Colibri FAQ

This document is for asking questions and getting answers to architectural questions and topics. Architects may answer questions in this document, or discuss answers in the Skype Colibri channel. The architect leading the topic discussion should record any exchanges of information on the Skype channel here as well.

Please follow existing formatting for any additions to this document.

RPC layer core

- Q1. Typical usage scenario.
- Q2. Internal scenario.
- Q3. Resolving failures.
- Q4. Reply lifecycle.
- Q5. What is a session and what are its details?
- Q6. How transactions work in Colibri ?
- Q7. Can there be more than one rpc item's per slot which are executed but who's updates's are not persistent and only in memory ?
- Q8. Will there be attempt to send the new request, while the client is doing the replay operation ?
- Q9. Persistency of sessions
- Q10. can we make sessions as a first stage in rpc-item-processing pipeline, before grouping?
- Q11. Then can sessions be a pipeline stage, after grouping?
- Q12. If say c2t1fs, sends two fops on same update stream and forming phase puts them in a single rpc, then?
- Q13. If db5 transactions are durable, then how is it going to be possible to create something like lustre's "async journal commit"?
- Q14. Sender is only aware about update_streams. Is session internal to rpc layer or is it visible to user of rpc-layer (like c2t1fs)?
- Q15. In the use case, of resend, it mentions that the server fetches the fol record from fol and returns reply to client without execution of operation but in our case we are going to use persistent cache for caching of the reply then is there need of it? if we are going to get the reply from fol?
- Q16. Can't we just have update_streams instead of slots?
- Q17. Interaction between different components of rpc-layer
- Q18. Typical use scenarios of using session module

Resource Manager

- Q1. Is resource owner(c2_rm_owner) posses one resource(c2_rm_resource) at a time?
- Q2. What is remote resource? Is it from another domain, service, group or another node?
- Q3. Who is the owner? Is the it domain, service or node?
- Q4. How owner know about the resources of remote owner before requesting access?
- Q5. Will there be more than one resource servers, if yes how to locate them for resource requests?
- Q6. For generic functionality, will database transactions(Log) required ? if yes, please explain why and how?
- Q7. There are two reference variables. c2_rm_resource_type::rt_ref and c2_rm_rsource::r_ref. Both are guarded by c2_rm_resource_type:rt_lock. I am not able to understand why two reference counters. Your new updated rm.h says
- Q8. And using c2_rm_resource_type:rt_lock for both is what i feel is not good. please clarify and when to increment and decrement ref counter?
- Q9. Is de-registered resource type structure from one domain can be registered to other domain? if yes what is to be done about resource list in resource type.
- Q10. Does c2_rm_resource_del means just delete resource from list or also free the memory taken by resource? I think resource can be reassigned to other owner after

deletion.
Q11. What will be default priority for incoming request? I guess it is zero
Q12. Resource manager terms
Q13. What is a domain (is it a collection of resources)? Is there a domain id (and/or name)? Is there a domain manager/controller?
Q14. Does c2_rm_remote represent “proxy” for a remote resource? If yes, why don’t we call it proxy?
Q15. How do you become “owner” of a resource in first place?
Q16. Where is owner look-up mechanism?
Q17. In HLD you have specified call back_function for right_get() but it is not specified in DLD. Does right_timedwait() is call back function?
Q18. What is uses of right_get_wait() function?
Q19. There is no pseudo code for right_get_wait() and right_timedwait() functions. Are we skipping those for this sprint?
Q20. In outgoing_complete(og, rc) , what is to be done with rc parameter and when this function get called? Currently it is not called form any function in pseudo code.
Q21. What outgoing_delete() does ? Does it call outgoing_complete() ? If yes then what about “rc” on outgoing_complete value?
Q22. Is right_copy function is generic ? I think, No. It is resource specific. If it is not generic then will it be part of this sprint?
Q23. right_intersect(which checks that does right intersects and return true/false) , right_diff, right_copy functions will be resource specific, am i right? If yes then ri_datum in c2_rm_right has value which is resource specific.
Q24. Does net_locate() and right_meet() are generic or resource specific?
Q25. I think current RM implementation builds a cache of resource ownership and their rights.Additionally there could be protocol associated with it.
Q26. What about the scalability? How many resources do we expect per domain? Are the linked list good enough to handle resources, rights?
Q27. One thing I would like to get clarity on is lease=loan and borrow=sub-let? Is that correct?
Q28. Nikita: continuing on presivous discussion. If node A opens a file (and is the first one to open), is node A owner? Is it a debtor or lesee? - of that file.
Q29. File, it's meta-data and data are different resource types. right?
Q30. irrespective of resource type, for the first open, does node A have ownership of the resources corresponding to a file?
Q31. Loans would be typically given proxy servers and leases to clients.
Q32. when it is pinned/held ?
Q34. who decides the lease time?
Q35. Scope of Generic RM.
Q36. Interface between RM-fop and RM-generic
Q37. Can we use c2_rm_outgoing for RM:fop reply?
Q38. When are rem_id or loan_id generated ?

Management operations

Q1)What are the options specified in mount command ?
Q2)What are the return values from the management server for mount fop request ?
Q3)When mount fop will be sent ?
Q4)What are things expected by configuration fop?
Q5)When Server Registration fop will be send and what is expected for this ?

SNS/Layout

Q1. Let's consider two files *file1* and *file2.* Each file has its logical block map. The layout (or layout formula) maps it to physical disk blocks in a storage pool. The layout formula makes sure that mapping is fairly distributed and supports PD-principles.

file1 logical map L1 maps to physical disk map (or interim virtual map) P1.
file2 logical maps L2 maps to physical disk map (or interim virtual map) P2.

How does layout formula make sure that P1 and P2 are disjoint sets? In other words how does it ensure that there is no collision? How does formula/formulae work on a set of files? We always look at the formula in isolation (i.e. for a single file).

> Very good question, because it shows that one fundamental property of SNS is not mentioned in the documentation. Our SNS is an *object RAID*, not block RAID. This means that for every file, a set of component objects is created, and the file is striped across these component objects. A component object is, roughly speaking, a file without meta-data, stored on a storage server. Technically, a component object is a *stob* (storage object, see core/stob/stob.h). In a typical configuration, each file would get a component object for each device in a pool. Different files get different sets of component objects, so there is no risk of stepping on each other.

Q2. Let's consider a cluster of G nodes. The SNS layout will use a pool of G nodes. Now S more nodes are added to the cluster. Does this operation affect layout for the existing files that earlier used G nodes? Will new files use G nodes or G+S nodes? We think existing files will not be affected.

> In a cluster of G nodes with D drives each, the pool contains G*D drives and a file would be striped across all of them. When a new drive or a new node is added to the pool, new pool configuration is created. File layout refers to a particular pool configuration. Existing files are not affected by a configuration change. New files would use space on new devices.

Q3. Let's consider a cluster of G nodes. Will layout always use a pool of G nodes? Can it use less than G nodes? Is it driven by a user policy or is it algorithmically decided by layout manager?

> Yes, this is a matter of policy. An administrator might partition the cluster into a collection of pools with different fault-tolerance properties.

Q4. How do you handle layout matrix? Is this a layout formula? Does such matrix exist? How do you make sure that it does not become large? How is it stored persistently and in

memory?

I am not sure I understand what layout matrix is.
A layout formula is a way to produce large number of layouts by substituting parameters into a pattern. Let's look at the example. Consider layout formula of the form "for i-th drive in the pool, use a component object with identifier X + i as a component object for this layout", where X is a parameter. By substituting different values of X different layouts, using different disjoint sets of component objects, are produced. (Assume that component object identifiers are unique within a node.) By means of such formulae layouts that stripe a file over hundreds of component objects can be compactly represented. The idea is to have a small collection of these layout formulae into which parameters are substituted to produce layouts for specific files. With this, there is no need to store large layout descriptors with files—it's enough to store the parameters.

Q5. How is partial disk (i.e. bad blocks) failure treated? Does SNS do anything here? Does bad-block re-vectoring algorithm of disk take care of this?

There are two facets to this problem: (i) media error detection and (ii) error correction. To detect a bit rot, when a device returns wrong block contents on a read, we are using check-sums, calculated by the clients, and stored on drives as meta-data. For error correction we use SNS redundancy, that is, reconstructing block contents from parity and re-writing.

Q6. Let's consider a scenario where a node failure triggers SNS repair. After repair the spare space will be consumed. When will new spare space become available? Is SNS repair supposed to allocate spare space before the repair starts? Who controls the repair operation (admin, policy, algorithm)?

The repair space is allocated as part of parity group. Consider a file striped according to N+K layout pattern (that is, with N data units in a parity group and K parity units). Each parity group for this file contains N data units, K parity units and K spare units, N + 2*K units total. These N+2*K units are distributed according to layout mapping function across the pool. All layouts in the pool use the same number of parity units that is, K. This means that the pool is able to recover from up to K drive failures. This also means that each parity group in the pool has K spare units. When a drive fails, a space unit number is reserved for the recovery of this failure. Say, spare unit number X is reserved where 0 <= X < K. Then for each parity group, lost data are reconstructed into X-th spare unit of this group.

The course of the repair operation can be monitored by the administrator. Repair starts automatically, see HLD of SNS repair for details.

Q7. If user change a file properties (such as stripe size), does it change layout formula?

A user cannot simply change layout parameters. What user can do is to *migrate* a file to a different layout. This involves reading file data and writing them according to the new layout.

Q8. Is every stripe unit stored on a different container?

Typically yes. This is required for fault-tolerance. But in some corner situations, like container migration, multiple units of the same parity group can temporarily end up in the same container.

Container

Q1. What is a container?

We think of container as self-describing block storage entity (and hence transportable). It has two partitions (or parts).
The first partition contains database that describes state of the raw blocks inside the second partition. We understand that in reality there may not be two partition. But there are two parts of this block storage.
Is there anything more to container than what is described above?

First, please refer to the following documents:

- Colibri data organization,
- C2 Glossary (look for definitions of device and container).

Container is a higher level notion than device. A container can be a device, a part of a device, a buffer in memory or something based on meta-data tables.

You are right that containers are self-describing and they (some of them at least) can migrate.

Q2. Consider the storage hierarchy - client RAM->Proxy->Storage. Does flushing container on client RAM cascade the flushing down the entire chain till storage resulting inordinate I/O delays?

Not necessary. The proxy server can (and usually will) accumulate multiple updates in memory or in flash storage before re-integrating them to a higher level proxy. That is, our caching is *not* write-through.

Q3. How is container aggregation handled? How are two or more container databases merged?

Consider a b-tree index used by db5. In this index records are sorted within leaf nodes and leaf nodes are sorted by the key of first record. This property makes merging of two trees into one a much simpler process, by using *merge* sort. Ideally, if key ranges of the trees are disjoint, the merge can proceed by streaming leaves of a smaller tree into larger one. This is roughly how we are going

to merge our containers, the details are not settled yet.

Q4. What is the size of container database? Do we always reserve the worst case size for the database for a given size of block storage?

A typical data container is quite large and, as Data Organization document mentions a large cluster is expected to have up to hundreds of thousands of containers. This is not too large a data-base. The space for container data-base will be allocated dynamically. The data-base itself will be maintained by a DHT interface of our middleware library.

Q5. What triggers container migration? Can partial container be migrated?

Container migration is triggered by *cache pressure*. Cache pressure component monitors various cached (data, meta-data, *&c*.) and triggers cache re-integration based on a policy. See the diagram of C2 conceptual design. The policy decision is based on cache sizes, age of cached data, server load, network utilisation, *&c*.

Q6. What exactly is the difference between container and a storage object?

A storage object is anything that looks like an array of blocks that one can read and write. A container has some internal structure: it contains objects inside. These objects can be storage objects or some other kinds of objects, like meta-data objects. Some storage objects are used to implement containers. For example, a raw device is a storage object, and device container is implemented on top of this storage object.

RPC layer core

Q1. Typical usage scenario.

Assume, user invokes write(fd, buf, count) system call.
The control gets to c2t1fs_write() function.
Then user has to do the follwing:
a) acquire a distrbuted lock, protecting the extent being overwritten;
b) send RPC describing operation and affected data.
Every operation of listed implemented via FOPs sending.
Internaly aquiring of the lock looks is like the following:

```
fop = lock_fop_construct(...); // create fop corresponding to aquire operation
result = c2_rpc_submit(&fop->item, lock_service, ...); // submit fop to rpc component cache
c2_rpc_force(&fop->item); // force it to be send ASAP, immediate rpc formation and sending
c2_rpc_wait_reply(&fop->item); // wait for reply
```

Sending of RPCs internally starts copying user buffer into kernel pages and submitting them:

```
        for_each_page(...) {
                copy_user_page(buf + offset, page[i]);
                build fop for page[i];
                c2_rpc_group_submit(group, &fop->item, data_service[i], ...);
        }
```

After that FOPs placed into the cache.

Q2. Internal scenario.

Assume, that user continiously submitting FOPs into RPC component.
The rpc layer should do the following:
 * put items into sub-caches, associated with specified services;
 * monitor the occupancy of the sub-caches;
 * when there is enough pages in a sub-cache to form an optimal rpc---form it and send.

Typically, user application is able to generate dirty pages much
faster than the file sysytem is able to process them. This means,
that sub-caches will be almost always "full", i.e., contain enough
pages to form multiple RPC.

A server allows only a certain amount of fops to be "in flight"
concurrently, because for each such fop is has to reserve resources.
RPC layer should send rpcs, until this limit is reached (usually this
limit is always reached), and then it has to wait for a reply for rpc,
before it can send next one.

Q3. Resolving failures.

Assume scenario:
* A client sends a fop.
* A server receives the fop and executes the required updates in memory.
* The server sends reply back to the client.
* Later (potentially much later) the server commits updates to the persistent storage.

Then the server crashes and restarted before the reply was received by the client.
There are multiple possible situations:
 * the server hasn't received the fop in the first place (e.g., the network message was lost);
 * the server received the fop, but hasn't completed its execution;
 * the server received the fop and completed its execution.

After the server failure, file system consistency have to be restored. Specifically, we want the client and the server to have the same view of the file system. To achieve this, the client will send the FOP again (resend).

When the server receives resent fop it has to determine whether the fop has already been executed or not. To tell whether a fop has been executed, the server maintains so called "reply cache".

This brings as to the notions of a session and a slot within a session. A session is established between a client and a server. It is a collection of slots. The number of slots is negotiated dynamically between the client and the server.

For simplicity, we can think that each slot contains a single number: the identifier of last fop send along this slot. This number is incremented by a client when a fop is sent via the slot.

The server maintains a matching number. This allows server to detect re-ordering, duplication and loss of fops. The key property of session is that slots are stored persistently (in some data base) and updated transactionally with FOP execution. When a server receives a FOP, it executes the fop in some transaction and in the same transaction increments the identifier in the slot. Because of this transactionality, when the server restarts after a failure, the slot in the data-base contains identifier of the last fop executed. This allows the server to distinguish fops that have already been applied (they have identifiers less or equela than one in the slot) from ones that haven't their identifiers are greater.

Q4. Reply lifecycle.

Reply life cycle is much simpler that for a forward fop. Replies are never resend and never replayed. Simply because if a reply is lost, the originator will resend the forward fop.

Q5. What is a session and what are its details?

- Session is shared by a client and server. Each session is uniquely identified by the session identifier.

- There are multiple slots in the session. Each session is associated with a particular incarnation (instance of the client). Client is identified by the client id.

- Each session has a slot table associated with it. Slot id is the index into this slot table. Each slot has a state associated with it which indicates whether that slot is in-use or not. It indicates that whether there is an out-standing request.

- There are two operations which need to be performed related to sessions viz. SESSION_CREATE and SESSION_DESTROY.

- For each request execution there is the entry on the reply in the reply cache (persistent cache). Note that for certain fop types, e.g., READ, reply cache does not contain the whole reply state, because it is too large. Instead the cache contains a pointer to the location of data (for eg. the block addresses of the data).

- Session state is stored in the data-base and updated transactionally.

- A slot represents an update stream, that is something through which rpc items (for eg fops, addb records) are sent serially with the EOS.

- Client remembers two numbers per slot : last-sent-rpc-item-id, last-committed-rpc-item-id.
last-sent-fop-id is incremented for each fop sent across the slot.

- The server keep last-seen-fop-id number for the slot. When a server receives a fop, it checks its id against last-seen-fop-id. This allows to check for lost messages (when there is a gap in fop-id sequence) and duplicate or re-ordered messages.

- If message is lost, the client resends it. if message is duplicated, the server ignores it. This deals with transient network failures.

- Colibri servers are asynchronous that is, they send a reply back to the client, before the transaction that was executed on the server as part of the operation commits (means becomes persistent).

- After a client reconnects to the server, the server returns last-seen-fop-id to it and the client _replays_ all fops with id-s larger than this, by sending them to the server again.

This means two things : -

- last-seen-fop-id must be stored on a persistent storage, transactionally with the fop execution.

- the client must remember all uncommitted fops. for this last-committed-rpc-item-id. It is send from a server to a client in replies. The client is free to discard all fops with id less than last-committed-fop-id from its "redo queue".

- Subtle point is in colibri we have slots for rpc-items and rpc-items are aggregated into rpcs. A single rpc will contain rpc-items from multiple slots. And a single rpc from a server might contain replies and last-committed updates for multiple slots.

- Session state is stored in the data-base and updated transactionally.After a server restarts from a crash, last-committed-id is always the same as last-seen-id.

- We can put mis-ordered fop in some (memory-only) queue with a timeout, hoping that preceding fop will arrive soon. This is a possible future optimization.

- We make use of fop for SESSION_CREATE and SESSION_DESTROY requests and replies.

- Only one reply per slot is cached.

- In general : in NFS, the server distributes slots to its clients. We want a more flexible protocol, where a client can indicate how many slots it wants.

- We should design our own algorithm for deciding the target_highest_slot_id value which the server sends to the client in the replies.

- An update stream is identified by a (session-id, slot-id) pair.
    - The upper layers that use rpc layer and that need ordered fop delivery, would create dedicated update streams and send fops across them in the proper order. Recall that "update stream" abstraction is exported by rpc layer. Some other users (e.g., addb) do not care about ordering, they

will send items not bound to any particular update stream. It is a responsibility of rpc layer to multiplex such unbound items across update streams.

- Some rpc items have replies, some do not. Some items are submitted within a specific update stream, for others (unbound items), the update stream is assigned by the rpc layer, &c.

- The persistent cache is the cache for all the item types.

- One useful method of clarifying design concepts is by enumerating and following their intended use cases. A typical use case for an update stream is an "FDML app", for example, a replicator. For the purposes of the present discussion, let's consider a very simple replicator model, consisting of two components: an agent running on a meta-data server and a processing component running on a separate node. The agent reads records from the server's FOL and sends them to the processing component. The latter "executes records", (somehow, we are not interested in the details), that is, replicate the changes, described by the FOL, to the target file system.

We want a sender (i.e., the agent) to be structured like the following:

```
for_ever {
        tx = tx_begin();
        fop = fol_get_next(tx);
        fop_send(tx, processing_component, fop);
        fop_reply_wait(tx, fop);
        tx_commit(tx);
}
```

and a receiver---like this:

```
for_ever {
        tx = tx_begin();
          fop = fop_receive(tx);
        fop_process(tx, fop);
        fop_reply(tx, fop);
        tx_commit(tx);
}
```

Specifically, we want the sender and the receiver to be relieved from a burden of handling various failures:

* network message re-ordering, duplication or loss;

* sender failure and restart with a loss of volatile state;

* receiver failure and restart with a loss of volatile state.

Furthermore, it is assumed that by default a transaction commit (tx_commit()) is asynchronous, that is, the results of a fop processing might be lost, due to a node failure, after the reply was sent.

Update stream is an interface abstracting these failures away. That is, an update stream is something across which items can be send to and received from, such that:

* items are delivered at the same order they are sent, without re-ordering, duplication or loss;

* if a receiver fails, all items that were sent, processed and subsequently lost in a failure, are resent;

* if a sender fails, it continues, after restart, from the last fop, processed by the receiver.

As one can see, update stream is a very useful and powerful abstraction: it masks network glitches from the participants and it masks failures of one participant from another. In fact, even if both participants fail at the same time, the update stream guarantees that processing after restart will be consistent with the persistent state that survived the crash.

Let's call these properties "full-duplex EOS" (exactly once semantics---a term used by the NFSv4.1 specification).

If one of the participants lacks persistent storage, in which case all tx-calls in its code are no-ops, a weaker form of EOS is still possible. This captures a use case, where the sender is a file system client and the receiver is a file system server.

An update stream is implemented by means of an "NFSv4.1 session" mechanism, which includes related data-structures of sessions, slots, reply cache, replay queues, &c.

The question of how exactly update streams should be mapped to the slots is a delicate one. Can multiple update streams share the same slot? Can an update stream migrate to a different slot in a "safe" situation (i.e., where there is nothing to reply)? These and similar questions should be addressed by the DLD.

- The update stream must survive a remote peer re-start, its identity should not change.

There are two possibilities for sessions:

* follow the NFS model: replay completes in the old session, after which a new session is opened. The update stream is reassigned to a slot in the new session;

* deviate from the NFS model: the same session is used after server re-starts.

To choose the right solution, it's necessary to look carefully at the NFSv4.1 at understand _why_ it enforces seemingly more complicated model.

Answer : -

NFSv4.1 mandates dropping a session of restart only to support simplistic servers that don't store enough state on the persistent storage. This is not applicable to Colibri. Our sessions should survive server failures and restarts.

- The server updates persistent rpc layer data-structures (reply cache) in the same data-base transaction where fop is executed.

Q6. How transactions work in Colibri ?

A typical server-side fop handler (see core/stob/ut/server.c:write_handler() for example) will function like this (it will be coded completely differently, due to non-blocking fom design, but interaction with the rpc layer is basically the same):

```
foo_handle(struct c2_fop *fop)
{
        tx = tx_begin();
        /* perform updates (add names to directories, create inodes, allocate and fill data blocks, &c.).
           All modifications are done as part of transaction. */
        result = do_foo(tx, fop);
        /* build a reply cache buffer. */
        construct_reply(fop, result, buf);
        /*
           Acknowledge the fop as delivered:

               - increment xid in the slot (as part of the transaction);
               - put the reply buffer in the reply cache (as part of the transaction).
         */
        c2_rpc_item_ack(tx, &fop->item, buf);
        /* Send the reply to the sender. */
        c2_rpc_reply_send(tx, &fop->item, buf);
        tx_commit(tx);
}
```

tx_commit() does not wait until the updates, made as part of the transaction, reach the stable storage. It simply does some log related book-keeping and returns immediately. One uses an expression "transaction commits" to mean tx_commit() call, that is, a point in time after which no further operations can be done in the context of this transaction and an expression "transaction is persistent" to mean that updates from the transaction reached persistent storage.

- There are two techniques for maintaining "last-committed": one which makes life easier for a client and another---for a server.

Client-friendly: a server monitors when transactions become persistent. For each transaction it determines which slot this transaction is associated with and updates last-committed in this slots in _memory_. The new values of last-committed for all slots are piggy-backed to any reply, which is sent back to the client. The downside of this approach is that a server has to maintain an additional structure in memory to store last-committed values.

Server-firendly: a server assigns a monotonically increasing transaction identifier for each transaction (this is needed for other purposes too). A fop is executed in a transaction. The identifier of this transaction is sent back as part of a reply. The server also stores in memory a single number: the identifier of the last persistent transaction ("last-committed-txid"). This number is sent to a client piggy-backed to all replies. The client orders items in its replay queue by the transaction number. When the client receives new last-

committed-txid from the server, it knows that all fops with smaller transaction identifiers have also committed and can remove them from the replay queue. The downside of this is that a server must commit transactions in the order of their identifiers.

Q7. Can there be more than one rpc item's per slot which are executed but who's updates's are not persistent and only in memory ?

Yes, a large number of such fops can exist. Storage is the slowest component of a cluster. To make storage operations more efficient, data-base engine gathers transactions in groups and sends them to the storage as a single batch. A fast meta-data server can have many thousands committed transactions in memory. All these transactions will be lost (aborted) on a server crash. Lost fops are replayed from clients. Multiple fops can be replayed for a single slot.

Q8. Will there be attempt to send the new request, while the client is doing the replay operation ?

If yes, then will the client have to do the provisioning for the delayed_list similar to that of lustre ?
If yes, then there can be at the most one element in that list, which would be maintained per slot and once the replay is done then sending of those rpc items from that list per slot would be done by constructing the rpc's out of them. Or the other way round, no need of this list and no new request's will be associated with the slots unless replay completes ?

Q9. Persistency of sessions

In that you gave the two use cases for update streams. One was of replicator and the other was of the c2t1fs. Update stream is an interface abstracting these failures away. That is, an update stream is something across which items can be send to and received from, such that:

*** items are delivered at the same order they are sent, without re-ordering, duplication or loss;**
*** if a receiver fails, all items that were sent, processed and subsequently lost in a failure, are resent;**
*** if a sender fails, it continues, after restart, from the last fop, processed by the receiver.**

So does this mean that we need persistency on the sender side also for all the rpc-items (fops) which may-be required to replay or resend after recovery from multiple failures takes place?
As per the discussion in that mail, I think for replicator it is needed and for c2t1fs its not needed ?
So, then do we have to design the structures on the sender side, by keeping in mind the persistency of the session related state or there is not persistent state maintained at the sender side?

In case of replicator, the sender is itself the server while in case of the c2t1fs sender is the client. If sender is a server, then its sessions are persistent. If it is a disk-less client, obviously is has no peristent state at all. Well... it would be persistent automatically, because the redo-queue in this case is stored in the FOL

- On a sender, in addition to fops we have file system objects, that the fops operate on (files, directories, &c.)

- All fops, affecting an object and linked into a queue hanging off this object.

- In this queue they are ordered by an object version number.

- There is an HLD on version numbers...

Note, that a fop might be linked into _multiple_ per-object queues.

E.g., LINK is linked into a parent directory queue and target object queue.

Q10. can we make sessions as a first stage in rpc-item-processing pipeline, before grouping?

grouping, clearly, must precede sessions, because sessions are directed to services, and it is grouping that maps items to services.

Q11. Then can sessions be a pipeline stage, after grouping?

Forming detremines which items go in an rpc.
Suppose, there is cached write into [0, 4K) bytes of a file F, and a cached write into [12K, 16K) extent of the same file...
Forming would detect this and _combine_ these items into a "vectored write" item.Which is a single item that occupies a single slot in some session. This is easier to do _before_ xids are assigned to items.

Q12. If say c2t1fs, sends two fops on same update stream and forming phase puts them in a single rpc, then?

when an item is sent via an update stream, a slot _is_ assigned from the very beginning.
there are "unbound" items, sent outside of update streams and "bound" items sent through an update stream.
For unbound item sessions module assigned slot and xid. For bound item, slot is already assigned (the slot is an attribute of the update stream) and only xid has to be assigned.
so, in some sense you are right: sessions _do_ precede all other components for bounded items.

Q13. If db5 transactions are durable, then how is it going to be possible to create something like lustre's "async journal commit"?

db5 transactions are _optionally_ durable.
c2_db_tx_init():
    flags = 0/*DB_READ_UNCOMMITTED*/|DB_TXN_NOSYNC;
DB_TXN_NOSYNC is the magic.

Q14. Sender is only aware about update_streams. Is session internal to rpc layer or is it visible to user of rpc-layer (like c2t1fs)?

well.. this is a subtle question. The whole reason to have multiple sessions in the same group is to attach different security attributes to them.
I cannot answer at the moment.

Q15. In the use case, of resend, it mentions that the server fetches the fol record from fol and returns reply to client without execution of operation but in our case we are going to use persistent cache for caching of the reply then is there need of it? if we are going to get the reply from fol?

Yes, we need reply cache, because some operations, e.g., read have no fol records.

Q16. Can't we just have update_streams instead of slots?

i) multiple update streams can be multiplexed over a single slot;

(ii) we want to export simple and clean abstraction from the rpc layer, hiding all the internal mechanics of sessions and slot inside and

(iii) in the future, more attributes can be associated with update streams.

Q17. Interaction between different components of rpc-layer

Assumptions {
* Multiple update streams can be associated with same slot.
* A slot is considered as "busy"
  FROM the point it assigns its sequence number (verno) to an item X
  TO the point, when session component gets notification about "reply of item X is received"
* item cache is maintained by GROUPING.
   * If an item is submitted along with update stream, then formation module does not do any kind of coalesce the item with any other item.
}

An rpc item can be sent with or without specifying update stream during c2_rpc_submit() {

If update stream is not specified then {
GROUPING: c2_rpc_submit() puts the item in item cache maintained by grouping.
GROUPING: The item becomes available to formation component.

GROUPING is not actually a module (in a sense FORMATION and SESSIONS are). GROUPING is a collection of indices inside of the FORMATION module. Some of these indices are "generic" (lists where items are sorted by dead-line, priority and target service) and others are item-type specific (e.g., for IO operations, an index where items are sorted by fid). Items on these lists are "cached". This cache is logically divided into two parts: "pending" items and "ready" items. "Ready" items can be send at any moment at the discretion of the FORMATION module. "Pending" items cannot be send immediately because of the restrictions imposed by the SESSIONS logic. That is, an item is ready if it is unbound or bound to a non-BUSY slot and is pending otherwise. State transitions in the SESSIONS module (e.g., a reply is received for a slot, a slot becomes BUSY) cause items to migrate between pending and ready parts of the cache. Whenever enough items are ready, the FORMATION module builds an RPC from them, subject to max-in-flight restictions. Note that this is a functional description, an implementation might look differently, but should function along the same lines.

Commented [1]: Text in blue is a comment by Nikita —amit_jambure

FORMATION: formation component calls c2_rpc_session_item_prepare() to fill session related details in the item.
SESSIONS: c2_rpc_session_item_prepare() finds out any "unbusy" slot in "any" ALIVE session with target end-point and assigns it to the item.

Note that for this to work, the FORMATION module must in advance know that there _is_ a non-BUSY slot.

FORMATION: Then formation embeds the item in some rpc and sends the rpc to output component.
}

If update stream is specified then {
GROUPING: c2_rpc_submit() instead of putting item in item cache, gives the item to session module along with update stream.
SESSIONS: Session module will put the item in c2_rpc_snd_slot::ss_ready_list. The c2_rpc_snd_slot will be identified by <session_id, slot_id> present in update_stream.
SESSIONS: If slot is not "busy" && c2_rpc_snd_slot::ss_ready_list has some item(s) then ----- (A)
session module takes out item from the head of ss_ready_list
fills all session related fiefs including sequence number (verno)
calls grouping module API to put the item in item cache.
GROUPING: makes the item available to formation component.
FORMATION: calls c2_rpc_session_prepare_item() on this item similar to "unbound" items case.
SESSION: session module doesn't do anything to fill this item because item already has all session related information.
FORMATION: puts the item in some rpc and hands over the rpc to output component.
}
}

When a reply item is received {
INPUT: component prepares a list of rpc-items from on-wire representation of RPC.
RPC-CORE: informs the session module c2_rpc_session_reply_received(*reply_item, **req_item)
SESSION: session module checks validity of reply item.
It advances sequence number of slot.
If ss_ready_list has some item then it repeats step (A)
if ss_read_list is empty then it marks slot as "unbusy"

Note that this contradicts the definition of BUSY you gave above, which means that a slot is un-busied as soon as a reply is received.

c2_rpc_session_reply_received() returns corresponding request rpc-item in an out parameter.
RPC-CORE: rpc-core informs formation module that reply is received, giving both request and reply items to formation module.
FORMATION: has some cleanup to do when reply is received.

The FORMATION module re-calculates the ready set, which possibly triggers immediate formation of a new RPC.

XXX Don't know what happens next :-). Can anyone fill it up?

Not much, I guess. The rpc is discarded.

}

Regards,
Amit

Thank you,
Nikita.

Commented [2]: Agree. As soon as the slot is marked as “unbusy” when reply is received, the condition [slot not busy && ready_list has item] satisfies. This results in taking item at the head of ready list and putting it in item cache and slot becomes busy again.
—amit_jambure

Q18. Typical use scenarios of using session module

* sender first creates a rpc connection with receiver
c2_rpc_conn_init(&conn, service_id, &rpcmachine);
This sends a conn create request to the receiver and immidiately returns.

* sender then waits for “connection establishment” to complete
c2_rpc_conn_timedwait(&conn, CONN_ACTIVE | CONN_INIT_FAILED, &timeout);

* sender creates a session with receiver
c2_rpc_session_create(&session, &conn);
Similar to conn create, above method simply sends “session create” fop to receiver.

*sender waits until session is created
c2_rpc_session_timedwait(&session, SESSION_ALIVE | SESSION_CREATE_FAILED, &timeout);

* sender creates an update stream
c2_update_stream_init(&update_stream, &session);

* sender can then send items associated with update stream
c2_rpc_submit(&svc_id, &update_stream, item, prio, deadline);

* there are routines to terminate and finalize session, rpc connection.

Resource Manager

Q1. Is resource owner(c2_rm_owner) posses one resource(c2_rm_resource) at a time?

c2_rm_owner has pointer to c2_rm_resource and there are lists of rights and loan like borrowed, sublet, owned , etc. Are these multiple right relates to one owner and one resource? I think owner may posses multiple resources.

An owner owns usage rights for a single resource:

rm.h:
A resource owner (c2_rm_owner) represents a collection of rights to use a particular resource.

Q2. What is remote resource? Is it from another domain, service, group or another node?

Yes, a remote resource is a resource located in a different domain. In a typical setup, there will be a domain per service which, in turn, means a domain per address space.

Q3. Who is the owner? Is the it domain, service or node?

Owner is separate from domain, service or node. Typically, for any resource there will be at most one owner in a domain. For example, a client has an owner for each (mount-point, inode) pair that is actively used.

Q4. How owner know about the resources of remote owner before requesting access?

I am not able to understand or visualize architecture of resource manager in colibri. I know TORQUE/SLURM resource manager(Linux cluster) which has one service which collects the resource info(disk,cpu,memory,etc) for each node and maintains the databases. Each node has client which provides info to main node. While submitting job user requests resources. How this is in case of colibri? Obviously colibri is file system and not the cluster management suit.

I would advise you to start from the HLD, that defines what resource management in Colibri is and gives some use cases. Note, that resources in Colibri are very different from ones that SLURM-like resource managers deal with.

Q5. Will there be more than one resource servers, if yes how to locate them for resource requests?

rm.h:c2_rm_remote:

A remote owner is needed to borrow from or loan to an owner in a different domain. To establish the communication between local and remote owner the following stages are needed:

- a service managing the remote owner must be located in the cluster. The particular way to do this depends on a resource type. For some resource types, the service is immediately known. For example, a "grant" (i.e., a reservation of a free storage space on a data service) is provided by the data service, which is already known by the time the grant is needed. For such resource types, c2_rm_remote::rem_state is initialised to REM_SERVICE_LOCATED. For other resource types, a distributed resource location data-base is consulted to locate the service. While the data-base query is going on, the remote owner is in REM_SERVICE_LOCATING state;

- once the service is known, the owner within the service should be located. This is done generically, by sending a resource management fop to the service. The service responds with the remote owner identifier (c2_rm_remote::rem_id) used for further communications. The service might respond with an error, if the owner is no longer there. In this case, c2_rm_state::rem_state goes back to REM_SERVICE_LOCATING.

  Owner identification is an optional step, intended to optimise remote service performance. The service should be able to deal with the requests without the owner identifier. Because of this, owner identification can be piggy-backed to the first operation on the remote owner.

Q6. For generic functionality, will database transactions(Log) required ? if yes, please explain why and how?

Yes, some persistent state will be needed for the generic resource manager. Typically, the records about granted, borrowed, sublet, *&c*. rights are volatile, but sometimes they should survive node failure and restart. For example, consider a right to use an extent of a certain file for write. A client sublets this right by querying the server. In a typical situation there is no need to make the record about subletting persistent, because if the server fails and restarts, the client would replay the right request. However, persistency is needed when a *disconnected mode* of client operation is supported, where a client can leave the cluster for a potentially long time and the rights that it obtained must be remembered to avoid conflicts.

Q7. There are two reference variables. c2_rm_resource_type::rt_ref and c2_rm_rsource::r_ref. Both are guarded by c2_rm_resource_type:rt_lock. I am not able to understand why two reference counters. Your new updated rm.h says

None of the resource manager structures, except for c2_rm_resource, require reference counting, because their liveness is strictly determined by the liveness of an "owning" structure into which they are logically embedded.

Q8. And using c2_rm_resource_type:rt_lock for both is what i feel is not good. please clarify and when to increment and decrement ref counter?

c2_rm_resource_type::rt_ref counts the resources of the type. It is incremented when a new resource is added to this type (think a file is opened, a connection to a new server is made, *&c*.) and is decremented when the resource is removed.

c2_rm_resource::r_ref counts owners (in the domain) for this resource. This counter is incremented each time an owner is created.

Typically neither of these counters would be updated frequently, so I don't see the problem with protecting them with a coarse grained lock. Definitely not before we see some profiling data. :-)

Does this answer the question?

Q9. Is de-registered resource type structure from one domain can be registered to other domain? if yes what is to be done about resource list in resource type.

I think it is not, while de-registering resource type, all lists and pointers should be cleaned.

You are right. A resource type can registered with a single domain only

In same line, does resource is supposed to be added with other same type of resource type in another domain after deletion?

Could you rephrase the question, please?

Q10. Does c2_rm_resource_del means just delete resource from list or also free the memory taken by resource? I think resource can be reassigned to other owner after deletion.

c2_rm_resource_del() should be a dual of c2_rm_resource_add(). That is, it should not free the memory occupied by the resource structure.

There are right lists(borrowed, sublet, incoming,etc) in c2_rm_owner. what to do If the all these list are not empty when c2_rm_owner_fini function get called? I think cleaning all list will not be good idea as there are some rights borrowed from others, right are given to others and so on. Please specify action for each list.

I added a new section to the documentation recently.

```
 When a resource owner is finalised (ROS_FINALISING) it tears down the credit
 network by revoking the loans it sublet to and by retuning the loans it
 borrowed from other owners.

...
/**
 c2_rm_owner state machine states.
*/
enum c2_rm_owner_state {
        /**
           Terminal and initial state.

           In this state owner rights lists are empty (including incoming and
           outgoing request lists).
         */
        ROS_FINAL = 1,
        /**
           Initial network setup state:

              - registering with the resource data-base;

              - &c.
         */
        ROS_INITIALISING,
        /**
           Active request processing state. Once an owner reached this state it
           must pass through the finalising state.
         */
        ROS_ACTIVE,
        /**
           No new requests are allowed in this state.

           The owner collects from debtors and repays owners.
         */
        ROS_FINALISING
```

```
};

...
 Owner state diagram:

 @verbatim

                                  +----->FINAL<------+
                                  |      |           |
                                  |      |     |
                                  |      V           |
                                  +---INITIALISING   |
                                         |           |
                                         |           |
                                         |           |
                                         V           |
                                      ACTIVE                 |
                                         |           |
                                         |           |
                                         |           |
                                         V           |
                                      FINALISING-----+

 @endverbatim
```

Q11. What will be default priority for incoming request? I guess it is zero

Assume that an incoming request already has a priority assigned.

Q12. Resource manager terms

a. **owner**
b. **borrowed**
c. **loaned**
d. **leased**
e. **sub-let**

(I'm familiar with NFSv4 'leasing' protocol). Ownership is usage right to a resource for a given time duration. No matter how the usage right is obtained the usage right makes an entity owner of resource. In this context, what would borrowed/loaned/leased mean?

Owner is an entity with which resource usage rights are associated. For example, on a client there will be an owner of "quota" resource for each user and there will be an owner of "file data" resource for each file for which the client has some data in the cache.

Owners are arranged into a hierarchy: originally all rights for a given resource are possessed by some dedicated owner (typically located on some well-known server). Other nodes, for example, proxy-servers, borrow rights from the original owner. That owner is said to sub-let a right to a borrower. Other nodes (*e.g.*, clients) can further borrow from the proxy servers, *&c*. In other words, a right transferring transaction has the following

components:

- a lender (also called cedent or creditor) who sub-lets the right;
- a borrower (also called debtor) who borrows the right;
- a loan which is the right being transferred.

Leases, which are indeed NFS-style deadline on rights are not yet fully incorporated into the DLD. According to the position of an owner in the hierarchy described above, one talks about "upward" and "downward" direction of operations.

Little more elaboration based on discussion with Nikita. All these term encapsulate distributed resource management within a Colibri cluster. These term encompass entities, protocol primitives and states. We will use analogy of client-server architecture to explain these terms. Please note that this architecture is not limited to client-server model; but can be evolved into peer-peer model. Although the terminology is explained in terms of client-server paradigm, the client and server (explained below) may not exactly correspond to Colibri client and server.

- **Owner** Resources are distributed within all the systems(nodes) in a cluster. The **owner** can be thought of as resource server. Obviously there is no single resource server. A system or a node could be a server (owner) for one resource and client(debtor) for another resource.
- **Borrow** Distributed resource management needs a protocol. **Borrow** is a protocol primitive that sends a request from a client to the server (resource owner) to obtain a desired usage right to the resource.
- **Sub-let** It's another protocol primitive for distributed resource management protocol. It's a successful response to a Borrow request. This results in granting a desired usage right to a client.
- **Lease** **Lease** represents a state of resource on a client. It means certain usage right for a given resource for certain duration of time.
- **Loan** **Loan** represents a state of resource on a client. It means certain usage right forever until revoked by the server (resource owner). Loan can be thought of as a lease with no timeout.
- **Creditor** It represents the state of a server/owner with respect to a resource. Once the owner (resource server) grants the usage right to a client, the server becomes **Creditor** for that resource.
- **Debtor** It represents the state of a client with respect to a resource. Once the owner (resource server) grants the usage right to a client, the client becomes **Debtor** for that resource.

Please note that a resource client can become a proxy resource server (owner); thus creating a hierarchy of owners. This in turn will facilitate peer-peer relationship with respect to resource ownership.

Q13. What is a domain (is it a collection of resources)? Is there a domain id (and/or name)? Is there a domain manager/controller?

A domain is a root data-structure from which all other resource manager structures eventually hang off. It is needed to allow multiple instances of resource manager to co-exists in the same address space.

Q14. Does c2_rm_remote represent "proxy" for a remote resource? If yes, why don't we call it proxy?

Interesting you mention this, because it was originally called c2_rm_owner_proxy. But then I recognized that it is something much simpler than a proxy object. c2_rm_remote is nothing more than an kind of identifier for a remote owner.

Q15. How do you become "owner" of a resource in first place?

As described above, the owners are arranged into a tree-like hierarchy. The root owner is initialised with the full set of rights (by c2_rm_owner_init_with()). Other owners borrow rights.

Q16. Where is owner look-up mechanism?

It is described in the comment on struct c2_rm_remote.

Q17. In HLD you have specified call back function for right_get() but it is not specified in DLD. Does right_timedwait() is call back function?

The call-backs are in struct c2_rm_incoming_ops, which is pointed to by c2_rm_incoming::rin_ops. ->rio_complete() call-back is invoked when the request enters RI_SUCCESS or RI_FAILURE state. Additionally, the caller can wait on c2_rm_incoming::rin_singal channel, on which request state changes are broadcast.

c2_rm_right_timedwait() waits (using ->rin_signal) until the request enters RI_SUCCESS or RI_FAILURE state or timeout occurs.

Q18. What is uses of right_get_wait() function?

c2_rm_right_get_wait() is a helper function that combines c2_rm_right_get() and c2_rm_right_wait(). See examples in rm/ut/rmut.c

Q19. There is no pseudo code for right_get_wait() and right_timedwait() functions. Are we skipping those for this sprint?

No. The DLD does not necessary contain the pseudo-code for everything. In fact, if you look over existing DLDs they typically contain no pseudo-code at all. The interfaces should be documented (which they are not yet).

Q20. In outgoing_complete(og, rc) , what is to be done with rc parameter and when this function get called? Currently it is not called form any function in pseudo code.

It is called when an outgoing request has completed. That is, when a reply has been received or timeout happened. The rc parameter is the request result. 0 for success, and -ve errno for failure,

e.g., -ETIMEDOUT for a timeout.

Q21. What outgoing_delete() does ? Does it call outgoing_complete() ? If yes then what about "rc" on outgoing_complete value?

outgoing_delete() finalises the outgoing request after its completion. It removes the request from the lists and frees it.

Q22. Is right_copy function is generic ? I think, No. It is resource specific. If it is not generic then will it be part of this sprint?

right_copy() would call some resource type operation to copy a right. As part of this sprint we should implement a couple of simple resource types.

Q23. right_intersect(which checks that does right intersects and return true/false) , right_diff, right_copy functions will be resource specific, am i right? If yes then ri_datum in c2_rm_right has value which is resource specific.

Yes, you are right and this is documented:

The meaning of a resource right is determined by the resource type. c2_rm_right is allocated and managed by the generic code, but it has a scratchpad field (c2_rm_right::ri_datum), where type specific code stores some additional information.

Q24. Does net_locate() and right_meet() are generic or resource specific?

net_locate() is generic, but not part of this sprint, right_meet() is resource type specific.

Q25. I think current RM implementation builds a cache of resource ownership and their rights.Additionally there could be protocol associated with it.

Correct, there is a network protocol (a collection of fop types) and a storage formats for resource data-bases.

Q26. What about the scalability? How many resources do we expect per domain? Are the linked list good enough to handle resources, rights?

For some resource types there will be a large number of usage rights on the owner lists, especially on a server.

The plan is that for now we use linked lists to get a working code. Later, we would allow resource types to use their own data-structures (for example, interval trees for extent locks) and instead of scanning the lists, some resource type methods would be called to find all rights "intersecting" with a given right.

But we need working code first to understand what these methods should be.

Q27. One thing I would like to get clarity on is lease=loan and borrow=sub-let? Is that correct?

Ah, a lease is a _timeout_ on a loan.
Debtor borrows, creditor _sub-lets_.

Loan transfers the ownership from creditor to debtor. Onwership reverts back to the creditor either when loaned right is revoked (involuntary operation), cancelled (voluntary) or expires (loan lease expires).

OK. Imagine there is a bank.
I have an account with this bank.

Q28. Nikita: continuing on presivous discussion. If node A opens a file (and is the first one to open), is node A owner? Is it a debtor or lesee? - of that file.

There are a few resource types associated with a file: meta-data locks and data (extent) locks at least.

Q29. File, it's meta-data and data are different resource types. right?

There may be multiple resource types associated with file meta-data and file data extents belong to a separate resource type.

Q30. irrespective of resource type, for the first open, does node A have ownership of the resources corresponding to a file?

Let's look at the data case, for simplicity.
That is, we ignore the whole open story and start with the actual IO (read and write) from a client.

There is a resource type, that we shall call, for the sake of discussion file_data.

Resource usage rights of this type have form (ACCESS:[START, END)).

Where ACCESS is READ, WRITE or READ|WRITE.

Such a right grants corresponding access to the specified extent of file

Extent maybe (and frequently is) larger than the file itself (that is, either START or END or both can be larger than file size).

c2_rm_right is a collection of these (ACCESS:[START, END)) elements.

In a typical implementation, c2_rm_right::ri_datum would point to a linked list or some other data-structure.

Imagine the file lives on a certain server (ignoring striping for simplicity)...

On this server there will be a c2_rm_owner that "possesses" a (READ|WRITE:[0, ~0)) right.

For this owner: ->ro_borrowed and ->ro_sublet are empty, ->ro_owned == (READ|WRITE:[0, ~0)).

OK. Now suppose a client C0 wants to read the first 1MB on the file.

On the client, a local owner c2_rm_owner is created (initially all its lists are empty)...

And a call to c2_rm_right_get() is made as a part of read(2) system call.

It is determined, that the request cannot be fulfilled "locally", because there is not enough possessed rights.

And there are no sublets to revoke.

So, ROT_BORROW outgoing request is sent to the server.

Let's move in a smaller steps.

On the server, this outgoing request creates an incoming request for (READ:[0, 1M)).

This request can be fulfilled locally.

Once it is fulfilled, the server has ->ro_owned == (READ|WRITE:[1M+1, ~0)), ->ro_sublet = (C0, (READ:[0, 1M))).

The reply is sent back to C0 and when it is received, the processing of the original request completes.

Now, the action of transferring (READ:[0, 1MB)) from the server to C0 is called a loan. The server "sub-let" the right to C0. C0 "borrowed" the right from the server. The server is an "upward" owner for C0 and C0 is a "downward" owner for the server.

A loan is associated with a timeout that is called a "lease" (it may be infinite).

Client ro_borrowed = (S, (READ:[0, 1MB)))

and its ->ro_owned[] is the same.

Q31. Loans would be typically given proxy servers and leases to clients.

if by "loan" you mean a loan with an infinite lease, then perhaps yes. But _every_ loan has a _lease_ (a timeout) associated with it.

It can be cancelled, but before cancel succeeds, data must be removed from the cache.

We cannot use loan and lease interchangeably. A lease is a timeout on a loan, it's roughly a single c2_time_t value, associated with a struct c2_rm_loan.

Everything given out is loan for borrower. Lease is only specifies the duration for which loan is given.

Q32. when it is pinned/held ?

A right is pinned when it is actively used. For example (READ:[0, 1M)) right is actively used while

read(2) system call executes.

While a right is pinned it cannot be cancelled.

When the call completes, the data protected by this right are still in cache, but right is no longer pinned.

Q34. who decides the lease time?

some algorithm.

Q35. Scope of Generic RM.

There are a few near-term problems with RM. One is that we cannot complete generic code because the rpc layer is not functional yet.

The interfaces to send and receive fops are already in place, but their are not implemented...

In go_out(). It should actually send the fop out.

And there should be a code that calls outgoing_complete() when a reply is received.

we take existing rpc layer and fop interfaces and use them. This means defining fop types for all types of outgoing rm request and their replies.

If so, we create a simple testing infrastructure where rpc passing will be simulated with function calls. This implies that all "remote" domains are actually in the same address space which mean testing with producer-consumer like testing infrastructure.

Q36. Interface between RM-fop and RM-generic

Processing Request: An RM-FOM corresponding to an RM-request is run. All the FOMs eventually call *c2_rm_right_get()* by filling out structure *c2_rm_incoming*. RM-generic queues the request and RM-FOM goes to sleep (reqh-wait-queue). Once RM-generic completes the processing, it will wake-up the FOM. FOM now needs to fill in the fields in the reply-FOP. However all the information is not available in *c2_rm_incoming* (which is embedded in the FOM).
How does RM-fop interface with RM-generic for this purpose? Also when the request comes in, how can RM-FOM identify owner? Which function can it use?

Answer: all the information, necessary to fill in the reply is (or should be) available in c2_rm_incoming. *Viz*., the join of rights on the c2_rm_incoming::rin_pins is exactly the right the is to be returned to the sender. What information do you think is lacking?

Following steps were decided for this:

1. Get resource type object: c2_rm_restype_get(uint64_t restype_id)
2. Use rest_type_ops to obtain resource object
3. Use resource object to obtain the owner

Following is the suggestion to change certain ops vector. This was discussed on Colibri T1. This is a proposal:

```
struct c2_rm_resource_type_ops {
    bool (*rto_eq)(const struct c2_rm_resource *resource0,
                   const struct c2_rm_resource *resource1);
    int  (*rto_res_get)(uint64_t restype_id,
                        struct c2_rm_resource **resource);
    };
```

There is no clarity on effect of c2_rm_resource_ops->rop_right_decode() on encoding/decoding. It's not clear if need bufvec at RM layer. RM:generic will interact with RM:fop. Decoding resource from bufvec into FOP will be awkward. Again FOP gets converted to bufvec.

Processing Reply: RPC-layer will call the callback function once it receives response for a RM-request-fop. As a result of processing reply, RM-fop reply callback needs to find out the RM-generic callback or channel that it can signal. Further it needs to return either the error code or the resource, rights values to RM-generic. Should RM-fop embed a cookie (pointer to *c2_rm_incoming* on the sender side)? How can it communicate the data back to RM-generic?

Answer: c2_rpc_item_ops::rio_replied() call-back takes the pointer to an rpc item as a parameter. Some structure like

```
struct out_packet {
        struct c2_rm_incoming *op_in;
        struct c2_rm_outgoing  op_out;
        struct c2_rpc_item     op_item;
};
```

should be defined and container_of() used to get to it from the rpc item.

Q37. Can we use `c2_rm_outgoing` for RM:fop reply?

Let's consider the interaction between a debtor and creditor for a resource Borrow request:

1. Debtor send a resource Borrow FOP to creditor.
2. Creditor on receiving the request, start a Borrow FOM.
3. Borrow FOM finds following resource objects, based on resource type id and resource id:
    a. Resource Type object
    b. Resource object
    c. Resource Owner object

4. Borrow FOM now allocates `c2_rm_incoming` structure and fills it in. It then calls `c2_rm_right_get()` and waits.
5. After Borrow request is processed by `c2_rm_right_get()`, a RM:generic function wakes up Borrow FOM.
6. Borrow FOM now fills up the reply FOP. It is looking for:
    a. rem_id (It's part of `c2_rm_loan.rl_other.rem_id`). It's not directly accessible. RM:generic can expose it in some data-structure or provide a function to obtain this value. Providing API is not a good solution as it will impact performance.
    b. Resource type id (Available in Borrow request FOP. It can be copied)
    c. Resource id (Available in Borrow request FOP. It can be copied)
    d. loan_id (`c2_rm_loan.rl_id` Currently not accessible). RM:generic can expose it in some data-structure or provide a function.
    e. Lease Time. Not available. Currently using 0.
    f. Resource right data (`c2_rm_right.ri_ops.rro_encode()`). Currently it's accessible from `c2_rm_incoming.rin_want`. *Here right data length is not known. Hence allocating data for resource right is a challenge*. We may have to add a function such as `ri_ops.rro_right_len()`.
    g. Resource data. Currently not used.
    h. error code. Currently relies on `c2_rm_right_get()`. This function is currently blocking. It should be made non-blocking. After that error-code must be supplied by RM:generic in some form.

Can we use c2_rm_outgoing to provide information for Borrow Reply? Other requests don't need much information (and most of it is available in request FOP).

Q38. When are rem_id or loan_id generated ?
Ans:

The purpose of rem_id and loan_id is similar to the purpose of "sender_id" in sessions module. It is generated by the receiver (think "server") to speed up further communications with the sender.

Let's look at a use case.
Suppose there is a client that wants to take locks on a file with the fid F. The client connects to the rm service and sends a BORROW fop. This fop contains right description. For generic RM this description is an opaque byte array. It's interpreted by the resource type code. Suppose this description has a form (FID, READ|WRITE, START, END).
When the FOP is received, the corresponding owner has to be located by fid. This incurs searching through some data-structure, like linked list or hash table. Later, the same client wants another lock on the same file. In sends another BORROW fop. Naive solution would be to include right description in the FOP. In this case, the server would have to search for the owner again.

There can be a large number (1e5 or more) clients talking to the same server. In this situation avoiding this search can be a large bonus. To avoid this repeated search, the first time when the search is done, we generate some _cookie_. When the client talks back to the same service and to the same owner, it sends back this cookie. This cookie is used to speed the search up. The simplest form of this cookie is the _address_ of c2_rm_owner structure in server's memory.

There are three important points:

1. Cookies are optional: the receiver must be ready that sender chooses to send either cookie or right description. The fop should be structured appropriately (use UNION).

2. A sender cannot interpret cookies. They are opaque byte arrays to it. Specifically, a receiver might decide to avoid using cookies for a particular sender (e.g., because it doesn't trust the sender) and send 0-cookie instead.

3. The receiver should perform some validation of a cookie, before using it. Perhaps uint64_t is not large enough to validate. In this case, cookies should be enlarged.
Similarly for loan_id.

Management operations

(mount,unmount,configuration and server registration)

Q1)What are the options specified in mount command ?

Ans: endpoint for connecting to management service and profile for multiple file systems.
profile is passed as a parameter to the management service mount fop.
So command would look like "mount -t c2t1fs -o endpoint:profile /mnt/c2t1fs "

Q2)What are the return values from the management server for mount fop request ?
Ans: rood fid in case of success and error code in case of failure.

Q3)When mount fop will be sent ?
Ans:

```
mount()
    --->sys_mount()
        -->get_sb()
            --->c2_fop_mount()
```

It could be called in system call handler , so it should be in kernel mode

Q4)What are things expected by configuration fop?

Ans:

c2_fop_mgmt_config fop will be sent from sender (client or data server) .

On management server side corresponding fom will be executed and which sends the configuration as reply to the sender.

Q5)When Server Registration fop will be send and what is expected for this ?

Ans: c2_fop_server_registration

server sends the fop to the management server and where it executes the fom in which it registers the sender and sends the status back to the sender.

Q6)**is there any need to create management service on the management server (to connect to it by using Endpoint) or is it already running on it?**

Ans: yes for a time being

Q7)**What could be done on client side after mount reply received?**

scenarios:

ans:root fid is associated with root dentry at mount point.

cd /mnt/c2t1fs

Q8)How the management server gets the root fid?

Any: currently a superblock will be created having rood fid and other parameters.

Q9)What could be done on client side after mount reply received?

Whether it is mapped to inode or not.

How the root fid is going to be used in open,read and write calls?

ans:root fid bounds to a directory at mount point , user process looks up it and open , read and write operations are carried out.

Q10)after mount command , is it necessary to establish connections with data servers also to do read and write operations?

Ans:configuration gives the information of data servers to which client is needed to connect after mount operation, for doing read and write operations.