High level design of Colibri Network Benchmark

By Maxim Medved <max_medved@xyratex.com>

Date: 2012/02/20
Revision: 1.0

[Text in square brackets and with a light-green background is a commentary.]

This document presents a high level design (HLD) of Colibri Network Benchmark. The main purposes of this document are: (i) to be inspected by C2 architects and peer designers to ascertain that high level design is aligned with C2 architecture and other designs, and contains no defects, (ii) to be a source of material for Active Reviews of Intermediate Design (ARID) and detailed level design (DLD) of the same component, (iii) to serve as a design reference document.

The intended audience of this document consists of C2 customers, architects, designers and developers.

0. Introduction

[This section succinctly introduces the subject matter of the design. 1--2 paragraphs.]

Colibri network benchmark is designed to test network subsystem of Colibri and network connections between nodes that are running Colibri.

[A note on notation.

The most unusual feature of the present specification format is a system of tagging used for requirements tracking purposes. Overall goals of requirements tracking are described in this document. The crux of the matter is the idea of using tags to track dependencies between system components and between layers of the system specification (architecture, high level design, detailed level design, implementation). A part of system architecture is a Summary Requirements table that contains basic architecture level requirements for many system components. Additional requirements for a particular component are collected during requirements analysis phase of development. A requirement is labelled by a tag having form [r.component.tag]. Requirements section of an HLD collects together all requirements for the component. As design is elaborated in functional and logical specifications, dependencies on properties of other components can arise. Such a dependency is recorded as a [u.component.tag] label in a footnote, added at the point where the dependency is introduced, where "tag" somehow denotes property the design depends on. If the property is required by the Summary Requirements table, " ST" is added to the label. All new dependencies introduced by the specification are enumerated in a Dependencies sub-section of Logical specification.

In addition to requirements for other components, design can introduce dependencies on the lower specification layers of the same component. For example, a high level design specification is usually correct under certain assumptions that can be validated only in detailed level design. A detailed level design specification is similarly dependent on requirements to component implementation. For simplicity, such intra-component dependencies are not recorded as footnotes. Instead they are enumerated a Refinement sub-section.

After a design has been described, it is necessary to show that it conforms to the stated requirements. To this end, a Conformance sub-section describes how requirements are discharged using [i.component.tag] labels.

With this notation in place, the following requirements integrity condition can be checked:

> for every [r.component.tag] in the system documentation there is a [i.component.tag], somewhere in the documentation. That is, every requirement is implemented.

Additionally, [u.component.tag] tags allow to back-track every requirement to its particular users, providing a way to estimate how the system would be affected if the requirement is to be abandoned or modified.

The following color marking is used in this document: incomplete or todo item, possible design extension or future directions.]

1. Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible. References to the C2 Glossary are permitted and encouraged. Agreed upon terminology should be incorporated in the glossary.]

- *c2_net_test:* a network benchmark framework that uses the Colibri API;
- *test client:* active side of a test, e.g. when A pings B, A is the client;
- *test server:* passive side of a test, e.g. when A pings B, B is the server;
- *test node:* a test client or server;
- *test group:* a set of test nodes;
- *test console:* a node where test commands are issued;
- *test message:* unit of data exchange between test nodes;
- *test statistics:* summary information about test messages for a some period of time;
- *bulk test:* a test that uses bulk data transfer;
- *ping test:* a test that uses short message transfer;
- *command line:* command line parameters for an user mode program or kernel module parameters for a kernel module;
- *mapping:* mapping from one test group to another. For example: how clients are mapped to servers, e.g. each client pings all servers so that each server being tested by all clients, or each client pings one server so that each server being tested by 1/N clients (N == # servers);
- *session:* a set of tests orchestrated by a test console. There can be only one session on each test node.

2. Requirements

[This section enumerates requirements collected and reviewed at the Requirements Analysis (RA) and Requirements Inspection (RI) phases of development. References to the appropriate RA and RI documents should go here. In addition this section lists architecture level requirements for the component from the Summary requirements table and appropriate architecture documentation.]

- **[r.c2.net.self-test.scalable]**: should target 100,000 test nodes in a test session;
- **[r.c2.net.self-test.statistics]:** should be able to gather statistics from all nodes;
- **[r.c2.net.self-test.statistics.live]:** display live aggregate statistics when the test is still running;
- **[r.c2.net.self-test.statistics.persistence]:** statistics should be saved persistently for later retrieval;
- **[r.c2.net.self-test.statistics.purge]:** purge saved stats;
- **[r.c2.net.self-test.statistics.complete]:** statistics saves enough information for analysis;
- **[r.c2.net.self-test.statistics.fatal-unsafe]:** statistics need not survive fatal errors;
- **[r.c2.net.self-test.test.ping]:** simple ping/pong to test connectivity, and measure latency;

- **[r.c2.net.self-test.test.bulk]:** bulk message read/write, with varying sizes, and optional integrity checks;
- **[r.c2.net.self-test.test.bulk.size]:** able to specify size of bulk tests, in a system independent way, e.g. in bytes instead of in pages, optional: in different offsets and # of fragments;
- **[r.c2.net.self-test.test.bulk.integrity]:** end-to-end data integrity checks, in various forms;
- **[r.c2.net.self-test.test.bulk.integrity.no-check]:** for pure link saturation tests, or stress tests. This should be the default.
- **[r.c2.net.self-test.test.bulk.integrity.simplistic]:** e.g. sender includes a well known magic every K bytes, to be checked by receiver.
- **[r.c2.net.self-test.test.bulk.integrity.checksumming]:** crc32 or whatever. Used when there's possible data corruption;
- **[r.c2.net.self-test.test.bulk.integrity.paranoid]:** each byte is populated with a well known pattern, e.g. pre-defined magic + byte offset, receiver verifies each byte;
- **[r.c2.net.self-test.test.duration.simple]:** end user should be able to specify how long a test should run, by loop;
- **[r.c2.net.self-test.test.duration]:** end user should be able to specify how long a test should run, by loop or by time interval. e.g. node A should keep pinging node B for 12,000 loops, or for 1 hour, or whichever comes first;
- **[r.c2.net.self-test.concurrency]:** on a test client, a single test should be able to run in several instances currently, e.g. node A should run 8 ping tests concurrently to B (for a duration of N seconds or M iterations);
- **[r.c2.net.self-test.kernel]:** test nodes must be able to run in kernel space;
- **[r.c2.net.self-test.user]:** test nodes must be able to run in user space;
- **[r.c2.net.self-test.cross-address-space]:** must run across address spaces over the network;
- **[r.c2.net.self-test.synchronized]:** tests startup must be synchronized over the all nodes;
- **[r.c2.net.self-test.session]:** a test session can be created - it is a test group with some properties;
- **[r.c2.net.self-test.session.node-exclusivity]:** a test node can belong to only one session;
- **[r.c2.net.self-test.session.multiple-tests]:** there can be multiple tests in a session;
- **[r.c2.net.self-test.session.test-independency]:** tests in the session are independent from each other.

3. Design highlights

[This section briefly summarises key design decisions that are important for understanding of the functional and logical specifications and enumerates topics that a reader is advised to pay special attention to.]

- **Bootstrapping**
  Before the test console can issue commands to a test node, c2_net_test must be running on that node, either as a kernel module, or a user space process. External mechanisms are used to load kernel module or spawn user space process. Either pdsh or MPI should suffice. pdsh can be used to load a kernel module.
- **Clock synchronization**
  Clock synchronization is vital to get meaningful cluster-wide aggregate statistics, and to synchronize actions of test nodes.
  Timestamps should be added to a message as late as possible, and checked as soon as possible

when received.

- **Statistics collecting**
  Kernel module creates file in /proc filesystem, which can be accessed from the user space. This file contains aggregate statistics for a node.
- **Live statistics collecting**
  pdsh used for live statistics.

4. Functional specification

[This section defines a functional structure of the designed component: the decomposition showing *what* the component does to address the requirements.]

- **[r.c2.net.self-test.test.ping]**
  - *Test client* sends desired number of test messages to a test server with desired size, type, etc. Test client waits for replies (test messages) from the test server and collects statistics for send/received messages;
  - *Test server* waits for messages from a test client, then sends messages back to the test client. Test server collects messages statistics too;
  - Messages RTT need to be measured;
- **[r.c2.net.self-test.test.bulk]**
  - *Test client* is an passive bulk sender/receiver;
  - *Test server* is a active bulk sender/receiver;
  - Messages RTT and bandwidth from each test client to each corresponding test server in both directions need to be measured;
- *Test console* sends commands to load/unload the kernel module (implementation of c2_net_test test client/test server), obtains statistics from every node and generates aggregate statistics.

5. Logical specification

[This section defines a logical structure of the designed component: the decomposition showing *how* the functional specification is met. Subcomponents and diagrams of their interrelations should go in this section.]

5.1 Kernel module specification

5.1.1 start/finish

After the kernel module is loaded all statistics are reset. Then the module determines is it a test client or a test server and acts according to this. Test client starts sending test messages immediately.
All statistical data remain accessible until the kernel module is unloaded.

5.1.2 test console

Test console uses pdsh to load/unload kernel module and gather statistics from every node's procfs. pdsh can be configured to use different network if needed.

5.1.3 test duration

End user should be able to specify how long a test should run, by loop. Test client checks command line parameters to determine the number of test messages.

5.1.4 clock synchronization

External mechanism is relied upon to synchronize clocks. The 'ntpd' should be able to synchronize clocks at a sufficient resolution. When possible, it should be configured such that the clock synchronization traffic goes to a different network than the one being tested. For example, when c2_net_test is testing the Infiniband network, clock synchronization should use Ethernet, which is almost ubiquitous, for its traffic; or when testing Ethernet, run clock synchronization over an IP over IB network.

However, c2_net_test must verify that clocks have been synchronized properly. This can be done:

1. when a message is being sent out, the current time is included;
2. when a message is received, the timestamp carried by it is compared with the current time, clocks must be out of sync if the received time is equal to or later than the current time.

5.2 Test message

5.2.1 message structure

Every message contains timestamp and sequence number, which is set and checked on the test client and the test server and must be preserved on test server. Timestamp is used to measure latency and sequence number is used to identify messages loss.

5.2.2 message integrity

Currently, test message integrity are not checked either on the test client nor the test server.

5.2.3 measuring message latency

Every test message will include timestamp, which is set and tested on the test client.
When test client receives test message reply, it will update round-trip time statistics: minimum, maximum, average, standard deviation. Lost messages aren't included in these statistics.
Test client will keep statistics for all test servers with which communication was.

5.2.4 measuring messages bandwidth

- messages bandwidth can be measured only in the bulk test. It is assumed that all messages in the ping test have zero size, and all messages in the bulk test have specified size;
- messages bandwidth statistics is kept separately for from node/to node directions on the test server and total bandwidth only is measured on the test client;
- messages bandwidth is the ratio of the total messages size (in corresponding direction) and time from the start time to the finish time in corresponding direction;
- on the test client start time is time just before sending network buffer descriptor to the test server (for first bulk transfer);
- on the test server start time in "to node" direction is time, when the first bulk transfer request was received from the test client, and in "from node" direction it is time just before bulk transfer request will be send to the test client for the first time;

- finish time is time when the last bulk transfer (in corresponding direction) is finished or it is considered that there was a message loss;
- ideally time in “from test client to test server” direction must be measured from Tc0 to Ts4, and time in “from test server to test client” direction must be measured from Ts5 to Tc8. But in the real world we can only measure time between Tc(i) and Tc(j) or between Ts(i) and Ts(j). Therefore always will be some error and difference between test client statistics and test server statistics;
- absolute value of the error is O(Ts1 - Tc0)(for the first message) + O(abs(Tc8 - Ts8))(for the last message);
- this error can be partially corrected using RTT for the small messages;
- with messages number increasing the relative error will tend to zero.

Bulk message send/receive example

Test client		EQ		Test server
send network buffer descriptors (for the passive send/receive) to the test server	Tc0	=	Ts0	
	Tc1	=	Ts1	receive network buffer descriptors from the test client; initialize bulk transfer as an active bulk receiver
start passive bulk sending	Tc2	!=	Ts2	start active bulk receiving
finish passive bulk sending	Tc3	!=	Ts3	finish active bulk receiving
p_send_cb callback	Tc4	!=	Ts4	a_recv_cb callback
	Tc5	=	Ts5	initialize bulk transfer as an active bulk sender
start passive bulk receiving	Tc6	!=	Ts6	start active bulk sending
finish passive bulk receiving	Tc7	!=	Ts7	finish passive bulk receiving
p_recv_cb callback	Tc8	!=	Ts8	a_send_cb callback

- T[cs](0-8) - points in time when operation is started on the test client/server;
- Tc(i) <= Tc(i+1);
- Ts(i) <= Ts(i+1).

Start/finish time

	test client	test server	
	total	to node	from node
start time, first message point in time	Tc0	Ts1	Ts5
finish time, last message point in time	Tc8	Ts4	Ts8

5.2.5 messages concurrency

Messages concurrency looks like the test client has a semaphore, which have number of concurrent operations as it initial value. One thread will down this semaphore and send message to the test client in a loop, and other thread will up this semaphore when the reply message received or message considered lost.

5.2.6 messages loss

Messages loss are determined using timeouts.

5.2.7 message frequency

Measure how many messages can be actually sent in time interval.

5.3 Bulk test

5.3.1 test client

Test client allocates a set of network buffers, used to receive replies from test servers. Then test client sends bulk data messages to all test servers (as an passive bulk sender) from the command line.
After that, test client will wait for the bulk transfer (as an passive bulk receiver) from the test server.
Test client can perform more than one concurrent send/receive to the same server.

5.3.2 test server

Test server allocates a set of network buffers and then waits for a messages from clients as a active bulk receiver. When the bulk data arrives, test server will send it back to the test client as an active bulk sender.

5.4 Ping test

5.4.1 test server

Test server waits for incoming test messages and simply sends them back.

5.4.2 test client

Test client sends test messages to server and waits for reply messages. If reply message isn't received within a timeout, then it is considered that the message is lost.

5.5. Conformance

[For every requirement in the Requirements section, this sub-section explicitly describes how the requirement is discharged by the design. This section is part of a requirements tracking mechanism, so it should be formatted in some way suitable for (semi-)automatic processing.]

- **[i.c2.net.self-test.statistics]** statistics from the all nodes can be collected on the test console;
- **[i.c2.net.self-test.statistics.live]:** statistics from the all nodes can be collected on the test console at any time during the test;
- **[i.c2.net.self-test.test.ping]:** latency is automatically measured for all messages;
- **[i.c2.net.self-test.test.bulk]:** used messages with additional data;
- **[i.c2.net.self-test.test.bulk.integrity.no-check]:** bulk messages additional data isn't checked;
- **[i.c2.net.self-test.test.duration.simple]:** end user should be able to specify how long a test should run, by loop;
- **[i.c2.net.self-test.kernel]:** test client/server is implemented as a kernel module.

5.6. Dependencies

[This sub-section enumerates other system and external components the component depends on. For every dependency a type of the dependency (uses, generalizes, *etc.*) must be specified together with the particular properties (requirements, invariants) the design depends upon. This section is part of a requirements tracking mechanism.]

5.7. Security model

[The security model, if any, is described here.]

5.8. Refinement

[This sub-section enumerates design level requirements introduced by the design. These requirements are used as input requirements for the detailed level design of the component. This sub-section is part of a requirements tracking mechanism.]

See Conformance section.

6. State

[This section describes the additions or modifications to the system state (persistent, volatile) introduced by the component. As much of component behavior from the logical specification should be described as state machines as possible. The following sub-sections are repeated for every state machine.]

6.1. States, events, transitions

[This sub-section enumerates state machine states, input and output events and state transitions incurred by the events with a table or diagram of possible state transitions. UML state diagrams can be used here.]

6.2. State invariants

[This sub-section describes relations between parts of the state invariant through the state modifications.]

6.3. Concurrency control

[This sub-section describes what forms of concurrent access are possible and what forms on concurrency control (locking, queuing, *etc.*) are used to maintain consistency.]

7. Use cases

[This section describes how the component interacts with rest of the system and with the outside world.]

7.1. Scenarios

[This sub-section enumerates important use cases (to be later used as seed scenarios for ARID) and describes them in terms of logical specification.]

Scenario	[usecase.net-test.test]
Relevant quality attributes	scalability, usability
Stimulus	user starts the test
Stimulus source	user
Environment	network benchmark
Artifact	test started and completed
Response	benchmark statistics produced
Response measure	statistics is consistent
Questions and issues	

Scenario	[usecase.net-test.client.request]
Relevant quality attributes	scalability
Stimulus	client have messages to send
Stimulus source	user
Environment	network benchmark
Artifact	message from client to server
Response	message is sent to server
Response measure	message is formed and sent to server
Questions and issues	

Scenario	[usecase.net-test.server.response]
Relevant quality attributes	scalability
Stimulus	server received the message from client
Stimulus source	client

Environment	network benchmark
Artifact	message from server to client
Response	message is sent back to client
Response measure	message is formed and sent to client message timestamp and sequence number is preserved
Questions and issues	

Scenario	[usecase.net-test.client.acknowledgement]
Relevant quality attributes	scalability
Stimulus	client received the message from server
Stimulus source	server
Environment	network benchmark
Artifact	message from server to client
Response	statistics is updated with information from message
Response measure	
Questions and issues	

Scenario	[usecase.net-test.statistics.gathering]
Relevant quality attributes	scalability
Stimulus	became necessary to collect statistics
Stimulus source	user or live statistics gatherer
Environment	network benchmark
Artifact	test statistics
Response	statistics is gathered from all test nodes
Response measure	statistics is complete
Questions and issues	

Scenario	[usecase.component.name]
Relevant quality attributes	[*e.g.*, fault tolerance, scalability, usability, re-usability]
Stimulus	[an incoming event that triggers the use case]
Stimulus source	[system or external world entity that caused the stimulus]
Environment	[part of the system involved in the scenario]
Artifact	[change to the system produced by the stimulus]
Response	[how the component responds to the system change]
Response measure	[qualitative and (preferably) quantitative measures of response that must be maintained]

Questions and issues	

[UML use case diagram can be used to describe a use case.]

7.2. Failures

[This sub-section defines relevant failures and reaction to them. Invariants maintained across the failures must be clearly stated. Reaction to Byzantine failures (*i.e.*, failures where a compromised component acts to invalidate system integrity) is described here.]

7.2.1 network failure

Messages loss is determined by timeout. If the message wasn't expected and if it did come, it is rejected. If some node isn't accessible from the console, it is assumed than all messages, associated with this node have been lost.

7.2.2 test node failure

If test node isn't accessible at the beginning of the test, it is assumed the network failure. Otherwise, test console will try to reach it every time it uses other nodes.

8. Analysis

8.1. Scalability

[This sub-section describes how the component reacts to the variation in input and configuration parameters: number of nodes, threads, requests, locks, utilization of resources (processor cycles, network and storage bandwidth, caches), *etc.* Configuration and work-load parameters affecting component
behavior must be specified here.]

8.1.1 network buffer sharing

Single buffer (except timestamp/sequence number fields in the test message) can be shared between all bulk send/receive operations.

8.1.2 statistics gathering

For a few tens of nodes pdsh can be used - scalability issues does not exists on this scale.

8.2. Other

[As applicable, this sub-section analyses other aspects of the design, *e.g.*, recoverability of a distributed state consistency, concurrency control issues.]

8.3. Rationale

[This sub-section describes why particular design was selected; what alternatives (alternative designs and variations of the design) were considered and rejected.]

pdsh was chosen as an instrument for starting/stopping/gathering purposes because of a few tens of nodes in the test. At large scale something else must be used.

9. References

[References to all external documents (specifications, architecture and requirements documents, *etc.*) are placed here. The rest of the document cites references from this section. Use Google Docs bookmarks to link to the references from the main text.]

[0] Colibri Summary Requirements Table
[1] HLD of Colibri LNet Transport
[2] Parallel Distributed Shell
[3] Round-trip time (RTT)

Requirements for the first milestone

Date: Tue, 14 Feb 2012 22:01:01 -0700
From: Isaac Huang <isaac_huang@xyratex.com>

To meet the first milestone, I think all the management mechanisms can be omitted. E.g. mapping from clients to servers, buffer management on server, gathering stats, coordinating client operations, and protocol versioning.

These are only useful at larger scale. For a demo involving just a few tens of nodes or less, most such mechanisms can be manually accomplished by 'pdsh' and some simple shell programming. Coordination issues only become a concern at much larger scale.

Something like this should suffice:

```
servers="s0 s1 s2"
clients="c0 c1 c2 c3 c4"
```

```
pdsh $servers bulk_server -s 1M -c 32 # 4 clients, 8 concurrent writes each
pdsh $clients bulk_write -s 1M --loop 4096 --target $servers
```

bulk_server and bulk_write are binaries that implement the bulk transfer test. For kernel space:

```
pdsh $servers modprobe bulk_server size=1M count=32
pdsh $clients modprobe bulk_write size=1M loop=4096 target="$servers"
```

Stats are printed alive to stdout. Many variables can be hard coded. E.g. to saturate most low latency IB links, 8 concurrent 1M transfers should suffice, so test concurrency can be hard coded as 8 and bulk transfer size can be hard coded as 1M.

Essentially we're only implementing the tests without all the support infrastructure. When we work on the infrastructure later, the code for the tests can be reused.

- Isaac