High level design of rpc layer core

By Nikita Danilov <nikita_danilov@xyratex.com>
Anatoliy Bilenko <anatoliy_bilenko@xyratex.com>
Date: 2011/02/18
Revision: 1.0

[Text in square brackets and with a light-green background is a commentary explaining the structure of a design document. The rest is a fictional design document, used as a running example.]

This document presents a high level design (HLD) of rpc layer core of Colibri C2 core. 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.]

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

RPC layer is a high level part of C2 network stack. It contains operations allowing user to send RPC items, containing *file operation packets (FOP)* to network destinations. *RPC layer* speaks about *FOPs* being sent from and to *end-points* and *RPCs*, which are containers for *FOPs*, being sent to *services*.

RPC layer is a part of a C2 server as well as a C2 client.

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.]

- fop, *file operation packet,* a description of file operation suitable for sending over network and storing on a storage device. File operation packet (FOP) identifies file operation type and operation parameters;
- rpc, is a container for fops and other auxiliary data. For example, addb records are placed in rpcs alongside with fops
- service, process, running on an endpoint, allowing to execute user requests.
- endpoint (not very good term, has connotations of lower levels), host on which service is being executed.
- message, communication mechanism, allowing to send requests and receive replies.
- network address, IP address.
- session, corresponds to network connection between two services.
- (update) stream, is established between two end-points and fops are associated with the update streams.
- slot, context of update stream.
- replay
- resend
- action (forward fop, *ylper*?), operation of RPC item transition from client to service.
- reply (backward fop), operation of RPC item transition from client to service with results of *"ylper"* operation.
- RPC-item, item is being processed and sent by RPC layer.
- fop group

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.rpccore.flexible] supports synchronous and asynchronous communication;
[r.rpccore.flexible] supports priorities;
[r.rpccore.flexible] supports sending various auxiliary information;
[r.rpccore.efficient] the network bandwidth is utilized fully by sending concurrent messages;
[r.rpccore.efficient] the messaging overhead is amortised by issuing larger compound message;
[r.rpccore.efficient.bulk] 0-copy, if provided by the underlying network transport, is utilized;
[r.rpccore.eos] support ordered exactly once semantics (EOS) of delivery;
[r.rpccore.formation.settings] support different setting like *max_rpc_in_flight, max_page_per_rpc, etc.*

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.]

To achieve the requirement of flexibility and efficiency design is based on multi-threaded principles. Parameters of efficiency, like thread count, network utilization, CPU utilization should be controlled on fly.

4. Functional specification

RPC core component (fig. 1) can be decomposed into the following subcomponents:

- RPC processing;
- Connectivity;
- Statistics.

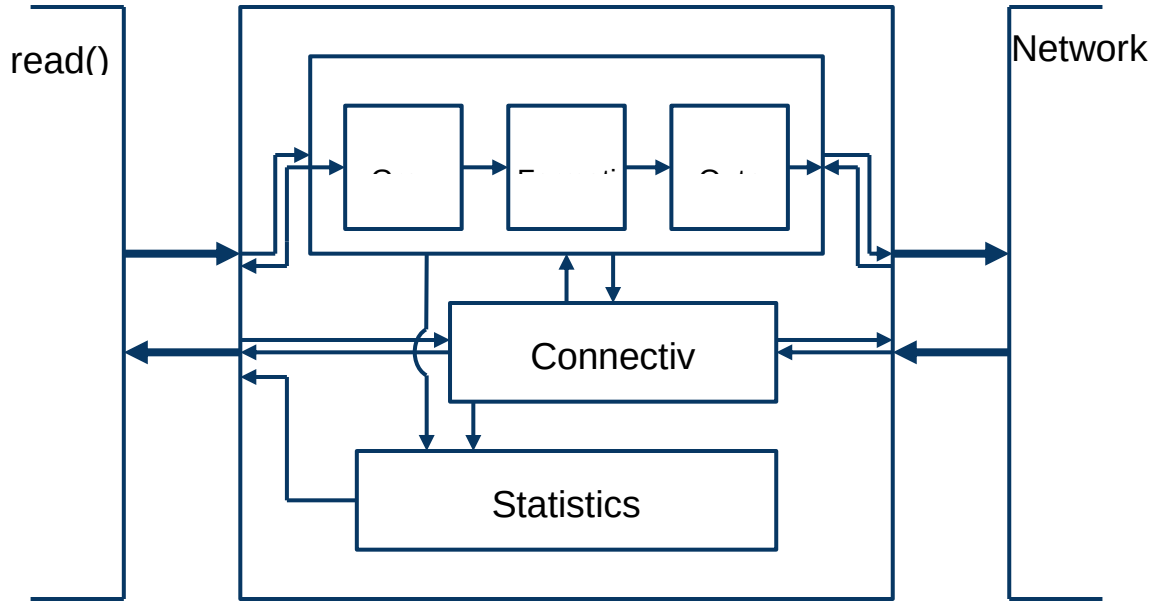

Fig. 1 – RPC core component data flow diagram.

Listed subcomponents share RPC component state e.g. RPC-items, connections, slots, sessions, channels, etc.
"RPC processing" is a subcomponent which is responsible of the following actions:

- *Grouping* input items (FOPs, ADDB, etc.) into containers (RPCs). Grouping subcomponent performs grouping of input items and produces the RPC items cache. Items are grouped by the following criteria:
  - By the endpoint item (FOP, ADDB) is following to (sessions);
  - By priority (assigned via ext. interface).
- *Formation* subcomponent proceeds with classification by some criteria and produces RPCs which are containers of RPC items. It is responsible for data preparation in an order that is optimal for network utilization. The following options are used:
  - Do grouping by fid of file;
  - Maximal count of RPCs in processing (C2_MAX_RPCS_IN_FLIGHT);
  - Maximal count of pages per RPC (C2_MAX_PAGES_PER_RPC).

  Further description of formation is out of the scope of this document and described in formation HLD [4].
- The main goal of *output* subcomponent is to communicate with networking layer optimally by producing *update streams* associated with *slots*. The main method here is to control the level of concurrency according to gathered statistics to influence network utilization level. It's also possible to control resources utilization via external interface (discussed below). Output subcomponent is also producing RPCs for bulk-transfers converting incoming RPCs from formation subcomponent if possible.

"RPC processing" exposes the following interfaces:

- Send RPC items (FOP and ADDBs) with:
  - Priority;
  - Caching type;
  - Synchronously/asynchronously;
  - To sevice id;
- Wait for reply items (FOPs) with the same options like in sending.
- Publisher/subscriber [6] based interface (synchronous callback) to control sequence of RPC-items transmission. User has to have an ability to subscribe on the following events:
  - Updatestream which contains given RPC-item created / deleted;
  - Given RPC-item processing started / finished;
  - etc.

Connectivity subcomponent has to provide asynchronous and synchronous functions to send/receive FOPs and bulks, and establish/close connections between *end-points* via networking layer and provide high level logic abstractions over connections by introducing sessions and slots. *Session* is a dynamically created server object, created and destroyed by client requests. It maintains the server's state relative to the connection(s) belonging to a client instance. *Slot* is created by a client request and attached to active client session. Slot is a *context* in which some operations on RPC items are being applied *sequentially one by one (*execution for FOPs, write to ADDB for ADDB-records*)*. RPC items related to different slots can be processed concurrently.
Connectivity exposes interfaces:

- Establish/close connection to end-point and return session.
- Create/destroy slot.
- Asynchronous and synchronous RPC-item transmitting functions.
- Specify properties of subcomponent like *max slots* existing simultaneously.

Further description of connectivity is out of the scope of this document and described in sessions HLD [5].

Statistics subcomponent is used to gather and share various statistics of RPC layer. This can be different vital and miscellaneous information regarding RPC component state and other properties. The following properties look to be important for RPC core component user:

- Items in caches and their count;
- Items in formation stage;
- RPCs count in output stage;
- Count of processing threads and their state;
- Number of opened sessions and slots;
- Time related statistics:
  - Min/max/average time of RPC-item processing;
  - Utilization of various resources (% of max network bandwidth used);
  - Usage ranking statistics (79% of RPCs were high-priority, 4% RPCs came from client X, etc.);
  - RPCs per second;
  - Some units measured in bytes/sec or pages/sec;
  - Etc.
- Some constant and slow varying properties like C2_MAX_PAGES_PER_RPC or

C2_MAX_RPCS_IN_FLIGHT.

Statistics subcomponent expose simple interface which takes property name or key and returns its value in raw form (fig. 2).

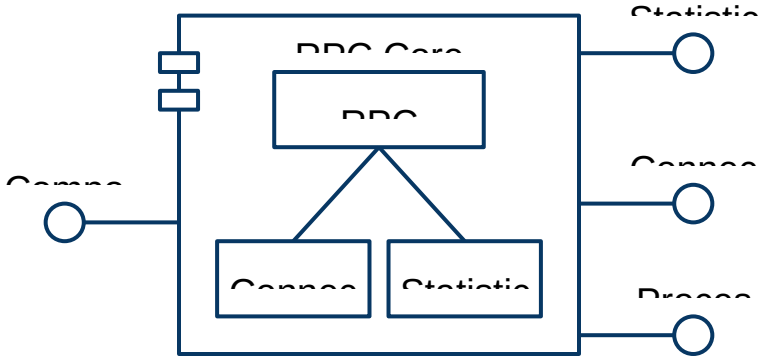

Fig. 2 – RPC component interface.

Interface description:
Component IF:

- int *c2_rpc_core_init*(...);
- void *c2_rpc_core_fini*();

Statistics IF:

- void* *c2_rpc_stat_get_prop*();

Connectivity IF:

- c2_rpc_session_id *c2_rpc_session_link*(c2_service_id);
- c2_rpc_slot_id *c2_rpc_slot_link*(c2_rpc_session_id);
- void *c2_rpc_session_unlink*(c2_rpc_session_id);
- void *c2_rpc_slot_unlink*(c2_rpc_slot_id);
- void *c2_rpc_slot_max_set*(uint32_t max);

Processing IF:

- int *c2_rpc_item_send*(c2_rpc_item*, c2_service_id, c2_rpc_prio, c2_rpc_caching_type, c2_rpc_callback*);
- int *c2_rpc_items_send*(c2_upstream*, c2_service_id, c2_rpc_prio, c2_rpc_caching_type, c2_rpc_callback*); // send with defined update stream
- int *c2_rpc_fop_wait*(c2_service_id, c2_rpc_prio, c2_rpc_caching_type, c2_rpc_callback*);
- int *c2_rpc_upstream_register*(c2_rpc_item*, c2_upstream_callback*);
- void *c2_rpc_upstream_unregister(*c2_upstream_callback**);*

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

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. 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.]

To implement high-speed RPC processing subcomponent (fig. 2) implementation should optimally utilize the resources of the host system. To perform this, Grouping, Formation and Output subcomponents should exploit multithreading. It's well known fact that CPU utilization is optimal when the number of "processing threads" is equal to CPU cores plus number of "IO-threads". That's why mentioned subcomponents should use thread pool with optimal parameters.
Output (transmitting) stage may have its separate pool, a number of threads which can vary and correlate with network statistics.

The level of parallelism can be also tuned from the outside of RPC core component. It's possible to control it by limiting number of slots being processed simultaneously (*c2_rpc_slot_max_set IF*).

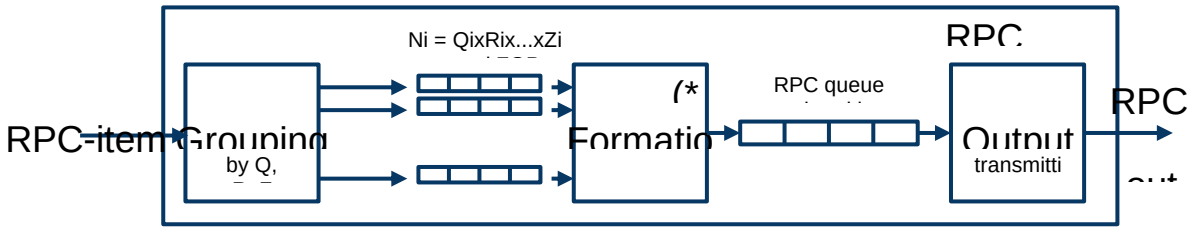

Fig. 3 – FOP Processing subcomponent organization ((*) - pool based input FOP queue items consumption, (**) – output FOP – synchronous or asynchronous callback based).

Simplification: FOP processing can be organized via pipeline multithreaded pattern[3]. It would be slower in performance aspect, but faster to implement, easier to support and it could be also reused (not insisting for some reasons).

As it was mentioned, subcomponents of "RPC processing" run in their own separate threads acquired from pool. To organize multithreaded interaction and mandatory (constraining) load balancing, "one-side" bounded queues[2] are used. Number of queues between stages and fetching policy is performance related subject and should be reviewed in DLD.

5.2. 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.]

1. Cached FOPs might have dependencies each on other. This could affect the order of fop sending. That's why formation subcomponent should analyse those dependencies and produce RPCs accordingly.

2. To implement the design shown on fig. 1-3 the Abstract Data Types shown on fig. 4 should be used (already defined in Colibri) or introduced. Fig. 4 shows relationship between these ADTs in terms of UML OMG.

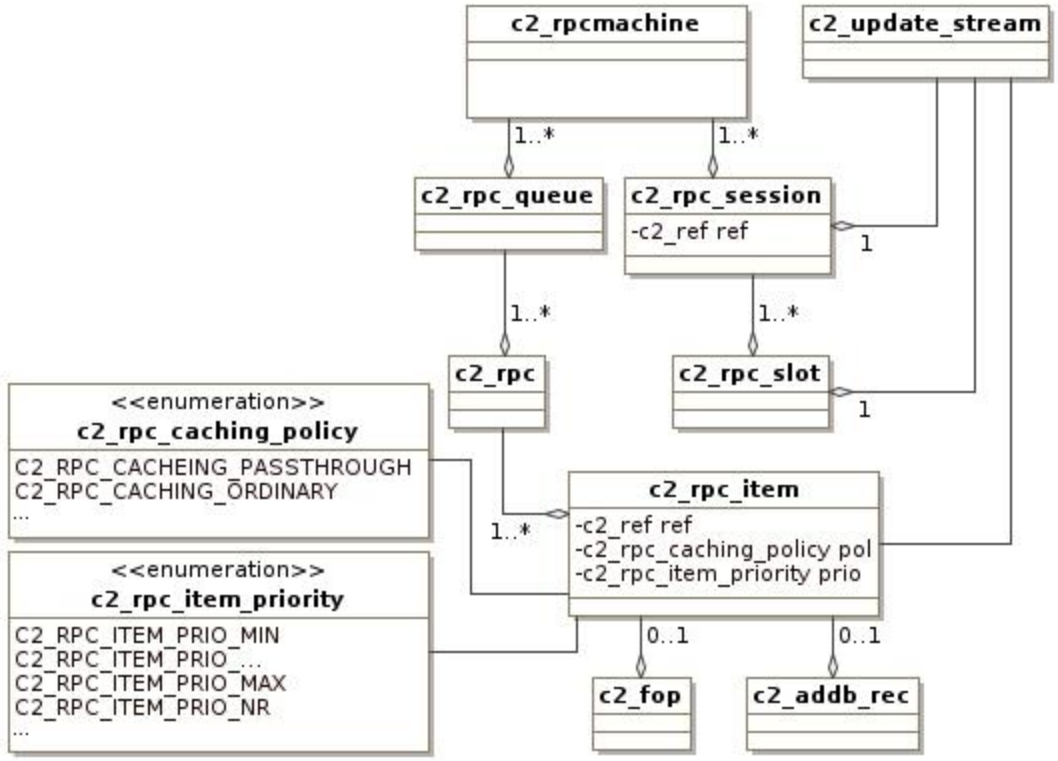

Fig. 4 – ADT relationship.

- *c2_rpc_queue* type is used to implement a queue for the staged interaction of the FOP processing subcomponent. This ADT can be based on a generic bounded queue for the first implementation, and specialized with more effective implementations for concrete case at later stages (fetching policy, locking/unlocking policy, etc.);
- *c2_rpc_session* and *c2_rpc_slot* map directly to sessions and slots described in [1,2];
- *c2_rpc* is a container of *c2_rpc_items*;
- *c2_rpc_item* is a container of *FOPs* or *ADDBs (*or something to be transmitted*)* with attributes related to *RPC* processing, like priority, caching, etc.
- *c2_rpcmachine* is a RPC processing machine, several instances of it might be existing simultaneously.

- *c2_update_stream* is an ADT associated with sessions and slots used for FOP sending with FIFO and EOS constrains.

5.3. Security model

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

5.4. Refinement

[This sub-section enumerates design level requirements introduces 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.]

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.]

Design has no explicit states. Data flow control depends on RPC-item placement in RPC queues and on some flags like caching and priority of RPC-item. The design is based on principles of *streaming processing,* where resources, like CPU, network, are allocated by a compulsory balancing scheme for each item being processed.

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.]

RPC component creates thread pool(s), service threads of which are allocated for processing FOPs and RPC items on each stage of RPC processing subcomponent. It should be possible to provide automated load balancing based on statistics of network utilization, CPU utilization, etc.

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.]

Typical scenario:
It's supposed that client handles IO read() / write(). On each such IO request a new FOP is created. These "reads" and "writes" run in a multi-threaded environment. After FOPs are built they are passed to "RPC processing". Fig. 5 shows how "RPC processing" operates in detail:

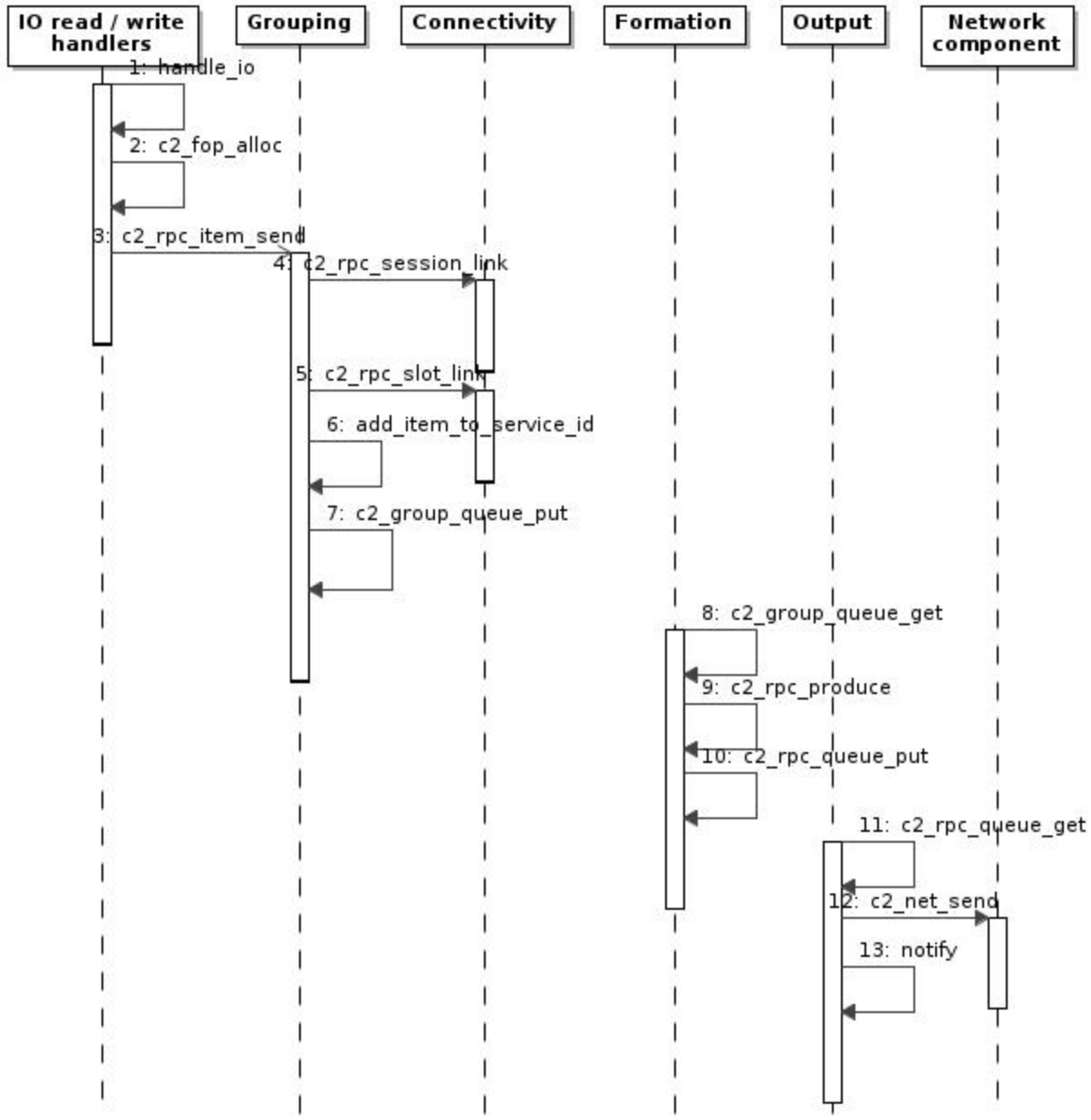

Fig.5 – Sequence diagram of c2_rpc_item_send() from start till callback notification.

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.]

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.]

It is believed that significant improvements in the RPC processing efficiency and general throughput can be achieved by limited number of threads in pool. Number of these threads should be controlled according to CPU usage and network utilization statistics, provided by statistics subcomponent and according to RPC component settings like *c2_rpc_slot_max*.

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.]

8.2. Rationale

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

9. Deployment

9.1. Compatibility

[Backward and forward compatibility issues are discussed here. Changes in system invariants (event ordering, failure modes, *etc.*)]

9.1.1. Network

9.1.2. Persistent storage

9.1.3. Core

[Interface changes. Changes to shared in-core data structures.]

9.2. Installation

[How the component is delivered and installed.]

10. 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] Networking architecture 1-pager
[1] Architecture review of rpc layer
[2] Bounded queue for MT environment
[3] Multithreaded pipeline and pipen filter (saw nicer referece somewhere...)

[4] Formation HLD
[5] Sessions HLD (will be linked when ready)
[6] http://en.wikipedia.org/wiki/Publish/subscribe

HLDIT