunrpc_dom_fini**().

The existing kernel client, core/c2t1fs/main.c, does not call **c2_net_domain_fini**(). Its **cleanup_module**() subroutine should be extended to do so.

Implementation steps

The lib/vec.h c2_diovec routines must be coded and unit tested.
c2_net layer must be coded. Simple unit tests can be done on the list management and event posting and waiting.
Implement non-blocking usunrpc client and server first. (First work in user space before the kernel.)
Then extend to implement bulk support.
Write a ping client and server to test. With async and then with bulk.
Get thread support working for kernel.
Implement ksunrpc client - async first, then bulk.
Extend c2t1fs server to test.

Misc

Notes on the existing code

core/net contains the abstract network layer. I think this is where we define the new interfaces. We may have to extend existing interfaces - c2_net_xprt_init() does nothing currently, for example.

core/net/usunrpc, core/net/ksunrpc contain specific domains where we will implement the new layers or emulate the RDMA.

Note that there is no kernel component of core/net/net_srv.c, because colibri servers are all user space. The kernel module for sunrpc (Colibri interfaces) is defined in net/ksunrpc. It symbolically links the upper layer interface files (net.c, net_cli.c, net_utils.c, connection.c) from the parent directory.

Unit test code for network calls is in core/net/ut (net.ff, client.c, net_fop.h, net_fop_init.[ch]). The client and server are in the same process, communicating over the loopback interface. They are usunrpc based.

> net_fop_init.c inlcudes “net_u.h” (generated), then “net_fop.h” and then “net.ff”. The last file expands to define the fop format c2_nettest_tfmt.

The c2_diovec support subroutines in lib/vec.c should be implemented. Support should be provided to create a vector of a given length, with appropriately malloced vector and buffer

pointers. The free subroutine should release both malloc'd values, and the buffer referenced.

Async notifications

Lnet event kinds (from Isaac's LNet Doxygen on Wiki):

```
/**
* Six types of events can be logged in an event queue.
*/
typedef enum {
       /** An incoming GET operation has completed on the MD. */
       LNET_EVENT_GET,
       /**
        * An incoming PUT operation has completed on the MD. The
        * underlying layers will not alter the memory (on behalf of this
        * operation) once this event has been logged.
        */
       LNET_EVENT_PUT,
       /**
        * A REPLY operation has completed. This event is logged after the
        * data (if any) from the REPLY has been written into the MD.
        */
       LNET_EVENT_REPLY,
       /** An acknowledgment has been received. */
       LNET_EVENT_ACK,
       /**
        * An outgoing send (PUT or GET) operation has completed. This event
        * is logged after the entire buffer has been sent and it is safe for
        * the caller to reuse the buffer.
        *
        * Note:
        * - The LNET_EVENT_SEND doesn't guarantee message delivery. It can
        *   happen even when the message has not yet been put out on wire.
        * - It's unsafe to assume that in an outgoing GET operation
        *   the LNET_EVENT_SEND event would happen before the
        *   LNET_EVENT_REPLY event. The same holds for LNET_EVENT_SEND and
        *   LNET_EVENT_ACK events in an outgoing PUT operation.
        */
       LNET_EVENT_SEND,
       /**
        * A MD has been unlinked. Note that LNetMDUnlink() does not
        * necessarily trigger an LNET_EVENT_UNLINK event.
        * \see LNetMDUnlink
        */
       LNET_EVENT_UNLINK,
} lnet_event_kind_t;
```

Clients

- Async rpc failure - should be captured by the c2_net_call structure.
- Out of memory for async response - Can this happen? If so, it would be a general event not tied to a specific queued c2_net_call.
- An RDMA read from the server completed (memory descriptors can be freed) - tie with respose fop.
- An RDMA write timed out (or is this subsumed by the corresponding rpc that involved the rdma?)
- Transport failure

Servers

- Async rpc response failure
- RDMA read completed
- RDMA read failed
-

Questions/Doubts

- Should we use the existing sunrpc transport interfaces and just extend them?
  Lots to gain by doing so - writing a new transport is tricky and in anycase, throw-away work (once LNET is picked up). Nikita(?) almost has a working async send, though not one that uses a send/receive model, but below c2_net_send, who can tell.
  Supporting sunrpc based async send/receive and rdma in the kernel may be hard, especially given that this is throw away code. However the c2t1fs filesystem remains the most “reaslistic” candidate for proof-of-concept (after the ping server/client).
- Do IB flow credits matter at this level? Should look at lustre lnet usage to see if that is exposed.
- Is c2_net_cli_send actually used yet? I don’t think so - couldn’t find an example of usage. **This implies that all those nice worker threads in the server have not been tested!**
- The c2_service structure already has s_handler() for dispatch, but this is only for a service. Clients also need to receive events, so should a general event queue based structure be provided that could be used by both servers and clients?
  - Maybe. Other candidates: c2_net_domain, c2_net_service
- **Is struct c2_diovec to be used?**
  The struct c2_dio_cookie type is not defined, and the subroutines for alloc, free and register are not defined. The definitions sit in lib/vec.h.
  If a subroutine is passed a c2_bufvec, then c2_diovec is not useful as it has to copy the c2_bufvec to the c2_diovec; a pointer to a c2_bufvec would have been more useful.
- How would the match bits be used by an LNET transport? We could use a 64 bit counter and assign a number to each MD. Probably better than using the address of the MD, which could get reused fairly quickly.
- Lots of small data structures to be maintained - consider allocator abstractions so that free queues can be maintained. Probably transport level. Would need some high/low water marks?
- What additional statistics should be collected. How do we get at these stats?

Commented [38]: Hm.. the worker threads are not supposed to send anything: they receive and reply (and reply sending is not a send in the current design) —nikita_danilov

Commented [39]: It is not supposed to. It will remain undefined, used for type-checking only. Layers that actually deal with memory registration will cast it to a pointer to void. —nikita_danilov

Commented [40]: I would imagine that interfaces like `->xo_md_alloc()` will take c2_diovec as a parameter. —nikita_danilov

Commented [41]: Lustre uses a "xid"---sequential counter per connection for this. —nikita_danilov

Documentation

Must supplement existing documentation - not enough description. Would at least like to put in the control flow I wrote up, plus the new message passing based rpc support we will add.

Specific interfaces that could be described better:
c2_net_reply_post
c2_service.s_handler()