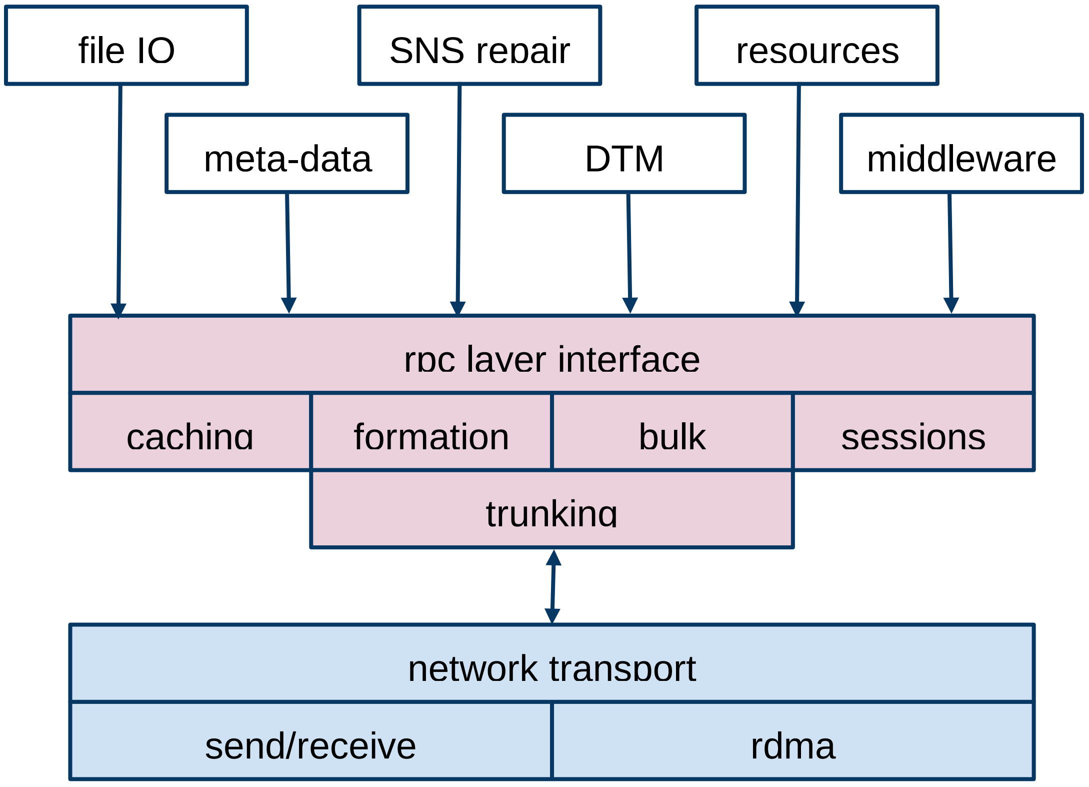
Architecture review of Colibri rpc layer

Colibri rpc layer comprises the higher level part of the network stack. Its purpose is to provide higher layers with an interface for communication with remote parts of the system that

- is flexible enough:
  - supports synchronous and asynchronous communication;
  - supports priorities;
  - supports sending various auxiliary information, like addb records;
- is efficient enough:
  - the network bandwidth is utilized fully by sending concurrent messages;
  - the messaging overhead is amortised by issuing larger compound message;
  - 0-copy, if provided by the underlying network transport, is utilised.

In addition, rpc layer should support ordered exactly once semantics of delivery, but this functionality is beyond the scope of the present document (it belongs to the sessions component).



Cache

As described in the Colibri networking 1-pager, Colibri rpc layer maintains a cache of *items* (a better name is needed. *Element*?) ready to be sent to the remote end-points. The layer contains a logic to place these items into containers called rpcs that are sent over the network. Cached items are directed at particular *end-points*, but rpcs are sent to *services* (network addresses). The mapping from end-points to services can change over time (see 1-pager).

Formation

The rpc formation algorithm is a subtle and critical part of an efficient IO system. It should take the following considerations into account:

- underlying network transport performs optimally when certain conditions are met. This includes:
    - a certain number of messages are in flight (between a pair of nodes). The algorithm tries to keep max-in-flight number of rpc to a given server in flight at any time. To see examples of this, search for cl_max_rpcs_in_flight in the Lustre sources;
    - messages should be large to amortise the message overhead (the overhead consists of networking hardware setup to send the message, various interrupt-related overheads at the receiver, *&c.*). The algorithm should try to form messages as large as possible, but no larger than max-message-size parameter. To see examples of this search for cl_max_pages_per_rpc in the Lustre sources;
    - rdma may impose restrictions on a total number of fragments in the message. That is, the message cannot contain more than max-message-fragments disjoint memory buffers. See comment about "fragmented" page array in osc_request.c:osc_send_oap_rpc() in Lustre;
- remote services perform optimally when certain conditions are met. For example:
    - a data service (handling data IO in a form of read and write fops) favours large rpcs for contiguous operations, with well-aligned starting position. See ending_offset and comments on it in osc_request.c:osc_send_oap_rpc() in Lustre;
    - meta-data service would benefit from an rpc where component fops are sorted by (say) fid of file, because this would result in a more sequential storage IO. No examples to look at, because Lustre doesn't have meta-data batching;
- upper layers can specify priorities of items;
- upper layers can specify caching properties of items. *E.g.*, for some items caching is write-through: the item is not cached and instead is immediately placed into an rpc, even is the latter would be sub-optimal. Other caching parameters, like maximal time the item can linger in the cache can also be specified.

Generally, the rpc formation algorithm is specified per (target end-point, network transport) pair, which implies that it should be carefully split into two cooperating policies.

Note, that at the moment T1 doesn't need a very sophisticated rpc formation algorithm, because there are no data caches on T1 clients (all IO goes in a cache-through mode), but the implementation should be extensible so that a Lustre-like algorithm could be plugged in.

Bulk

Bulk in the rpc layer deals with the network part of a complete 0-copy path. The general idea of bulk transfer is to split large fop (for example, read or write fop, containing data pages) into two parts: a small "setup" part and a large "bulk" part, consisting of pages. The setup part of the fop specifies operation parameters and contains information necessary to start rdma of the data pages. This message is received by the server and processed (possibly after some queuing delay). When the server needs actual data (for write) it initiates *bulk transfer*, *i.e.*, rdma of data pages from the client. When the server wants to return pages back to the client (for read), it initiates bulk transfer to populate client pages with data via rdma. Search for "bulk" in the Lustre source code for examples.

Bulk implementation interacts closely with the rdma capabilities exported by the underlying transport. At the moment, Colibri uses ONC RPC over TCP as a transport. ONC RPC, which by itself is stream-based, has an rdma extension. This extension is described in rfc5666. rfc5666 introduces rdma at the level of XDR encoding, making it transparent for the higher layers. T1 has an (almost completed) support for rfc5666 in the "rdma" branch. On the client side, this implementation (HLD, HLDIT, DLD is in the source as usual) uses rfc5666 compliant sunrpc module in Linux kernel, and on the server side, it uses Infiniband verbs through a user space ibverbs library.

It is planned, that Colibri would switch to use Lustre Networking (LNET) as its network transport layer. LNET is fundamentally designed around rdma as the main method of transmission, which would make bulk implementation on top of it simpler.

Upper layers, including rpc layer, access network transport through interfaces defined in core/net/. These interfaces might require changes to accommodate for the new rpc layer features. Specifically, an interface for rdma must be added that would be compatible with the LNET. For the time being (until LNET support is added) this interface should be backed by a dummy implementation.

Sessions

Colibri rpc layer supports ordered exactly once semantics (EOS) of fop delivery. To this end an abstraction of an *update stream* is introduced. An update stream is established between two end-points and fops are associated with the update streams. By cooperating with other sub-systems (for example, persistent storage and local transaction engine), the rpc layer is able to guarantee that:

- fops are delivered to the receiving service in the same order they were sent and
- a fop is delivered exactly once

These guarantees hold in the face of network and receiver failures (which makes their implementation non-trivial). How exactly this is achieved is described elsewhere, see the references in the 1-pager.

Trunking

Trunking module provides an ability to multiplex a session on top of multiple transport layer connections to utilise the aggregate throughput of multiple links. Underlying connections are different *physical* network addresses. Trunking module maintains mapping from physical to logical network addresses, which is a part of cluster configuration.

Interface

Colibri rpc layer exports interfaces to:

- add items to the cache, with caching flags described above;
- group item submission, used by a higher layers to add collections of items. The rpc formation algorithm should use groups as indications that more items would follow shortly;
- monitor status of the items;
- register handlers invoked when items of a particular type (identified somehow—this should be determined by the HLD) are received by the node. This part of interface should precisely define the ownership of the network buffers;
- special handling of reply fops. When a fop is sent from a service A to a service B, B usually sends a reply, which contains operation status and results, back. A reply goes through the normal caching and rpc-packing stages. The rpc layer should introduce methods to match replies with the "forward" fops (sessions already have all the data necessary for that) and to wait until the reply is received;
- bulk interfaces. Again, buffer ownership should be dealt with very carefully.

These components are spread over 5 T1 tasks:

- core: external interfaces and cache data-structures;
- sessions. The scope is clear. Note that current effort does not include "ordered" part of the guarantees (*i.e.*, there is no resend);
- batching: simple rpc formation algorithm (max-in-flight, max-message-size, max-message-fragments). This includes definition and dummy implementation of transport interfaces to return these parameters;
- bulk transfer: 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.
- FOP:core: wire format of rpcs. A fop already knows how to encode itself to the wire and decode back. The scope of this task is to define a format of rpc in terms of items (an abstract item data-type should be introduced with operations to query item size, alignment requirements, decode, encode, *&c.*) and provide implementation of this type for fops and addb records. In addition, rpc should have a common header storing generic information, like epoch numbers.

References

See Lustre sources as indicated above
See core/net/* for Colibri transport layer.