Date:
Feb. 15, 2013

High Level Design – Reduction Network Discovery

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
Feb. 15, 2013	1.0	Isaac Huang

Introduction

The Server Collectives module relies on the Reduction Network Discovery module to discover the current presence of servers or the current membership, and maintain a consistent view of the membership across all servers, in a highly scalable way. This can be accomplished by an implementation of the gossip protocol.

The gossip protocol has proven ideal for scalable node health monitoring in large networks due to its propagation delay logarithmic in the size of the system and negligible protocol overhead. Our implementation of the gossip protocol runs over the Lustre Networking stack, a.k.a. the LNet, and benefits from its support of many different types of networks.

Definitions

The following definitions are used throughout the rest of this document:

- **Gossip**: the actual protocol to implement Reduction Network Discovery.
- **N**: the total number of participants in the protocol, which includes those that are currently not alive.
- **LNet**: the Lustre Networking stack.
- **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.

Changes from Solution Architecture

Consensus turned out not necessary for upper layer protocols, and thus removed from this design.

Specification

The goal of this section is to clearly explain how you will implement what is described in the Solution Architecture. Focus on Functional specifications and augment with logical specifications where applicable; limit the amount of pseudo-code in the specification to that which clearly explains particular points.

Protocol

Initialization

A participant is initialized with the following protocol configuration parameters:

- Gossip interval in milliseconds.
- NIDs of all participants in the protocol.

In addition, the protocol version number for the current implementation is another important protocol parameter, which is hard coded in the implementation. It's a requirement that all participants run a same version of the protocol.

All participants should agree with the values of these parameters; otherwise the protocol will not function properly.

A participant first does sanity checks on these parameters to make sure the values are valid:

- Gossip interval is not allowed to be less than 200 milliseconds, i.e. 1/5 of a second. Otherwise it'd be difficult to make the protocol stable due to jitters in message delivery delay in the network, and jitters in local scheduling delay in the operating system.
- Gossip interval is also not allowed to be less than the half of the estimated Round Trip Time of the network.
- All participants in the protocol must belong to a same Lustre network. It's not supported to run the protocol over different Lustre networks, e.g. on both the @tcp0 network and the @o2ib0 network.

During the first N gossip cycles, these parameters are also carried in the headers of each outgoing message in order to make sure that all participants agree on the values. Note that:

- It's not feasible to include an enumeration of all participant NIDs in a message header. Instead, a SHA-1 digest of an ordered list of all NIDs is included and compared against local digest on the recipient.
- To reduce protocol overhead, the parameters are carried in headers only during the initial N gossip cycles. With good pseudo random number generation, N cycles is more than sufficient to cover all participants – there's N outgoing messages and N incoming messages so parameters are checked with 2N participants at random.
- Whenever parameter disagreement is detected on a participant, it stops participating in the gossip protocol and prints an error message on the local console. It must be solved by administrators – there's no automatic parameter negotiation.

Local State Management

Each participant maintains two local states: a Lamport clock of global gossip cycle number and a vector of ages for all participants.

Lamport clock of global gossip cycle

The Lamport clock keeps record of the current global gossip cycle number. It is initialized to zero and updated according to the following rules:

- It is incremented by one right at the beginning of each gossip cycle, i.e. before each outgoing gossip ping message is sent.
- It could be synchronized with another participant when processing each incoming message, either a ping or a reply: if an incoming message contains in its header a

Lamport clock of value C which is no less than the local clock, then the local Lamport clock is set to C+1.

Usually the Lamport clocks of all participants are fairly in sync with each other. When a participant reboots itself or has just been added, it should be able to synchronize its Lamport clock often in just one gossip cycle – most likely by a reply to the first ping message it sends.

The Lamport clock is useful to detect obsolete message and to help debugging, both described in details later.

Age vector

The age of a participant A as perceived by participant B is defined as the number of gossip cycles since the last time participant B heard of A directly or indirectly. For examples:

- Participant A pings B directly, then B updates its copy of A's age as 1.
- Participant C pings B and tells B that its copy of participant A's age is 3, then B updates its copy of A's age to 4 unless its local copy is less than 4 already.

Each participant maintains a vector of ages for every participant in the protocol including itself, and this age vector is updated according to the following rules in the following order:

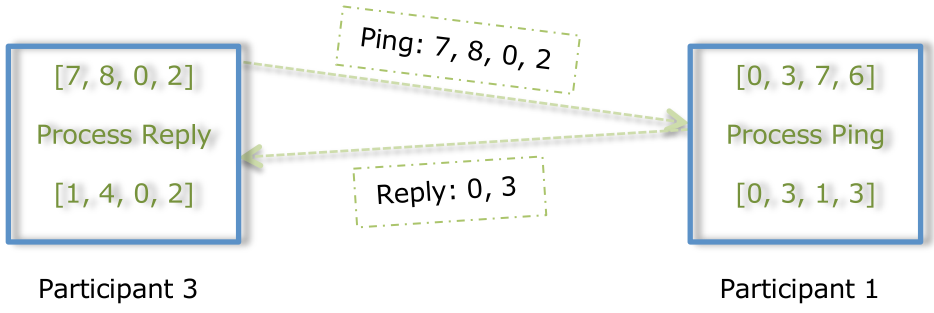
1. On each gossip cycle, ages[i where i >= 0 && i < N] += 1 - all entries in the age vector is incremented by one, i.e. every entry has aged by one cycle.
2. Upon receiving an incoming message, the local age vector is updated from entries in the remote vector carried in the message: local[i] = MIN(local[i], remote[i]+1), i.e. local age vector is updated if the incoming message carries more recent information. Entries in the remote vector are first incremented by one under the assumption that it took the network no more than one gossip interval to deliver the message.
3. Ages[myself] = 0, i.e. the current participant is always alive at any moment.

Rule 2 implies that the gossip interval should be longer than half of Round Trip Time in the network.

State Exchange

The gossip protocol works by each participant simply exchanging age vectors with one other randomly chosen participant in each gossip cycle.

Each state exchange comprises of two messages: a ping and a reply. The following graph illustrates the protocol:



There're four participants in the gossip protocol:

1. Participant 3 has local age vector [7, 8, 0, 2] and sends a copy of this vector in a ping message to participant 1. Note that entry 3 has value 0 because it stands for participant 3 itself.

2. Participant 1 receives the ping message, and updates its local age vector because age vector entries 3 and 4 in the message are more recent. Note that age vector entries in the message are incremented by 1 before use, under the assumption that the message left its origin no more than 1 gossip cycle ago.

3. Participant 1 sends a reply message back to participant 3. Note that the reply message is shorter than the ping message and contains only two entries in the age vector, because participant 3 only needs these two entries to update its local age vector.

4. Participant 3 receives the reply message and updates its local age vector because the message contains two newer age entries. Note that at this point, the local age vectors on both participants are more in sync than before the gossip message exchange.

Gossip Ping

At each gossip cycle, each participant chooses another one to gossip with at random. Then it sends the chosen participant a gossip ping message that contains:

- A header that includes its current Lamport clock, and for the initial messages the protocol configuration parameters.
- A complete copy of its local age vector.

The sender does not need to keep any state information about the outgoing ping message that hasn't been replied yet – i.e. when the reply comes back later the sender can process it without any information from the matching ping message. In this regard, the protocol is stateless, which simplifies implementation.

Gossip Reply

Upon receiving an incoming ping message, a participant first updates its own Lamport clock and age vector according to rules given previously. Then it checks protocol configuration parameters if they're included, and if there's no discrepancy it prepares a reply message that contains:

- A header that includes its current Lamport clock, and for the initial messages the protocol configuration parameters.

- A partial copy of its local age vector. Since sender's age vector is now known from data in the ping message, the reply doesn't need to contain a complete copy of the local age vector. Only those entries that are at least two cycles more recent need to be included. A useful optimization here is lazy reply: for the purpose of detecting participant health, the age vectors can be just loosely synchronized. For example, if the local age for a participant is only 4 cycles newer than the sender's copy, it might be fine not to include the age entry in the reply message. By using this laziness fudge factor, bandwidth overhead of the protocol can be reduced without any impact on propagation delays. The value of the fudge factor depends on log(N) – log(N)/4 should be fine but it should never be more than log(N)/2.

When receiving a reply message, a participant updates its local Lamport clock and age vector from data in the reply, in exactly the same way as how local states are updated from a ping message with the only difference that there's going to be no further message exchange. Protocol configuration parameters are also checked if they're included in the header.

Detect Obsolete Message

An important assumption of this implementation of gossip protocol is that it should take less than a gossip interval for the network to deliver a gossip message – i.e. the age vector included in any gossip message carries entries no more than one cycle old. If a message arrives at a recipient many cycles after it was sent, it should be discarded because the data in it could be way obsolete, e.g. it may contain an age entry of value 2 for a participant that is already dead. If such messages are not detected and discarded, it may take the protocol much longer to stabilize. And if there're too many of such messages the protocol might not be able to stabilize at all. Furthermore, data in a single obsolete message will be propagated by the recipient to others who will then believe that the propagated data is recent.

Our gossip implementation employs two techniques to minimize risks of obsolete messages:

- It's required that the gossip interval to be longer than half of the estimated Round Trip Time of the network. Since Round Trip Time of a network is dynamic and gossip interval can't change while the protocol is running, the gossip protocol doesn't try to measure the RTT. Instead, it's up to the administrator to specify a safe estimated value of RTT.
- Gossip messages are treated with higher priority through available QoS mechanisms in the network.

While by the help of these two mechanisms the risks of much-delayed messages can be reduced, they can't be completely eliminated. The Lamport clock included in every message header is designed to help detect and ignore such obsolete messages:

- When a ping message is received: if the Lamport clock in the ping header is more than log(N) older than the current local Lamport clock, the message is considered obsolete and is discarded, then the recipient responds with a reply message which contains only a Lamport clock in the header but no data in it – this empty reply message is necessary in the case that the sender has just begun to participate and needed to have its Lamport clock synchronized with others.

- When a reply message is received: if the Lamport clock in the reply message header is more than log(N) older than the current local Lamport clock, the message is considered obsolete and is simply discarded with no further action.

It's possible to discard messages more aggressively than the log(N) difference because:

- The log(N) difference is too slack and only expected in the worst possible case of out-of-sync Lamport clocks.
- The gossip protocol itself is very robust against message loss: a small rate of falsely ignored messages is no different from a small rate of message loss by the network. Simulation results showed that even at 5% message loss, there was little impact on the propagation delay. In other words, the cost of ignoring some good messages is fairly low while the cost of accepting a single obsolete message can be very high, so the implementation should lean toward more aggressive dropping of potentially obsolete messages.

Determining Participant Health

Participants whose ages is more than log(N) cycles old is considered to be dead. But when choosing a participant to ping at each gossip cycle, dead ones should not be excluded in case they've just come back to life.

Implementation

The gossip protocol is implemented as a kernel module that runs on top of the Lustre LNet kernel module, which is administrated by a user space tool via ioctl system calls.

Administration Control Utility

The operations of the gossip protocol are controlled by a user space utility program called gossipctl, which communicates with the kernel module by ioctl system calls. The gossip kernel module processes one command at a time – there's no need for concurrent administration commands.

Note that for the purpose of the first demonstration, the administration tool might not be implemented. Instead, parameters could be passed by kernel module options and the protocol would be started or stopped at module loading or removal.

Initialization: gossipctl init

The *gossipctl init* command takes additional options to specify the gossip protocol configuration parameters and other options to initialize the gossip module:

- Expected Round Trip Time in milliseconds of the network which the participants belong to.
- Gossip interval in milliseconds.
- A specification of NID addresses of all participants. It could be one of the following formats

    - A simple list that enumerates all participant NIDs.
    - A format that specifies a range of participant NIDs, for example 192.168.[1-2].[1-128/2]@o2ib0.
    - A combination of the above two forms.
    - A filename that contains strings of any format mentioned above. However, file reading and parsing should be done by gossipctl in the user space.

  Note that this is the reason why the options can't be passed to the gossip kernel module by using kernel module options – the Linux kernel puts a 4K limit on the length of string options, and that could be insufficient for some large networks.

- A seed for pseudo random number generation. The gossipctl tool reads it from the /dev/random file, which is based on hardware mechanisms when available and system events with real entropy like arrival of network packets. The read could block until there're sufficient entropy events if there's no hardware support, but it should not be a problem as the gossipctl program only needs to read a few bytes from it.

Upon receiving the options from user space, the gossip kernel module performs the following initialization tasks:

- Validate all the options. For example:
    - Gossip interval should be no less than half of the specified expected Round Trip Time.
    - All participants must belong to a same LNet network.
- Create and initialize the Lamport clock and the age vector.

Start protocol: gossipctl start

When receiving the start command, the gossip kernel module performs the following actions:

1. Initialize LNet, and find out which local LNet interface is going to participate in the gossip protocol by looking for a match in the participant list, supplied by the *gossipctl init* command.
2. Create an LNet event queue for all gossip messages.
3. Set up the LNet portal for gossip messages as a lazy portal.
4. Allocate message buffers and post them to LNet to receive incoming messages.
5. Begin the gossip cycles and start running the gossip protocol, i.e. pinging others and replying to incoming pings.

Stop protocol: gossipctl stop

The gossip kernel module performs the following actions:

1. Stops sending ping messages to other participants and replying to incoming gossip pings.
2. Revoke buffers to receive messages from the LNet and free them.
3. Destroy the gossip message LNet event queue.
4. Finalize LNet.

Note that reclaiming resources allocated during initialization in response to *gossipctl init* is delayed until either the gossip kernel module is unloaded or another *gossipctl init* command is received with different parameters.

Buffer Management

The maximum size of any incoming message can be now calculated as the number of participants is known.

Ideally during each gossip cycle any participant would only receive two incoming messages: one incoming ping message from another random participant, and one reply to the ping message that was just sent. However, more than one participant could choose to ping a same node during a same gossip cycle as the random choice of participants to ping can't be close to truly random. Moreover, messages sent from previous cycles could be delayed by the network and end up arriving in a same cycle. To accommodate these factors:

- Each participant posts and keeps 4 free buffers in any gossip cycle to receive incoming messages.
- The LNet portal for incoming messages should be set as "lazy", so that incoming messages wouldn't be dropped when there's no free buffer on the portal to receive them. Instead they are blocked until there's more buffer made available by the gossip module.

Matching Buffers with Incoming Messages

LNet matches incoming messages with posted buffers by using match bits. Buffers can be posted to match only a specific match bit or any match bit, which is contained in LNet message headers.

Buffers for both ping and reply messages are posted on a same portal as match-any buffers. Ping messages are different in that they are unsolicited so match-any buffers should be posted to receive them. But it is not necessary to use match-one buffers for reply message, because it could add unnecessary complexities to the buffer management code:

- A separate portal may be required to dedicate to reply messages, from the limited portals space, as match-one and match-any buffers can't co-exist on a same LNet portal.
- Match-one buffer has to be reclaimed because the only incoming reply that it is expecting to receive might not arrive.

Although match-any buffers are posted to receiving ping and reply messages, senders can still use different match bits for them in order to distinguish ping messages from reply messages at the receiver. As a consequence, there's no need to add a type field in the gossip message headers to tell whether the message is a ping or a reply – match bits in LNet message header, available as part of LNet events, will enable the gossip module to distinguish them.

Buffer Size

Although the maximum size of incoming messages is known, it can be complex to post buffers that has just enough room to receive messages of the maximum size. This is because:

- The maximum message size depends on the number of participants, and also the current protocol version. If some participants are added later, then all current participants may have to repost all buffers, since their current buffers may be insufficient to accommodate additional age vector entries for the newly added participants.

Instead, all participants post buffers of size LNET_MTU (which is 1M bytes), the maximum size for LNet messages, so that there's no need to repost buffers after new participants have been added. Also, there's very little waste of memory since each participant keeps only four free buffers for incoming messages.

Message Passing

All outgoing gossip messages are sent by the LNetPut API, with the *src_nid* parameter set to be the active local NID in case the participant is multi-homed (in order to make sure that the message will be sent by the correct interface). As the gossip protocol does not require reliable message delivery, there's no need to time out and retransmit outgoing messages, which simplifies the implementation a lot as a proper timeout mechanism should adjust to the current network Round Trip Time dynamically.

On-wire Message Format

One byte is sufficient for each age vector entry sent over the wire because:

- An age of 255 cycles is more than sufficient to declare a participant as dead for cluster sizes of our concern, as the propagation delay is logarithmic with the size of the system.

There's room for about one million age vector entries in a single LNet message, which is sufficient for server-side health monitoring in this project. Then one gossip message can be sent by just one LNetPut call.

Furthermore, it cuts down bandwidth overhead of the gossip protocol by a lot, and avoids byte-order issues - e.g. it eliminates the need to detect byte order difference and convert between different byte orders.

High priority messaging

Currently the LNet API does not provide any Quality of Service support. However, we can add a simple and effective priority mode for gossip messages without worrying about penalizing other traffic, because of the low protocol overhead – in each gossip cycle any participant usually sends only two messages, a ping and a reply in response to one incoming ping.

The priority messaging mechanism works as follows:

1. A new option is added to LNet MD, i.e. the LNet memory descriptor, to tell LNet that this is a high priority message.
2. The gossip module turns on this option for memory descriptors of all its outgoing messages, and calls LNetPut to send them out.
3. LNetPut recognize this option and short cuts all queues in LNet and LND layers. Of course if there's still any pending high priority message, the new one shouldn't overtake any of the previous ones.

This mechanism only prioritizes gossip messages on the end points, but not on the path between end points. The latter requires the ability to distinguish gossip messages from other traffic without the knowledge of LNet/gossip message formats, and is an optional requirement.

Debugging

Our implementation of the gossip protocol is fairly simple by intentional design choices, but still debugging distributed application can be a tough task as always.

Each participant maintains O(N) state information, so there could be O(N**2) distributed states in total to examine in order to troubleshoot any problem. Furthermore, the O(N**2) state information changes constantly in every gossip cycle, so likely some history of the states would be necessary to nail down bugs. Therefore debugging support must be considered as early as in the designs to facilitate maintenance of the code in the future.

The following mechanisms are designed to help debugging:

- Each debug message is prefixed with the current local Lamport clock value. If the event involves an incoming message, Lamport clock included in the message header is also added as part of the debug message. The Lamport clocks can be helpful to infer partial order of distributed events in the whole system.
- Debug messages are divided into multiple severity levels.
- Debug messages should be used sparingly to increase signal to noise ratio. For example, while it'd be excessive to generate a debug message each time a local age vector entry gets updated, a debug message should not be omitted if the age change triggers a participant health status change too.
- Local state information should be exported to the user space via files under the /proc pseudo file system. This should include at least the protocol configuration parameters, the Lamport clock, and the age vector. This makes it possible to use normal utilities like cp or tar to capture and save the states easily.

API and Protocol Additions and Changes

Open Issues

Risks & Unknowns

It’s still not clear to us how much the Lamport clocks can go out of synchronization in clusters of different sizes. Simulations need to be done to help find out. The log(N) value should be a very safe baseline value to use, but it limits our ability to detect obsolete messages. TODO: run simulations to find a more aggressive value to better detect obsolete messages yet without ringing too many false alarms.