Date:
December 13, 2012

SOLUTION ARCHITECTURE – SERVER COLLECTIVES

FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST

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

Table of Contents

Revision History

Date	Revision	Author
2012-12-13	1.0 Draft for Review	Isaac Huang, Intel Corporation
2013-01-03	1.01 Updates to include optional requirements	Isaac Huang
2013-01-09	1.02 Clean up formatting	Isaac Huang

Introduction

A cluster of servers should act as a coherent unit rather than a group of separate entities. For example, clients should be able to establish connection to a group of servers by connecting only once, rather than having to connect to each server in the group separately. On the other hand, connection termination should also be done collectively.

The server collective services is necessary to allow servers to do many-to-one and one-to-many communications efficiently across different types of networks, and in a scalable way without creating hot-spots in the network. Collective communications are reliable – messages are retried until when they succeed or when the node is known to be dead.

The current presence of servers or membership is discovered and maintained through the gossip protocol. The gossip protocol is deal for node health monitoring in large networks due to its propagation delay logarithmic in the size of the system and negligible protocol overhead.

The gossip protocol and the server collectives both run over the Lustre Networking stack, a.k.a. the LNet, and benefit from its support of many different networks and routing between them.

Solution Requirements

All requirements below that are not explicitly marked as optional, are mandatory for successful completion of acceptance criteria.

1. Server membership discovery with gossip protocol

Discover server presence and all present servers should reach agreement as to who is present in bounded time. But:

- The set of servers that could be present are known in advance. Among this set of servers, we try to discover who is currently present – we don't attempt to discover totally new server which is previously unknown to us.
- Adding new servers to the current set of known servers may involve interruption to the gossip protocol.
- Network partition is handled by requiring more than half of the known servers to be present, i.e. minimum quorum must be bigger than half of the total number of known servers. Until the minimum quorum is reached, a subgroup considers itself isolated and can't believe that they are the whole cluster.
- We're not expecting to make progress with unstable nodes that appear and vanish in seconds.
- We can't handle Byzantine faults, e.g. a malicious node forging messages with false data.
- Keep protocol overhead low.
  The overhead of most concern to us is the bandwidth and message rate on every node – they must be kept within low limits and must stay even among all nodes. Also, CPU and memory overhead on each server node should not be excessive.
- Use high-priority messaging (**optional**, may requires excessive changes to LNet).

    - At end points: make use of existing LNet QoS facilities, e.g. queue gossip messages in a separate high priority queue.
    - In the network fabric: make it possible to easily distinguish gossip messages so that network administrators can set up QoS policy for gossip traffic. For example, use a dedicated IB service level for gossip messages, so that proper IB QoS policy can be configured for the service level in the IB fabric.

2. Topology change detection and membership agreement

When one or more new servers join or some existing ones go down, the changes should be detected and all present servers should reach agreement about the changes and a new server membership, i.e. they must all have a same set of currently present servers.

3. Reliable collective communication

Collective communications are reliable, and the underlying point-to-point communications should be transparently retried until when they succeed or when the node has become dead.

Collective communication can be delivered more than once. On each participant node, the collective service could deliver a same message to server subsystems for more than one time. It's the caller's responsibility to ensure that all collective messages are idempotent. For example, in client eviction messages an ID of the client connection instance to be evicted should be included, so that if the eviction message is delivered twice the second delivery wouldn't be able to terminate a new connection from the client established after the first eviction message was delivered and processed.

Collective communications can be delivered out of order. There is no guarantee that multiple collective communications are delivered in the same order by which they are initiated.

If a membership change happens when a collective communication is in progress:

- When new servers have joined, active collective communications simply ignore them. It's the server subsystem's responsibility to decide what to do with the new servers, for example repeat the collective over the set of new servers.
- When participant servers have gone down, the collective communication is aborted on all servers still alive.

4. Collective communication should be scalable and complete in bounded time.

Avoid communication patterns which could cause hot-spot in the network, e.g. ACK implosion when all destination nodes of a collective communication send acknowledgements directly to the source node.

5. Collective communication should use high-priority messaging (optional).

The collective API allows server subsystems to use high-priority messaging for collective communications. The messages should be given high priority both on the end points of communication and also on their paths inside the network.

6. Collective communications should work across different networks.

Participants in a collective communication need not to reside in a same network, or networks of a same type. They may span different types of networks. For example, some in TCP and the rest in Infiniband.

7. No backward compatibility with old protocol versions

Servers are required to run a same version of the protocol, and in case of a future protocol change all servers must be upgraded at a same time to a same new version.

8. Non-blocking API and modular design

The server collectives API should be non-blocking, and completion is signaled with events.

Also, a modular design should be adopted to make it easy to implement new collective algorithms, e.g. one that is optimized for a particular site topology.

9. Collectives are asynchronous

For many-to-one collective communications it should not be assumed that senders are synchronized and begin sending at a same time.

Use Cases

1. Client connection management

Servers should act like a coherent unit, managing client connections consistently, rather than requiring clients to connect to or disconnect from each of the servers one by one.

Connection establishment:

1. Client C connects to server A.
2. Server A broadcasts to all servers that "Client C wishes to connect".
3. All servers acknowledge back to server A that "I've established necessary connection state". The acknowledgements are aggregated on the their way to server A.
4. Server A replies to client C, who can now communicate with any server. Note that any per-server state the client requires (e.g. last committed) has to be established lazily when the client actually needs to talk to that server.

Connection termination:

1. Servers X, Y and Z concurrently decide to evict client C. They send notifications to server A, the client's "home" server for the current server cluster membership.
2. Server A broadcasts "evict client C" to all servers.
3. All servers acknowledge back to server A that "Client C has been evicted OK". The acknowledgements are aggregated on the their way to server A.
4. Server A broadcasts again that "Client C has been evicted globally".
5. All servers acknowledge that "OK I know we have now all evicted client C". Again

the acknowledgements can be aggregated.

Aggregation during connection and eviction storms are handled by servers processing requests lazily. An adaptive heuristic adjusts laziness by gradually increasing or decreasing the amount of time server waits for requests in hope of aggregation, based on the number of recent incoming requests. The mechanism is described in details later in the Solution Proposal.

2. Recovery

Collective communications can be used to coordinate rollback/rollforward across all VOSDs.

3. Scalable client health monitoring

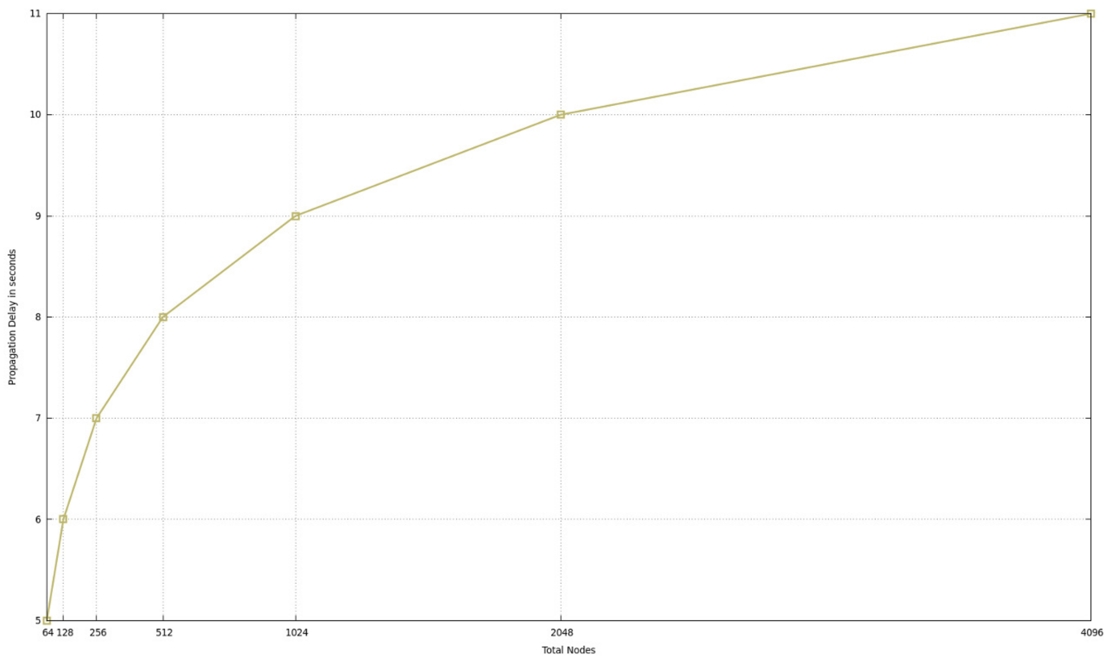
Client failures can be notified to all servers by collective communications, so servers don't have to burden themselves by joining the client monitoring network, e.g. gossip networks running over client clusters.

Solution Proposal

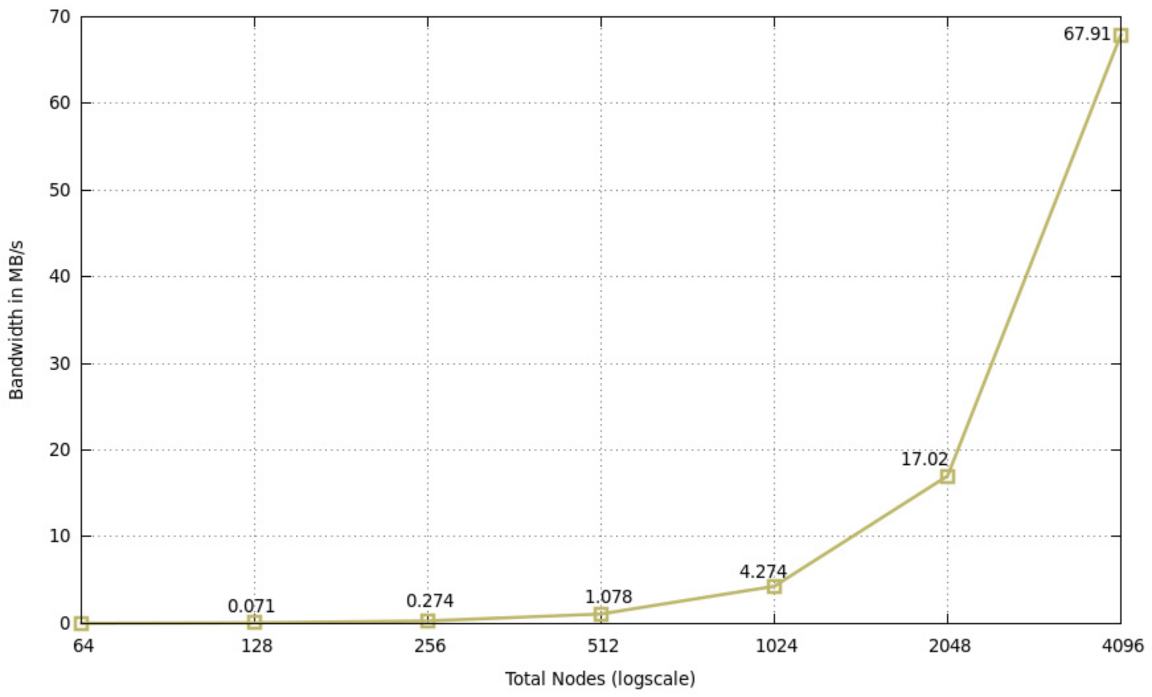
1. Server membership discovery and change detection by Gossip protocol

Gossip protocol is known to be fault tolerant and scalable with minimal protocol overhead. Propagation latency is logarithmic in size of the system.

We've simulated a gossip model with the Rensselaer's Optimistic Simulation System (ROSS). Our simulation results have verified the logarithmic propagation delay (gossip rate at once per second):



The aggregated bandwidth overhead of the protocol is O(N²), N being the size of the cluster, but with careful protocol design to avoid unnecessary data exchanges we're able to keep total bandwidth overhead at only 67.91MB/s for a 4096 node cluster, i.e. 16.98KB/s per node (gossip rate at once per second):



The low bandwidth overhead also makes it possible for the network to give gossip messages higher priority without penalizing other traffic. High-priority messages is not just desirable but also necessary in order to keep propagation delay bounded logarithmic in system size. Simulations showed that large jitters in the network caused propagation delay to grow out of bound.

We've also verified fault tolerance of the protocol. For example, at 5% message loss rate, our simulations showed almost no increase in propagation delays for clusters of sizes from 64 to 4096 nodes. The fault tolerance comes from the redundancy in the protocol. For example, if a message from A to B is lost, B could still get the same data indirectly from C if A has recently communicated with C.

Consensus of server membership

Servers can reach consensus of the current membership by:

1. Each server maintains a SHA-1 digest of a list of NIDs of the currently dead servers, and a version number of this digest. This list is sorted in increasing numerical order, and server NIDs are kept in network byte order. Each time a server notices a membership change, it computes a new SHA-1 digest and increment the version number by one. Note that:
   - The SHA-1 digest can be updated lazily, e.g. once in each gossip cycle if necessary, rather than at each event of membership change.
   - The digest is computed over the list of dead servers instead of the list of live servers, because the requirement of minimum quorum guarantees that there's always less dead servers than live ones. When there's more dead servers, the gossip protocol keeps exchanging server timestamps in hope of the network recovering from the partition, but membership digests are no longer updated and exchanged.
2. When servers gossip with each other, in addition to the liveness timestamps exchanged, they also exchange the version numbers of all SHA-1 digests they know about. When server A receives a gossip request message from server B:
   1. Server A looks at each digest version in the message:
      - If the version is newer (i.e. larger) than the version server A knows about, it sends a request for the actual SHA-1 digest in the gossip reply it's going to send back to B.
      - If the version is older, then include server A's copy of the digest and version in the gossip reply to be sent to B.
   2. Server B looks at the reply from A:
      - If there's requests for digests, send A another message containing the requested SHA-1 digests.
      - If there's digests and versions, update its local copy of the digests and versions with those found in the reply message.
   3. Server A updates its local knowledge of digests and their version numbers from the data sent by B, if any.
3. Servers know that a consensus has been reached when all SHA-1 digests of all present servers are the same. Note that digests versions may still be different as they are updated only locally by each server.

The overhead of computing and exchanging digests should be negligible because:

- SHA-1 digests are computed over the list of dead servers, the number of which is often fairly low.
- Digests are updated lazily, rather than at each membership change event.

- Actual digests are exchanged only when necessary. Usually only the versions are exchanged.

Weak consensus

By the protocol outlined above a strong level of consensus can be reached. But absolute consensus is still not possible as some digests might be already obsolete and the newer ones haven't been propagated yet. On the other hand, the collective communication service does not require a very strong level of consensus, because collective communication can be aborted and retried if the membership digest in a collective message is found to be different from the digest on some participant node.

This allows us to explore alternative protocols that creates less overhead at the price of yielding weaker consensus, e.g. by exchanging digests more lazily. Alternative protocols should be modeled and simulated so that a good trade-off can be found between strong consensus and minimal overhead.

Protocol version check

Backward compatibility with older protocol versions is not supported, as it's required that servers all run a same version and are upgraded to new versions at a same time. However, it's still possible that some servers run an older version, e.g. due to a human mistake.

The current version of the protocol is included in message headers. Any message that carries a different version is simply ignored. Therefore any server that runs a wrong version will be essentially treated as dead by the gossip protocol and prevented from participating any collective communication.

2. Collectives by point-to-point communications over server trees

Collective communications are implemented by generic point-to-point primitives that work across all networks supported by the Lustre networking stack.

The branching ratio of the spanning trees is an important factor in determining the latency and efficiency of collective communications. For example, when the number of participants is small a one-level tree should be used, which is equivalent to a linear algorithm where the source(s) directly communicates with the destination(s). A good default value for branching ratio should be chosen after experiments. Administrators are allowed to change this value though a same branching ratio must be used on all servers.

A deterministic algorithm computes any required spanning tree over any given subset of servers. Therefore in a collective communication, all participant servers will arrive at a same tree given the same inputs (i.e. root node, server subset and branching ratio).

Message aggregation

Point-to-point messages in collective communications can be merged together:

1. Messages belonging to different collective communications passing between the same pair of nodes may be concatenated to reduce per-message overhead.
2. Messages belonging to the same collective communication are merged together when possible on their path from the leaves back to the root of the spanning tree.

There are two separate queues for each possible peer server:

1. Opportunistic Aggregation (OA) queue: All collectives (requests or acknowledgements) get queued here for a short period of time in the hope they can be concatenated to reduce messaging overheads. Incoming requests traveling from root to leaves are replicated immediately to all child OA queues.
2. Acknowledgement merge queue: Incoming acknowledgements traveling from leaves to root are queued here **indefinitely** until they have merged with all their sibling acknowledgements. When a new incoming message is added to this acknowledgement merge queue:
    1. If no sibling message is found in the queue, it is simply added.
    2. If there's a sibling message already, it is merged.
    3. When the final acknowledgement message is merged, it is moved to the Opportunistic Aggregation queue, so that it could be concatenated with messages from other collectives by chance.

All queues of both types are discarded on a group membership change that removes any peer servers, except that the acknowledgement merge queue can possibly stay in some cases, e.g. if none of the missing peers are on the path to the root for their tree or are the children been waited on for acknowledgements.

For example, when a group of servers are notifying server A that they have all evicted client C, server A is only interested in the event that all servers have evicted – it doesn't help to notify server A eagerly when only some servers have evicted successfully. Therefore nodes can actually block for messages from their children in order to merge them together, until there's a server membership change. If at the same time, there happens to be another message from a different collective, it can also be concatenated together but nodes can't block for such messages.

Aggregation by server subsystems

Message aggregation in the collective service layer is essentially concatenation, which reduces message rate but can't reduce bandwidth rate, because the collective service layer does not understand data in the collective messages. However, the server subsystems, i.e. the users of the collective service layer, can do a much better job at message aggregation. For example, if server A received three connection requests from clients C1, C2 and C3, it can initiate one collective that says "Clients C1 C2 and C3 want to connect", resulting in one collective message. If the server chooses to send three collectives one for each client, the three messages can still be aggregated into one at the collective service layer, but the difference is that the message size will be much larger.

Servers can aggregate collectives by handling incoming requests lazily. This is particularly important to cope with a storm of connection requests or eviction requests, e.g. when a client cluster boots up or an inter-switch link to some client network has gone bad. Servers can use different levels of laziness to handle different requests, e.g. connection requests should be handled more eagerly as clients are eagerly waiting for progress while eviction requests can be handled more lazily.

Moreover, laziness can be dynamically adjusted, for example by increasing the amount of time to wait for more requests to come if server has become more confident that more requests are coming based on recent workload.

Reliability

As a message traverses a server spanning tree, the receiving node sends an acknowledgement back to the sending node, and sending node will retry the message if it hasn't heard the acknowledgement in time. The retry can be done eagerly because the gossip protocol ensures that the receiving node is very likely to be alive and messages are treated with high priority.

A consequence of the sender retrying messages is that the receiving node may end up receiving multiple copies of a same message. A receiving node therefore should keep at least some minimum states of an incoming message for as long as the sender could still be retrying it and the final retried message could still be in transit in the network, so that it's capable of detecting some duplicates. When a duplicate message is received:

- The collective service should not deliver the message to server subsystems, i.e. the users of collective service. This can't guarantee delivery for at most once, because we don't try to detect all possible duplicates.
- The collective service should not pass the message on to the next node in the tree. Otherwise a cascading resend of the duplicate may ripple through the rest of the spanning tree, which is a great waste of network resources.

Unit/Integration Test Plan

All tests should be run with simulated message droppings and delays at random to ensure robustness of the protocols. All tests should be run over different types of networks, e.g. at least TCP and Infiniband, and a mixture of networks of different types - there should not be any functional and behavioral changes.

1. Initial discovery at cluster boot up

When all servers boot up with different booting time, every server should discover everyone else in bounded time. Order in which servers come up to life should cause no difference in discovery time. Partition is not allowed in any case.

2. Change detection when multiple servers go down and up

When multiple servers go down and up, the changes should be detected by all live servers and agreement should be reached among them in bounded time.

Also, ephemeral membership agreements should be avoided.

3. Gossip protocol scalability

Run tests at scale as long as test facilities allow, and simulate at larger scales if necessary.

- Propagation delays should be bounded and logarithmic in size of the system.
- Protocol overhead (number of messages per second, and number of bytes sent/received per second) should be kept minimal.

4. Collective communications

- One-to-many collective: latency should be bounded and logarithmic in the

number of participants.
- Many-to-one collective: messages should be aggregated as they travel to the destination.

5. Collectives during topology changes

Collective communications should notice the changes and:

- Ignore addition of new servers.
- Abort when participant servers have become dead.

6. Collectives during network congestion

Verify that when normal communications are heavily congested:

- Gossip protocol could still make steady progress without ringing false alarms, i.e. mistakenly treating a live server as dead.
- Collectives with priority messaging can still make progress.

Acceptance Criteria

Server collective communications should be considered functional when:

1. Discovery and change detection complete in bounded time.
2. Discovery and change detection continue to work in case of dropped or delayed messages.
3. Collective communications complete in bounded time, and are aggregated when possible.
4. Collective communications abort itself in case of topology changes.
5. Gossip protocol and collective communications with priority messages make progress when the network is congested.

All tests identified above complete.