Date: June 05, 2013	**DAOS Server Collectives Design** **FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O**

LLNS Subcontract No.	B599860
Subcontractor Name	Intel Federal LLC
Subcontractor Address	2200 Mission College Blvd. Santa Clara, CA 95052

NOTICE: THIS MANUSCRIPT HAS BEEN AUTHORED BY INTEL UNDER ITS SUBCONTRACT WITH LAWRENCE LIVERMORE NATIONAL SECURITY, LLC WHO IS THE OPERATOR AND MANAGER OF LAWRENCE LIVERMORE NATIONAL LABORATORY UNDER CONTRACT NO. DE-AC52-07NA27344 WITH THE U.S. DEPARTMENT OF ENERGY. THE UNITED STATES GOVERNMENT RETAINS AND THE PUBLISHER, BY ACCEPTING THE ARTICLE OF PUBLICATION, ACKNOWLEDGES THAT THE UNITED STATES GOVERNMENT RETAINS A NON-EXCLUSIVE, PAID-UP, IRREVOCABLE, WORLD-WIDE LICENSE TO PUBLISH OR REPRODUCE THE PUBLISHED FORM OF THIS MANUSCRIPT, OR ALLOW OTHERS TO DO SO, FOR UNITED STATES GOVERNMENT PURPOSES. THE VIEWS AND OPINIONS OF AUTHORS EXPRESSED HEREIN DO NOT NECESSARILY REFLECT THOSE OF THE UNITED STATES GOVERNMENT OR LAWRENCE LIVERMORE NATIONAL SECURITY, LLC.

© 2014 Intel Corporation

Table of Contents

Introduction 1

Definitions 1

Changes from Solution Architecture 2

Specification 2

- **The API** 2
  - Collective Service 2
  - Broadcast to a Group 4
  - Service Callbacks: Request Propagation 6
  - Service Callbacks: Reply Aggregation 7
  - Broadcast Completion 7
  - Error Handling 8
  - Abort Requests 9
- **The Protocol** 9
  - Populate Group Membership 11
  - Build Spanning Trees 13
  - Broadcast Message 20
  - Reply Aggregation 23
  - Buffer Management 24
  - Group Lifespan 26
- **Design Notes** 26

API and Protocol Additions and Changes 27

Open Issues 27

Risks & Unknowns 28

Revision History

Date	Revision	Author
June 05, 2013	1.0	Isaac Huang
May 01, 2014	2.0	Isaac Huang

Introduction

Server collectives enable any server in a cluster to initiate an RPC that is executed on an arbitrary subset of its peers and complete with a result computed across them all. This mechanism must be scalable, adding at worst O(log n) overhead with increasing server cluster size, and robust, providing deterministic results in the face of failures of all sorts.

Server collectives will be used to implement capability distribution, collective commit and container layout replication across container shards. It may also be used to implement global notifications such as client eviction and permit more scalable client health monitoring and connection establishment.

Definitions

Server collectives allow a function to be executed over a set of entities called a group, for example a set of OSTs. A group member is identified by (tag, node), where node identifies a server and tag identifies an entity on the server, e.g. an OST. There can be multiple members on a single member node, for example (ost0, oss0), (ost1, oss0), and (ost2, oss0).

In addition, the following definitions are used throughout the rest of this document:

- **LNet**: the Lustre Networking stack. The server collective module calls the LNet API to send and receive point-to-point messages.
- **NID**: address of an end-point in a Lustre network, comprised of an address within its network and a network ID separated by a '@', for example 192.168.10.124@o2ib0. The NID is represented by C type lnet_nid_t.
- **LNET_MAX_PAYLOAD**: maximum payload size of a single LNet message, usually 1M bytes but never less than 1M.
- Upper layer services/protocols: users of the server collectives API, which invoke the collective service to broadcast messages and aggregate replies. For example, DAOS service daemons.
- Collective message: either a message broadcasted by root node and forwarded downstream by member nodes, or a reply message forwarded upstream from leaf nodes back to the root.
- Remote Memory Access (RMA) descriptor: a handle to an area in local memory that can be passed over the network so that peers can read from or write to it. The local memory area can be discontiguous.
- Member node rank: an integer of range [0, # total nodes in a group), which identifies a member node within a group of nodes. Rank is relative to group. A node can have different ranks for different groups it belongs to. Rank is used internally to identify member nodes and not exposed to users of the collectives API.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

Changes from Solution Architecture

No change has been made, but there are some unclear issues in the original Solution Architecture that need to be clarified:

- Server collectives are reliable in the sense that: A collective communication either fails, or succeeds with guaranteed delivery to all participants.
- The presence of LNet routers between server nodes is out of scope: There's no strong use case for the need of LNet routers between server nodes.
- Server collectives don't guarantee order. Broadcast messages may be delivered and processed out-of-order on member nodes.
- It's required that the size of a broadcast message can't exceed 4K bytes. Bulk data must be passed in the form of RMA descriptors. See Buffer Management for details.

Specification

The API

This section gives a brief overview of the server collectives API. The next section The Protocol describes how the API and callback functions registered through the API are involved in the complete process of a collective.

All API function names are prefixed by "scoll_" which stands for server collectives.

Collective Service

A collective service is registered with server collective, and is responsible for handling incoming requests and aggregating replies and forwarding aggregated back to root node. There can be multiple collective services on a node.

A collective service is registered with the scoll_service_register() API, which specifies a service ID and a set of service callback functions. Details on the callback functions will come later. Service IDs are statically assigned and agreed upon globally.

A collective service is unregistered with the scoll_service_unregister() API.

Entity, Target, and Group

A request is broadcast to a set of entities, called a group. An entity is a 128-bit tag that identifies an object on a node. A target is a node that consists of one or more entities.

A group is associated with a collective service, which handles broadcasts over the group. Therefore service ID, target NID, and entity tag together identify a unique object globally. For example, (OSS, 10.0.0.1@tcp, 1) and (MDS, 10.0.0.1@tcp, 1) identifies two different objects, even though the target and entity parts are the same.

Regular Group

A regular group can be created by API:

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

```
int

scoll_group_create (scoll_service_id_t srvid, const char *name,
                    const scoll_target_t *targets, int cnt,
                    scoll_tree_type_t tree, bool populate_now,
                    scoll_create_group_cb_t cb, const void *private);
```

This API initiates group creation process, which involves all group members to establish group states. The completion of group creation is reported asynchronously by the scoll_create_group_cb_t callback, where a group handle is returned. The group handle contains the name and private pointer given here.

Note that a target in the targets array points to all entities on that target, please see scoll_target_t.

There are two ways to create a group: immediate, or delayed, as specified by the boolean parameter 'populate_now':

- When it's delayed, the scoll_create_group_cb_t callback is called immediately from within scoll_group_create(), without actually populating the group states on all member nodes. Later when user does the 1st broadcast over this delayed group, the group creation parameters are piggybacked into the broadcast and the group states population on all member nodes happens together with the 1st broadcast request. Note that in subsequent broadcasts, group creation parameters are not piggybacked since the parameters have already been propagated during the 1st broadcast.
- When it's immediate, the server collective does a broadcast to all member nodes to have group states populated, and calls the scoll_create_group_cb_t callback only after the broadcast has completed successfully. The user can't do a broadcast over the group until the group handle is returned when the group population process completes.

In both cases, the completion is notified by the scoll_create_group_cb_t callback, so the API semantics are the same. Here's some hints about which one to use:

- Immediate group is the only choice if the group is too large to be piggybacked on the broadcast, or the broadcast is too large to have sufficient room left for piggybacking data.
- Delayed group is the choice if the user wants to send a broadcast asap, i.e. the user doesn't want to wait for the immediate group creation to complete.

Out-of-band Group

Sometimes there already exists a global ID that identifies a set of entities. For example, a Lustre FID globally identifies a set of storage objects. Such IDs can be directly used as group identifiers, where the membership has been established by a mechanism out of the control of server collective, hence the name out-of-band groups.

An OOB group can be created by API:

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

```
scoll_group_t *

scoll_oob_group_create (scoll_service_id_t srvid, const char *name,
                        scoll_oobgrp_id_t oobid,
                        scoll_tree_type_t tree, const void *private);
```

OOB group creation is a local operation, since the membership has already been established out-of-band. Therefore a group handle is returned directly from this function, rather than from an asynchronous callback as in scoll_group_create().

The advantages of OOB groups are:

- The creation is local and synchronous. No networking operation is involved.
- Group membership is not propagated over the network to all members.

But the service associated with the group must have registered the scoll_oobgrp_cb_t() callback, which is responsible for mapping OOB group ID into an array of group targets (each of which points to entities on that target).

The group handle returned can be used later in other server collective APIs. Note that there's no difference between an OOB group handle and a regular group handle.

Broadcast to a Group

Once we have a group handle, we can broadcast a request over the group identified by the handle.

Create Broadcast Request

A request is created by API:

```
scoll_request_t *

scoll_bcast_new (scoll_group_t *pub_group, const void *buf, int len,
                 const lnet_kiov_t *bulk_iovs, int niov, unsigned int flags,
                 cfs_duration_t service_timeout, const void *private);
```

The request payload is specified by buf and len, so for now it must be contiguous. An API to support fragmented payload can be added should the need arise. A request handle is returned for later use.

The request payload and bulk buffers, if any, can be freed only when the request has completed, i.e. once the scoll_post_reply_cb_t callback is called.

4.2. Prepare Reply Buffers

A request is broadcast to a group using spanning tree for scalability. Before a request can be sent, the root node must post buffers to receive replies from its child nodes.

To learn about who the child nodes are, use the following APIs:

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

- int scoll_get_nchildren (scoll_group_t *group);
- int scoll_get_childtree_nodes (scoll_group_t *group, int child);
- int scoll_get_childtree_entities (scoll_group_t *group, int child);

The total number of child nodes is returned by calling scoll_get_nchildren(). The return can be 0, i.e. if the current node is a leaf node in the group. Each child node is identified by an integer index, of range [0, scoll_get_nchildren()). This index is passed to the rest of the APIs to query about a child node:

- scoll_get_childtree_nodes() returns total number of nodes in the tree rooted at the child, including the child node itself.
- scoll_get_childtree_entities() returns the total number of entities in the tree rooted at the child, including entities on the child node itself.

With these information, a service can figure out the number of replies expected, and the max size of each. Then for each child node, a reply buffer must be posted by calling API:

```
int

scoll_reply_buffer_post (int child, const void *buf, size_t len,

                         scoll_request_t *pub_req, void *private);
```

An API to support fragmented reply buffers can be added should the need arise. A reply buffer can be freed once the corresponding scoll_incoming_reply_cb_t callback is called.

Send Broadcast

Once reply buffers have been posted, a request can be broadcast by calling API:

```
int

scoll_bcast_send (scoll_request_t *pub_req);
```

Bulk Data

To include bulk data in a broadcast:

- Root node specifies bulk data when calling scoll_bcast_new(). Bulk data is given in the form of a kernel IO vector array. If bulk data is contiguous, just pass a one-entry array. Pass NULL if there's no bulk data.
- In scoll_pre_request_cb_t() callback, a service must check the srq_bulk_len field of the incoming scoll_request_t. If it's positive, then the request carries bulk data with it. In this case, the service must prepare buffer to receive bulk data. IO vectors for the bulk buffer must be saved in the srq_bulk_iovs and srq_bulk_niov fields before the service returns from scoll_pre_request_cb_t().
- In scoll_incoming_request_cb_t() callback, a service can read bulk data, if any. Note that services can't modify bulk data, because the bulk buffer is being passed down to child nodes. If a service has to modify bulk data received, then it must make a copy first and modify the copy. Note that this is different from request

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

buffer. Service callbacks can freely modify request data, because the server collective makes a copy to be forwarded to child nodes before the buffer is passed to the service.

- In scoll_post_reply_cb_t() callback, the bulk buffer and its IO vector array, allocated in scoll_pre_request_cb_t(), can be freed. The service must not free the buffer or its IO vector array any earlier, because child nodes could still be reading from it.

Bulk data is propagated in a same spanning tree as the request data.

Request Time Out

The server collective expires requests waiting for replies, based on:

- Spanning tree depth
- Estimated network RTT
- Estimated request processing time

The estimated request processing time must be supplied by users in one of following ways:

- On the root node it's supplied as the service_timeout parameter to scoll_bcast_new().
- On non-root nodes, the scoll_pre_request_cb_t() callback can set it in the scoll_request_t::srq_service_timeout

If the request processing time is insignificant compared to the network latency, it can be set to 0, or simply ignored in the scoll_pre_request_cb_t() callback.

When a request has expired, i.e. not all replies have arrived in time, for each reply that hasn't arrived yet the request gets a scoll_incoming_reply_cb_t() callback with -ETIMEDOUT status. The scoll_post_reply_cb_t() callback will also be called. Note that in this case, the service should send a reply to parent node, because some replies may have arrived before the time out.

Service Callbacks: Request Propagation

Once root node sends a request, server collective propagates the request message over the target group organized in a spanning tree. On each node, the request is handled by the registered callback functions of the service associated with the target group.

Once a request arrives at a node, server collective invokes the registered scoll_pre_request_cb_t callback:

typedef int (*scoll_pre_request_cb_t)(scoll_request_t *request);

A collective service should do two things in this callback:

- Do sanity checks on the message and return error immediately if the message appears corrupted.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

- If the message looks good, post reply buffers for child nodes, if any. The process is the same as how the root node posts reply buffers, please see 4.2.

Once scoll_pre_request_cb_t returns successfully, the server collective proceeds with two tasks in parallel:

- Propagate the message down the spanning tree to child nodes, if any.
- Call the service callback scoll_incoming_request_cb_t().

Note that the scoll_incoming_request_cb_t() is called only once regardless of the number of entities targeted on the node. The parameter 'target' points to all entities targeted by the broadcast request.

In the scoll_incoming_request_cb_t(), a service should initiate the processing as requested in the request. Once it has processed the request, create a reply.

However, the local replies should not be forwarded back to the root node yet.

Service Callbacks: Reply Aggregation

On a leaf node, once all local replies have completed, a collective service should aggregate them into an aggregated reply, and forward it back to the parent node by:

```
int

scoll_reply (scoll_request_t *user_req, const void *buf, size_t len);
```

On an internal node, for each reply from a child node, the callback scoll_incoming_reply_cb_t() is called once:

```
typedef void (*scoll_incoming_reply_cb_t)(scoll_reply_t *reply);
```

Once all replies have been received, the server collective notifies a service by calling its callback scoll_post_reply_cb_t:

```
typedef void (*scoll_post_reply_cb_t)(scoll_request_t *request);
```

In this callback, a service should aggregate all replies and forward the aggregated reply to parent node by:

```
int

scoll_reply (scoll_request_t *user_req, const void *buf, size_t len);
```

Note that on leaf nodes, the scoll_post_reply_cb_t() is also called even though there is no child reply expected. This simplifies collective service - no matter whether a node is a leaf or not, it should always aggregate replies and call scoll_reply() in the scoll_post_reply_cb_t() callback.

Broadcast Completion

On a non-root node, a broadcast is complete when reply has been sent to parent node. This is notified by the service callback scoll_reply_out_cb_t(). Once this callback returns the request handle is no longer valid and should not be used any more.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

On the root node, a broadcast is complete when all replies have been received, i.e. in the scoll_post_reply_cb_t() callback. Once this callback returns the request handle is no longer valid and should not be used any more. This is also when the root can examine the replies for results of the broadcast.

Error Handling

Errors during the life time of a collective request are handled differently according to the time they happen.

Errors before scoll_pre_request_cb_t callback is called

Since no service callback has been called yet, the service knows nothing about the request where the error has happened. Such errors are handled completely in server collective layer, and the service will get no callback about the request.

Server collecitve reports the error back to the parent node. On the parent node, the service callback scoll_incoming_reply_cb_t will be called to inform the service about the error. From this point on, it's up to the service how to deal with the error. The server collective will not propagate the error further. The error is reported in the srp_status and srp_error_type fields of the scoll_reply_t.

Error returned from scoll_pre_request_cb_t callback

If the scoll_pre_request_cb_t callback returns error for a request, server collective handles it in the same way as in 8.1. In addition, the request will be aborted, so:

- The service will not get any more callback on the request.
- The request handle passed to scoll_pre_request_cb_t cannot be used any more.
- The request will not be forwarded to any child node.

Errors after scoll_pre_request_cb_t has returned

Such errors are reported to the service by the scoll_incoming_reply_cb_t callback. Service collective will not propagate them. It's up to the service how to deal with them.

For example, if server collective fails to propagate a request to a child node A, it calls the scoll_incoming_reply_cb_t callback (scoll_reply_t::srp_src == A) to inform upper layer service about the error. Now the service knows that reply from A will never come, but replies from other nodes may still arrive. When all other replies have arrived, the service should create an aggregated reply and send it to the parent node. The aggregated reply should include the error on A. It's up to the service how to represent and aggregate errors if there are multiple errors.

Bulk errors

If a request contains bulk data and error has happened fetching the bulk data from parent node, the error is reported to user by setting the srq_bulk_status field of the scoll_request_t to an error code and calling the scoll_incoming_request_cb_t() callback. In this case:

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

- scoll_incoming_reply_cb_t() will be called with errors, to notify that valid replies will not arrive.
- The bulk buffer returned from scoll_pre_request_cb_t() callback does not contain valid data.
- Server collective reports the error to parent node, so user services should not call scoll_reply().

Abort Requests

When a service wishes to stop, the first thing to do is to make sure no more incoming requests will be accepted, by simply returning an error from the scoll_pre_request_cb_t callback. Then server collective drops all new requests and reports error back to parent nodes.

If there's still active requests, they can be aborted by calling:

void scoll_bcast_abort (scoll_request_t *pub_req);

The service must wait for all pending callbacks before buffers associated with the aborted request can be freed.

The Protocol

A typical server collective communication is executed in a server group in the following steps:

1. Server service on root node explicitly creates a group over all members. Or alternatively, group membership may have been established from an out-of-band mechanism, in which case the group ID is determined by the out-of-band mechanism. Spanning tree topology is determined by server collective module on root node, and propagated to member nodes.
2. Server service on the root node broadcasts a message to a server group, specified as a group ID.
    a. The server collective module finds out group membership by looking up the group ID in its group table, and builds a spanning tree over all group member nodes with itself being the root node of the tree. The mapping from group ID to group members must have been established either explicitly or out-of-band, i.e. outside the control of the server collective module.
    b. The message is forwarded down the tree, together with a collective header that specifies group ID and service ID among other things.
3. When receiving a broadcast message, a member node:
    a. Finds out group membership from the group ID included in message header. The member node may not need to build a complete spanning tree from group membership. It'd suffice to know which nodes are its immediate children. Also, perform sanity checks on message header, e.g. service ID must have been registered locally.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

b. Server collective module calls the *pre_request_handler()* callback function registered by server service as identified by service ID in the message header. Typically in this callback function, server services should:

   i. Perform sanity checks on the message.

   ii. Perform any required pre-processing before the message can be forwarded to the child nodes. For example, read bulk data from parent node and prepare RMA descriptor to be forwarded to the children.

   iii. Decide how to aggregate replies from child nodes, and prepare reply buffers.

c. Once *pre_request_handler()* has returned, and server collective module can proceed with the following two steps in parallel:

   i. Forward the broadcast message to child nodes. Note that *pre_request_handler()* may have modified the message, e.g. to include a new RMA descriptor.

   ii. For each member entity on this node, call *incoming_request_handler()* of the corresponding service to execute the request on this member entity.

4. Upper layer services on all member nodes reply by calling *scoll_reply*(). Replies are aggregated and forwarded up the tree. On each node when a reply message is received, server collective module calls the *incoming_reply_handler()* callback registered by server service. In this reply handler callback, server service could:

   a. Aggregate the current reply with the ones already received.

   Once all replies are received on a node, i.e. all members that belong to the subtree rooted at the current node have replied, the server collective module calls the *post_reply_handler()* of the server service to signal that all replies have now arrived. In this call back, server service could:

   a. Perform any required post-processing before the reply can be sent back to the parent node.

   b. Return an aggregated reply message to the server collective module. The server collective module then forwards the aggregated reply to the parent node.

   A server service may choose to process replies one by one as they arrive in the *incoming_reply_handler()*, or process them all in one go in the *post_reply_handler().* It's up to the service, but the *incoming_reply_handler()* may allow more overlap between communication and computation.

5. Root node receives aggregated replies and delivers it to upper layer by the *post_reply_handler()* callback. The callback invocation marks the completion of the collective communication.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

6. Group states on member nodes are destroyed when the root node destroys the group or when the root node has died or rebooted.

Note:

- Steps 2 and 3 are separate steps here only for clarity. In implementation, they can be combined in the request processing – there's no need to distinguish root node from other member nodes. Similarly, steps 4 and 5 should be combined in reply processing in the implementation.
- A node can belong to multiple groups, where it may or may not be the root.
- A group ID only maps to a set of members, and does not tell where the root node is. The group ID together with a root node determines a spanning tree, so there could be multiple trees over a same group. This can be useful when more than one servers in a group wish to broadcast messages.
- Only the root can broadcast message down the tree.
- Message is delivered to upper layer on all member nodes, i.e. there's no dedicated forwarding node that only forwards messages and has no upper layer services running on it.
- A member node must aggregate replies from upper layer service on itself together with replies from its immediate children.

Populate Group Membership

Collectives are executed on all members of a group, specified by a unique group ID. Groups consist of an ordered list of members and all members must agree on group membership and order, and on an algorithm that generates spanning trees for the group rooted in any arbitrary member. Any member can therefore determine its children and parent given the root node index.

The group ID consists of a unique index and a cryptographic checksum computed over the group members to ensure that the same group ID cannot be used for groups with different memberships. Every node in a server cluster maintains a group table indexed by group ID. Entries in this table may be created in either of the following two ways:

- Implicit group join through out-of-band mechanisms, i.e. external to the server collectives module.
- Explicit group creation collective to populate group membership information from the root node to all member nodes.

Group Join Out-of-band

Often the server services already know about the group they belong to, through some out-of-band mechanism, i.e. out of the control of the server collective module. For example, servers already know about all the storage targets for a certain Lustre FID, so the FID can serve as a part of out-of-band group ID. Since such groups are populated out-of-band, their membership can also change out-of-band. Messages over out-of-band groups should all carry a digest of the membership computed at the root when the message is created, in order to detect membership changes.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

Out-of-band groups are created when a message to the group is first sent (on the root node) or received (on any member node):

1. Server services register with the collective module by calling *scoll_service_register()*, and supply a callback function *oob_group_handler()*. This callback function returns a list of group members from an out-of-band group ID.
2. When a server wishes to broadcast a message over an out-of-band group, it calls *scoll_bcast(group_id, request, flags)*, and it'd become the root node of the spanning tree that covers the group. An out-of-band group ID consists of two parts: service ID, and service group ID
3. In *scoll_bcast*(), the collective module checks whether it knows about group_id or not. If group_id is unknown, i.e. the group has not been created yet, it calls the out-of-band group callback routine *oob_group_handler()* of the service whose ID can be extracted from out-of-band group ID, registered in step 1. The callback returns a list of members, and then the collective module creates the group states and builds a spanning tree covering the member nodes. A digest of group membership is computed and included in the message header. Then the message is broadcasted down the tree.
4. When a message for an out-of-band group is received, the collective module calls the *oob_group_handler()* routine to get group membership, in a same way as step 3. Then it checks group membership against the group digest in the message header. If the digest doesn't match, the group membership must have changed and the message is dropped, i.e. not forwarded down the tree. At last the message is forwarded down the tree. Therefore, group states for an out-of-band group are created as the first broadcast message are sent on the root and forwarded down by members.

Note that:

- A server service registers *oob_group_handler()* callback only if it supports out-of-band groups.
- Out-of-band group ID consists of the service ID so that member nodes can figure out which *oob_group_handler()* to call among all registered services. The rest of the out-of-band group ID is opaque to the server collective module and meaningful only to the corresponding *oob_group_handler()*.
- An out-of-band group ID only determines the membership, i.e. an array of (tag, NID) for all members, but not the spanning tree. The tree is determined by group ID, root node, and a well-known algorithm to build the tree. Any node in an out-of-band group can become the root node of a spanning tree if it wishes to send a message to the group.
- Although server collectives can detect membership changes of out-of-band groups by using digests, services should try to make sure that the mapping from out-of-band group ID to member list doesn't change when there's ongoing broadcast over the group. Otherwise network resources will be wasted as the broadcast would fail in the middle of mapping changes.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

Explicit Group Creation

When only the root node knows about all the members initially, it needs to propagate this information to all members before it can obtain a group ID and broadcast any message over it.

The root node of a group initiates the creation of the group. Group creation is a collective process where all group members participate. The ID of the group consists of a digest of the ordered list of group members.

The explicit group creation is essentially a broadcast message that contains group membership information in its payload. Member nodes look into the message for list of group members, in contrast with out-of-band group join, where member nodes rely on server service callback to find out this information by an out-of-band mechanism.

The API to create group explicitly is *scoll_group_create(members, completion_callback())*. The *completion_callback()* is invoked when the root node has received aggregated replies from all its children. The ID of the group is returned in this callback.

Build New Group from Existing Groups

After some groups have been created, new groups can be built from operations on existing groups. For example, a server may want to broadcast to the intersection of two existing groups.

At this point, we see group subset to be most useful group operation. In the future when need arises, more group operations can be added, e.g. group intersection.

Group Subset

After a group has been created, sometimes a server would want to broadcast over only a subset of this group. For example, a server may want to broadcast to all OSS servers which are still alive, i.e. the subset of live nodes among all known OSS servers.

Group subset is specified by the root when it is broadcasting over the sub-group. The specification consists of the ID of the parent group, and a subset specification. Currently we plan to support only one type of subset specification: all nodes that are currently alive according to the server discovery module, in the form of a digest that all nodes must agree upon.

In the future, if necessary, more types of subset specification may be added, e.g. all odd or even members.

Build Spanning Trees

Spanning trees are used to propagate messages between nodes. Therefore, in the discussion of spanning trees, member entities on a same server node are irrelevant – spanning trees cover member nodes, not the member entities. In this section, member refers to a member node, and group refers to the group of nodes that a collective group of member entities belong to.

Group membership can be represented by a simple NID array: lnet_nid_t members[N]. A node's rank within the group is said to be its index into the members[N] array.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

As described earlier, the members[N] array can be established by an out-of-band mechanism or by an explicit group creation process. Once all member nodes have agreed upon the group membership, they can participant in group broadcasts, i.e. forwarding broadcast messages downstream to immediate children and aggregating replies from all descendants and sending aggregated copies upstream to parents.

With a copy of the members[N] array and rank of root node and a well-known algorithm to build a spanning tree, there's enough information on every node to build a complete tree that covers the whole group, but it is often not necessary to do so.

There's no single tree shape that is universally optimal for all network configurations and different server services, and the optimal choice can be platform dependent. Therefore the building of spanning tree should be made a pluggable function – it should be made as simple as implementing a few new functions in order to add a new spanning tree topology. Initially we plan to implement k-nomial tree and k-ary tree.

Binomial Tree

Binomial tree is commonly used in MPI collective implementations for its scalability at broadcasting small messages. Here's an example of broadcasting a message over a 4-degree binomial tree of 16 nodes:

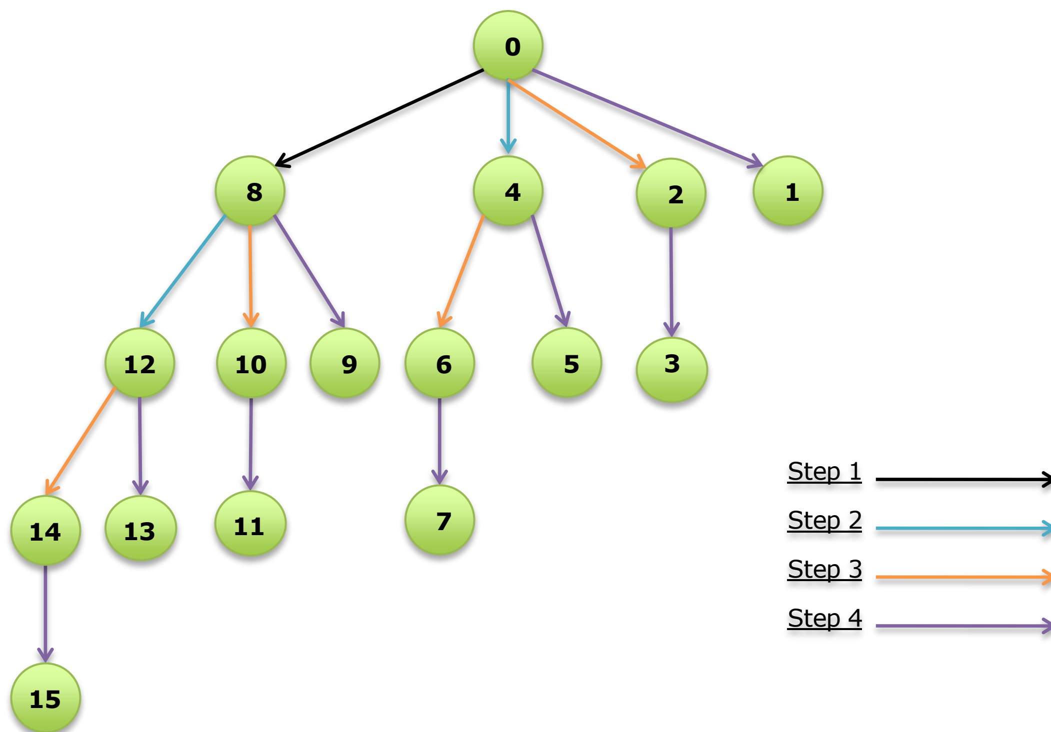


Figure 1: Binomial Tree

In each step of the broadcast, all nodes that currently have a copy of the message forward it to one of their children:

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

1. Node 0 forwards the message to node 8, since node 0 is the originator of the broadcast (also root the spanning tree), i.e. the only one who has a copy of the message at the beginning. Now 2 nodes have copies of the message.

2. Nodes 0 and 8 forward to nodes 4 and 12 respectively. Now 4 nodes have copies of the message.

3. Nodes 0, 8, 4, and 12 forward to nodes 2, 10, 6, and 14 respectively. Now 8 nodes have copies of the message.

4. Nodes 0, 8, 4, 12, 2, 10, 6, and 14 forward to nodes 1, 9, 5, 13, 3, 11, 7, and 15 respectively. Now all the 16 nodes have copies of the message, and the broadcast completes. Note that all leaf nodes receive a copy of the message at the final step. Now node 0 the root has received replies from all its descendants and the reply aggregation completes.

Aggregation of replies happens in the reverse order of the broadcast. In each step, all nodes that have received replies from all its descendants forward an aggregated copy of the replies to their immediate parents:

1. Nodes 1, 3, 5, 7, 9, 11, 13, and 15 send their replies to nodes 0, 2, 4, 6, 8, 10, 12, 14 respectively, i.e. all leaf nodes send replies to their parents.

2. Nodes 2, 6, 10, and 14 aggregate their replies with replies from descendants and send the aggregated replies to nodes 0, 4, 8, and 12 respectively.

3. Nodes 4 and 12 aggregate their replies with replies from descendants and send the aggregated replies to nodes 0 and 8 respectively.

4. Node 8 aggregates its own reply with replies from its descendants and sends the aggregated reply to node 0.

Tree Representation

The tree can be completely represented by the members[N] array, with tree topology encoded into the ranks in the following way:

1. My rank is R.

2. Parent: My parent's rank P equals my rank R with the least significant 1 bit cleared. If P equals R, then I'm the root. For example, the parent of node 8 (binary 1000) is node 0 (binary 0000, i.e. least significant 1 bit of 1000 cleared); the parent of node 15 (binary 1111) is node 14 (binary 1110, i.e. least significant 1 bit of 1111 cleared).

3. Sub-tree rooted at me: the degree K of the sub-tree rooted at myself is log2(R-P), and the size of the sub-tree is 2**K=R-P. For example, the sub-tree rooted at node 12 has degree log2(12-8)=2, and size 2**2=12-8=4. The sub-tree rooted at node 7 has degree log2(7-6)=0 and size 2**0=7-6=1, thus node 7 is a leaf node – all nodes with odd ranks are leaf nodes, and vice versa. Root node needs special handling as there's no P.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

4. Immediate children: each node has K children, with ranks R+2**e where e is from 0 to K-1. For example, node 12 has degree K=log2(12-8)=2, so it has 2 children 13 and 14, i.e. 12+2**0 and 12+2**1.

Partial Tree

When N is not a power of two, there are not enough nodes to fill up a complete binomial tree. In such cases a partial tree is built from a full tree with nodes that don't exist removed. Here's an example of a partial 4-degree tree of 14 nodes:

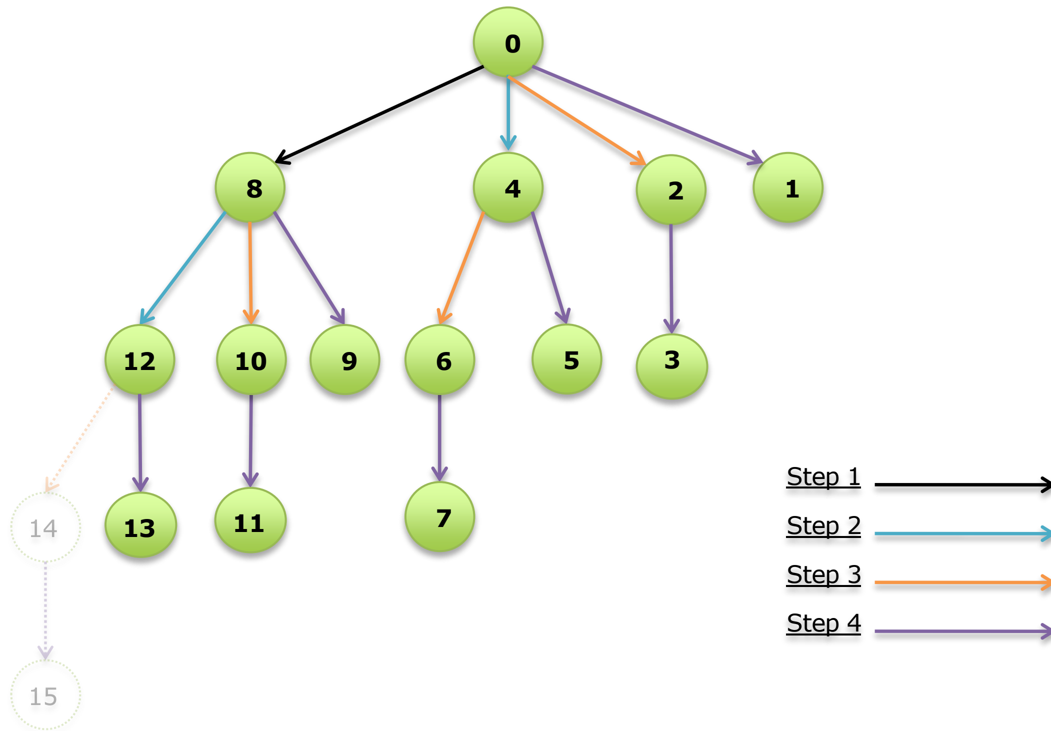


Figure 2: Partial Binomial Tree

As shown in the graph above, nodes with ranks 14 and 15 are removed, shown in dashed lines, because there are only 14 entries in the members array. The topology algorithms need to be amended to take into account partial trees:

1. Sub-tree rooted at me: let D=R+(R-P)-1=2R-P-1, i.e. the biggest rank among my descendants if N were power of two. If D<N, then my sub-tree size is still R-P, i.e. the sub-tree is full; otherwise the sub-tree size is R-P-(D-N+1)=N-R, i.e. with D-N+1 nodes excluded from an otherwise full sub-tree. For example, the sub-tree at node 8 has size N-R=14-8=6, because D (15) >= N (14).

2. Immediate children: nodes of ranks R+2**e where e is from 0 to K-1 **AND** R+2**e < N. For example, node 12 has children 13 only, and 14 is excluded because 14 is NOT less than the total number of nodes.

Note that node 12 can skip step 3 and go directly from step 2 to step 4, since there's nothing to do for step 3.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

K-nomial Tree

Let **o** be the time to inject a message into the network, i.e. the time between successive sends on a node, let **l** be the time for the network to deliver a message from sender to receiver, and **k** the degree of the tree (i.e. k = log(N)). From Figure 1 we can see that in a binomial tree, the time for a message to reach a leaf node on level **x** (root has level 0, the leftmost leaf has level k) is x*l+(k-x)*o = k*o + (l-o)*x. Therefore:

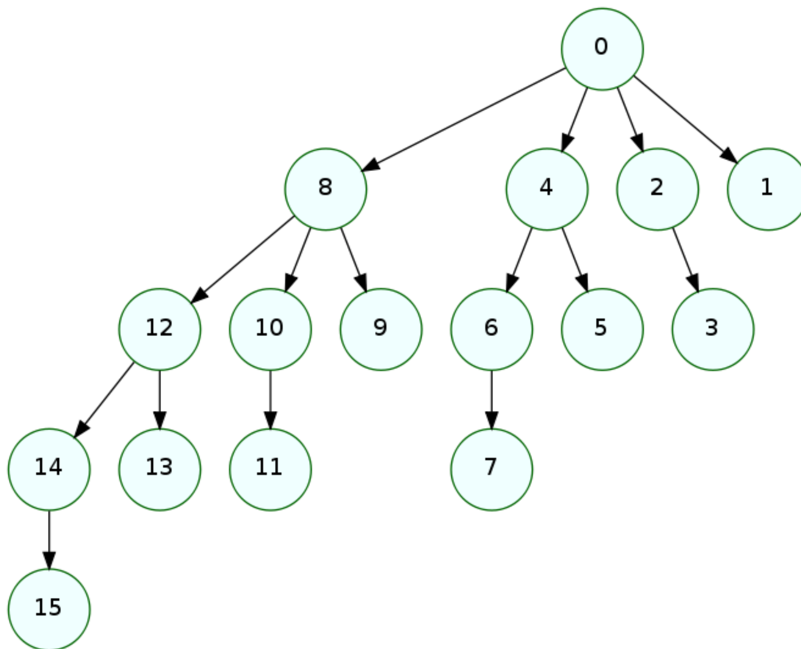
1. When l equals o, the message reaches all leaf nodes at a same time, i.e. k*l.
2. When l is less than o, the message reaches leaf nodes from the left to the right, i.e. deepest leaf gets the message the first.
3. When l is greater than o, the message reaches leaf nodes from the right to the left, i.e. deepest leaf gets the message the last.

The case 1 is where binomial tree works best – all nodes who have received the message keep forwarding it until it hits all leaf nodes at a same time, thus no one is idle while waiting for others to propagate the message. This is the most ideal situation for binomial tree.

Case 2 may happen when there's additional gap between successive sends, e.g. back pressure from congestion control or another application completing for outgoing bandwidth. We don't consider it to be a valid case for us as we give broadcast messages high priority, e.g. by using priority queue and pre-allocation of resources, and when there's congestion control back pressure the latency **l** should increase too due to the congestion.

In case 3, the root becomes idle while nodes on the left-side sub-trees are still propagating the message, and as a consequence the latency for the broadcast to complete becomes higher. Therefore we want a tree that is shallower and wider than the binomial tree, yet it should have the unbalanced structure like the binomial tree to keep all nodes busy. This is exactly a generic k-nomial tree with k greater than two.

Here's a comparison of a binomial tree and a quadnomial tree, both of 16 nodes:



The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

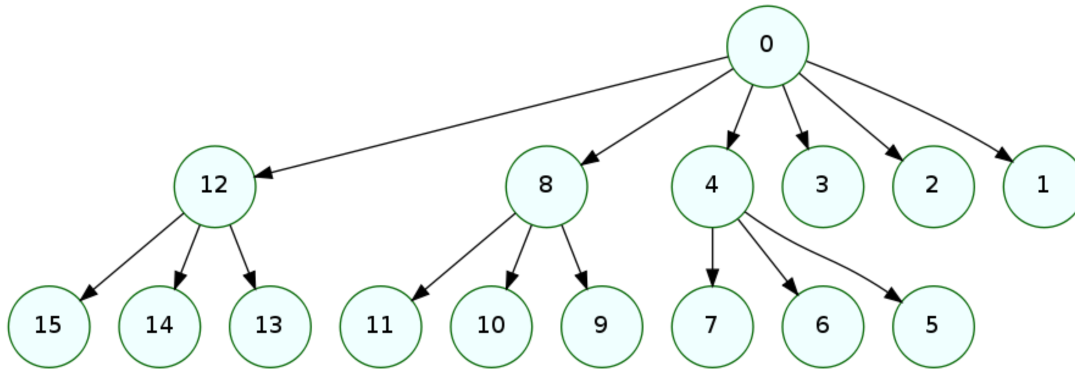


Figure 3: Binomial vs Quadnomial

In the quadnomial tree, there are two less levels and the root node has two more children.

Tree Representation

This is actually a generalised version of the binomial tree algorithm.

The tree can be completely represented by the members[N] array, with tree topology encoded into the ranks in the following way:

1. My rank is R.

2. Parent: My parent's rank P equals my rank R with the least significant non-zero bit cleared from the base-K representation of R. If P equals R, then I'm the root. For example, node 14's rank in base-4 is 32, with the 1st bit cleared we get 30 which is 12 in base 10, so its parent is node 12; node 12's rank in base-4 is 30, with 2nd bit cleared we get 0 which is the parent's rank.

3. Sub-tree rooted at me: the degree D of the sub-tree rooted at myself is the integer floor of logK(R-P), and the size of the sub-tree is K**D. For example, the sub-tree rooted at node 12 has degree log4(12-0)=1, and size 4**1=4. The sub-tree rooted at node 7 has degree log4(7-4)=0 and size 4**0=1, thus node 7 is a leaf node. Root node needs special handling as there's no P.

4. Immediate children: each node has D*(K-1) children, with ranks R+K**e * p where e is from 0 to D - 1 and p is from 1 to K-1. For example, node 12 has degree D=log4(12-0)=1, so it has D*(K-1)=1*(4-1)=3 children 13 and 14 and 15, i.e. 12 + 4**0 * 1, 12 + 4**0 * 2, and 12 + 4**0 * 3.

Complete K-ary Tree

Compared with K-nomial trees, the main drawback of K-ary trees is that once a parent sends the data to the children, it is idle and has to wait until the data propagates to the leaf node. However, in a K-ary tree the load of broadcasting messages is evenly distributed – all nodes but the leaves send exactly K times, while in a K-nomial tree the root always has most immediate children and consequently most work to do. Therefore the K-ary tree can be a better choice for server services that are not sensitive to latency, for example a service which initiates a broadcast and then goes on doing other things without blocking for completion.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

Once the members of a group are known, a complete K-ary tree can be built by following a set of simple rules:

1. Each node in the tree is populated with K immediate children as long as there's enough nodes left in the group
2. Nodes in level L are populated before any node in level L+1 or higher are populated, i.e. the tree depth is kept at the minimum by populating nodes one level at a time.
3. On a same level, nodes to the left are populated first, for maximum space efficiency.

An example of a complete 3-ary tree of 12 nodes:

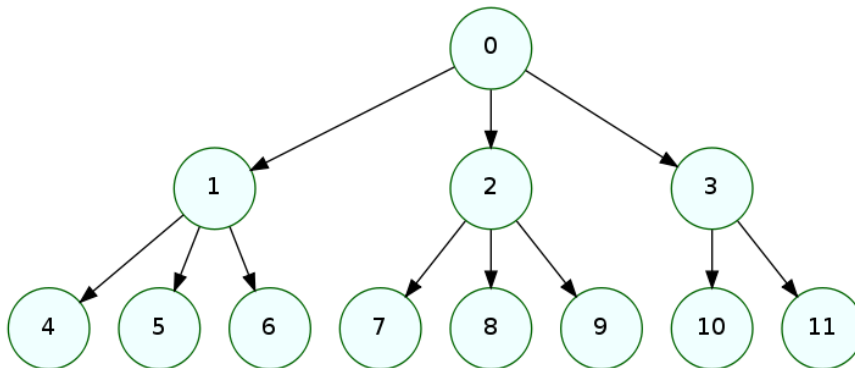


Figure 4: Complete 3-ary Tree

Note that node 3 has only two children since there're not enough nodes in the group to keep the tree full.

Tree Representation

There's no need to send a tree structure to all nodes from the root. Instead, the K-ary spanning tree can be represented by a simple array: lnet_nid_t members[N], and the branching factor K.

When a group is created, the root node builds this array with itself being members[0] – the index in the array is said to be the rank of the corresponding node in the tree, so root is always rank 0. Then the root node sends this array and the branching factor to all member nodes. Any member node can calculate information about topology by the following formulas, with R being the rank of the node itself:

- Parent rank: (R – 1) / K
- Children ranks: C[K] = R x K + [1, K], as long as the resulting rank is less than N.
- Size of sub-tree rooted at R, by the recursive tree_size() routine: tree_size(R) = 1 + tree_size(C[0]) + tree_size(C[1]) + … tree_size(C[K - 1]).

It's possible to build a member array such that the corresponding tree topology matches the network topology, e.g. nodes in a same cabinet are directly connected in the tree. All

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

such topology information is known to DAOS services – it's to be determined how this information is passed down to the server collective module, and how to build better trees based on the information.

Choose Spanning Tree

The choice of what spanning tree (i.e. both type and options, e.g. K-ary tree with K=4) to use must be made by the administrator, via a kernel module option, or via a parameter to the server collective initialization routine should the need arises for server services to use different spanning trees.

The choice is made manually based on characteristics of the network and the requirements of server services. Automatic tuning of the algorithm is out of scope for this project. However, the server collective module should support benchmarking of collective communications, e.g. measuring the latency from when the root begins sending to the moment the last leaf node receives data.

Broadcast Message

First, the root node checks with the server discovery protocol to see whether all member nodes are alive. If not, fail immediately; otherwise it begins to propagate the message to its children.

All nodes enter the following broadcast procedure once they have received a copy of the broadcast message from the parent, or on the root node from upper layer services:

1. Build group state if necessary. For example, for out-of-band groups there's no explicit group creation – groups are created upon receiving the first broadcast message.

2. Perform sanity checks on the received message, e.g. it should come from my parent node and service ID for the group must map to a registered service.

3. Call the *pre_request_handler()* callback for any preprocessing necessary before the message can be forwarded down the tree. In this callback, upper layer needs to:

    a. Prepare reply buffers to receive aggregated replies from child nodes.

    b. If there's RMA descriptor of bulk data in the broadcast message, fetches the bulk data, prepare RMA descriptor for local copy of the bulk data, and returns a new message that contains the RMA descriptor of local bulk data copy to be forwarded to the children.

4. Once the *pre_request_handler()* callback returns, server collective module proceeds with the following two tasks in parallel:

    a. For each immediate child node of rank C, ordered from the highest rank to the lowest: send node C a copy of the broadcast message or the message returned by upper layer callback in step 3. Note that the message should be sent to children in the order of sub-tree depth rooted at the children, i.e. the deeper the sub-tree the sooner the child should be sent the message. For both k-ary trees and k-nomial trees, our tree-building algorithm guarantees that a child of higher rank has a deeper sub-tree than a child of lower rank. Therefore for k-ary trees

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

and k-nomial trees the message should be sent to children in the order of decreasing ranks.

b. For each member entity on this node, call *incoming_request_handler()* of the corresponding service to execute the request on this member entity. If the handler returns successfully, the service must call a scoll_reply() to generate a response. The service may call scoll_reply() before returning from this handler or later from a service thread when the requested operation completes. Note that the handler can return failure, e.g. when the request message is invalid, in which case scoll_reply() will not be called from the server service. When there are multiple members on a same node, the *incoming_request_handler()* invocations may be issued in parallel so the handler must be re-entrant.

5. Wait for all children to send replies.
6. Once all replies have been received, aggregate them together with the replies from upper layer service on myself and forward it up to my parent P. In the case of the root node, deliver the aggregated reply to upper layer service which started the broadcast.
7. Broadcast procedure completes. If this is the final broadcast for the group, as indicated by the last-broadcast flag, then destroy the group state and reclaim all resources.

Reliable delivery

Each member node sends collective messages by directly calling the LNetPut API. LNet point-to-point messages are reliable since routers are excluded from the path – protocols in LNDs and layers beneath LNDs already implement time out and retry. There's no need to implement another layer of acknowledgements and retries in the server collective module. In the case where the target node has died or rebooted after a message has been delivered, the server discovery module will notify us.

However, the LNet API doesn't notify callers on delivery:

- When LNet notifies that a message has been sent successfully, we can rely on LNDs for reliable delivery.
- When LNet indicates a failed sent, we might still retry the message as the failure is local.

With acknowledgements removed from server collective protocol, there's only one pass of upstream messages for each broadcast, i.e. the aggregation of replies from leaf nodes up to the root. The aggregated reply tells the root node whether the broadcast message was delivered successfully to all member nodes. With a separate pass of acknowledgement aggregation, the root node may know a bit earlier whether the broadcast has been delivered to all nodes, but there's no strong need for that earlier delivery notification from upper layer services and the overhead of a separate pass of acknowledgements up the tree can be eliminated.

Notification of delivery failure

A point-to-point message delivery can be considered as failed when:

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

1. There's repeated local LNet failures, i.e. LNet events indicating failures to send.
2. The server discovery module notifies that a node has died or rebooted, after LNet successfully sends out a message.
3. If it's a downstream broadcast message, a reply hasn't come back after a static timeout and the above two cases haven't happened. This is just a safeguard against bugs in the server discovery module, which is supposed to reliably detect node failures. More details on reply timeout in the coming section on reply aggregation.

Upon such failures, member nodes:

- Simply give up if it's a reply message going upstream.
- Create an empty reply message which indicates that the sub-tree rooted at the target node hasn't received the broadcast message, and forward the reply message up the spanning tree.

Reply Timeout

In the case where a broadcast message has been forwarded successfully to a child and the server discovery module hasn't detected any status change of the child node, the member node still can't wait indefinitely for the child to reply. The reply timeout, however, must take into consideration both the network RTT and the processing time for upper layer services to respond.

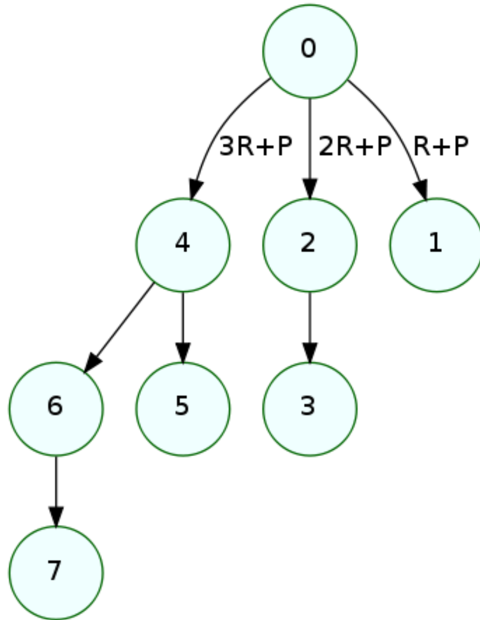
Different broadcast messages require vastly different processing time – some may involve disk IO, e.g. commits, and some may need only manipulation of in-memory states. When upper layer service at the root node initiates a broadcast, it must specify an estimate of upper bound for processing time for recipients to process and respond to the message. This estimate is carried in the header of the broadcast message, so that nodes down the tree can calculate a proper timeout to wait for replies.

Note that upper layer services should be conservative or pessimistic about the upper bound estimate of processing time, because the estimate is used to timeout waiting for reply only for the unlikely case that the server discovery module has failed to detect a status change of the node.

Cascading Network Timeout

Member nodes should use different timeout values to wait for aggregated replies from their immediate children. This is determined by sub-tree depth rooted at the child node, because it takes an extra Round Trip Time for messages to reach each additional level of nodes. For example, in the binomial tree shown below, with Round Trip Time estimated to be R and processing time estimated to be P:

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.



The root node 0 should wait for aggregated reply from child node 4 with timeout 3R+P, child node 2 with 2R+P, and child node 1 with R+P, since sub-tree depths at the child nodes are different. In contract, with K-ary trees, the sub-tree depths are more likely the same and hence the timeouts could be the same for all children nodes.

Note that P, the estimate of processing time, should not be cascaded like R, the network RTT, because processing on nodes in a sub-tree overlap with each other.

Reply Aggregation

Replies can be aggregated in different ways, e.g.:

- Simple concatenation. But meta-data needs to be added to concatenated message so as to split the concatenated reply at the root.
- Status code aggregation: e.g. only node 1 to 12 failed – root only sees NIDs of those who failed, so aggregated reply is often small. This should work for most cases. Also, upper layers may not be interested in error codes – it may suffice to know who has failed to execute its command.

Aggregation should be done by upper layer services, because the server collective module doesn't understand upper layer protocols and thus can't aggregate any better than simple concatenation of reply messages. Reply aggregation happens in the following steps:

1. Upper layer service registers callback functions with the server collective module for reply processing, named *incoming_reply_handler()* and *post_reply_handler()*, during initialization.
2. Server collective module calls *incoming_reply_handler()* for each reply message it receives.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

3. The first time *incoming_reply_handler()* is called for a broadcast, the function could estimate size of the aggregated reply and allocate all resources for the aggregation, based on spanning tree topology and the nature of the broadcast.

4. Each time *incoming_reply_handler()* is called, it may either aggregate the reply message incrementally, or wait for the final *post_reply_handler()* invocation to aggregate them all in one go. Incremental aggregation has the advantage that the aggregation overlaps with communication. However, it may not be feasible for some broadcasts. It's totally up to the upper layer service how to best aggregate replies since upper layer has sufficient knowledge to make the most optimal decision.

5. Once all *incoming_reply_handler()* invocations have returned, the service collective module calls the *post_reply_handler()* which returns the aggregated reply buffer or buffers to server collectives layer, which then sends it back to the parent node.

Note that:

- A service may register a NULL *incoming_reply_handler()* if it doesn't need any per-reply processing.

- The *incoming_reply_handler()* must be re-entrant. It can also race with the *incoming_request_handler()*.

- The size of an aggregated reply can exceed LNET_MAX_PAYLOAD, in which case the reply can't be aggregated further with other replies:

    - If aggregated replies can't fit into one LNet message of LNET_MAX_PAYLOAD bytes, they are aggregated into as few LNet messages as possible.

    - When multiple reply messages result from reply aggregation, the messages can be sent to parent node as soon as they become ready – there's no need to wait for the final aggregated message and send them all in one go.

Buffer Management

Downstream broadcast messages are unsolicited. Member nodes can't predict when they are coming, where they are coming from, how many are coming, or what their sizes are. Therefore:

- Broadcast messages are sent to a lazy portal so that the messages don't get dropped if the target node doesn't have sufficient buffers to receive them on their arrival.
- LNET_NID_ANY is used to match incoming messages from any node, and all match bits are ignored.
- Member nodes post buffers of size 4K bytes to this portal, closely monitor its usage, and post more when it's running low on available buffers.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

Note that the server collective module requires that broadcast message can't exceed 4K bytes in size. Please see Bulk Data for reasons behind the size limit and how bulk data are propagated.

Upstream replies can be expected in terms of time, source address, size, and quantity. Therefore:

1. Before forwarding a broadcast message to child nodes, a member node must calculate the sizes of the aggregated replies to be expected from each child, and allocate and post reply buffers:

    a. Calculate the sub-tree size rooted at each child node, e.g. 1 if the child is a leaf node.

    b. Call the server service *pre_request_handler()* callback function to estimate the maximum sizes of the aggregated reply from the children, based sub-tree size and information in the broadcast message. The callback also allocates and returns reply buffers to the server collective module.

2. Member node posts a reply buffer for each child. The buffer can only match reply message for the current broadcast from the particular child node. Then the broadcast message is forwarded to the child. Note that:

    a. If estimated size of the aggregated reply exceeds the LNET_MAX_PAYLOAD bytes, then multiple buffers are posted as the aggregated reply would come in several parts.

    b. As a member node can't begin forwarding a broadcast to a child until it has posted reply buffers to receive the aggregated reply from the child, it's important to minimize the time for posting reply buffers in order to reduce the latency between successive sends. There are several ways to minimize the latency:

        i. Sub-tree sizes don't change. Calculate once only and save them.

        ii. Pre-allocate reply buffers of several fixed sizes. Buffers don't have to be of exactly the same size as the replies they match.

        iii. Make use of LNet SMP CPU affinity framework to make efficient use of parallelism in host processing, e.g. by minimizing lock contention.

3. Aggregated reply is sent to a non-lazy portal on the parent node using unique match bits that match exactly the reply buffer posted by the parent node for this aggregated reply.

Bulk Data

The collective broadcast message is the only unsolicited messages used in the protocol and servers are required to buffer these eagerly and losslessly. For this reason, requests are constrained to be small such that the maximum buffering requirements are a small fraction of server memory.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

When a server needs to broadcast bulk data, it should broadcast the RMA descriptor of the bulk data instead. Child nodes will fetch the bulk data using the RMA descriptor, and forwards the bulk data down the tree by forwarding RMA descriptor of its own copy of the bulk data.

Note that a member node has to change the broadcast message before it's forwarded down the tree, because the RMA descriptor in the message must be changed to be the descriptor of its local copy of the bulk data.

Group Lifespan

Group states can't persist on member nodes forever. It must be explicitly destroyed by root node or implicitly recycled, in order to reclaim the resources.

Root node can explicitly destroy a group:

1. When it is broadcasting the last message over the group, a flag is set in the header of the broadcast message to indicate that there'd be no more messages for this group. This is called the last-broadcast flag. Member nodes can then destroy the group and free all resources once the broadcast completes, i.e. once the aggregated reply has been delivered to the parent node.
2. When it decides that there'd be no more broadcasts over the group, e.g. when server service is quitting execution, the root node sends a notification to all group members to destroy the group. The notification is essentially a zero-sized broadcast message over the group with the last-broadcast flag set. It's very similar to case 1 above, with the only difference that the last-broadcast flag is not piggybacked on the final broadcast message.

Member nodes can also destroy a group by themselves without any explicit notification from the root node:

- When it notices that the root node is dead. The server discovery module can notify the collective module when a node of interest is dead. All the trees rooted at the dead node are destroyed. Note that when a member node is dead, it's up to the root what to do.
- When it notices that the root node has rebooted without destroying its groups, e.g. by power cycling the root node. All groups created by the previous incarnation of the node must be destroyed, but not those created by the current incarnation.

Design Notes

A couple of design choices require that the server discovery module should be able to detect server reboots reliably. Member nodes need to be notified about server reboots in order to:

- Stop waiting for reply from a server that has rebooted.
- Destroy all groups rooted at a node that has just rebooted. To be exact, only the groups created by previous incarnations of a node should be destroyed.

The current server collective design and implementation can't guarantee this if a server node reboots itself very quickly. An amendment to the design is necessary to reliably detect quick reboots:

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

1. Upon startup the server discovery module blocks for a few cycles, during which it's said to be in stealth mode. In stealth mode:
    a. A node doesn't send out any gossip messages, so that others would believe that the node is dead.
    b. A node can still process incoming gossip messages, e.g. to synchronize global gossip cycle number and update aliveness timestamps.
2. The stealth mode only needs to last long enough so that all others would consider the node dead. The length of the stealth mode is determined by size of the gossip group and it grows logarithmically with group size.
3. Stealth mode can be ended earlier if an incoming gossip message contains a very old timestamp of the node, which indicates that all others have already considered the node to be dead, e.g. the node is not rebooting quickly at all.
4. All nodes keep an aliveness counter for everyone else, without propagating this counter in the gossip protocol. This counter is incremented by one each time a node goes from alive to dead.

Now the server collectives module would know that a node has rebooted if the local aliveness counter for the node has changed.

This mechanism essentially forces a node to pretend to be dead (and make sure all else agree on it) before it begins tell everyone else that it has just become alive. All states created in the previous incarnation would have been cleaned up before the current incarnation begins to create new states. For example, a node that just quickly rebooted itself blocks in stealth mode, during which server collectives can't do any work. By the time the blocking ends, all other nodes must have agreed that this node is dead and have destroyed all groups that belong to its previous incarnation. Therefore, groups created by the new incarnation wouldn't be destroyed by mistake.

In addition, this mechanism doesn't add to protocol overhead of the server discovery module, because the aliveness counter is not exchanged by the gossip protocol.

API and Protocol Additions and Changes

Open Issues

In all spanning trees except flat trees, the root node only directly communicates with its immediate children which don't include everyone else. This brings up a potential problem that upper layer services don't have a change to negotiate protocol versions or capabilities among group members, e.g. by exchanging connection flags like what the PTLRPC does.

Upper layer services should include a version number in their protocols, and choose one of the following strategies to deal with multiple versions:

- Simply fail and abort the broadcast when there's a version mismatch. It's mandatory that all servers run a same version of the protocol.
- When receiving a broadcast message with a higher version number, a member node simply reply with an error code and its own version number.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.

However, the broadcast message should still be forwarded down the tree to all children. Later when aggregated reply reaches the root node, it knows about the versions of those nodes who has failed for version mismatch and can try again with the lowest version of the protocol found in the aggregated reply.

In other words, upper layer services should be able to deal with this problem on their own.

Risks & Unknowns

Large-scale test resources seem scarce and difficult to get hold of.

The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2014, Intel Corporation.