High level design of Mero Spiel API

By Igor Vartanov <igor.vartanov@seagate.com>
Egor Nikulenkov <egor.nikulenkov@seagate.com>
Date: 2015/02/25
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 Mero Spiel API. The main purposes of this document are: (i) to be inspected by Mero architects and peer designers to ascertain that high level design is aligned with Mero 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 Mero 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.]

1. Definitions

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

CMI - Spiel Configuration Management Interface

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.]

Spiel requirements

- **[r.m0.spiel]** Spiel library is to implement API that allows controlling cluster elements state/subordination/etc. opaquely for API user
- **[r.m0.spiel.concurrency-and-consistency]** Every CMI API call results in originating a *Spiel transaction* that is to:
  - Allow concurrent calls be originated by several API users
  - Allow keeping cluster configuration database consistent in concurrent environment
- **[r.m0.spiel.transaction]** Currently expected the transaction is to:
  - Compose entire configuration database from scratch based on the information from Spiel supplementary functionality (HALON/SSPL)
  - Post the new configuration to all instances of **confd** databases currently known in Spiel environment
  - Result in newly introduced configuration change propagation across all cluster nodes having:
    - **confd** database instances eventually consistent
    - **confc** caches eventually matching to stable **confd** database version
- **[r.m0.spiel.conf-db-abstraction]** From API user's standpoint the configuration database is treated as an abstract element tree where physical database replication aspects have to remain obscured from API user, and resolved at proper lower levels and/or with corresponding mechanisms
- **[r.m0.spiel.element-id]** A database element is identified by ID unique across the entire cluster
  - This implies that Spiel API must resolve element ID to node endpoint, where the action is to be executed, based on information initially provided as command line parameters but later updated from recent configuration transactions
  - API user remains unaware of any details related to transport level.
- **[r.m0.spiel.element-id-generation]** Spiel API user is responsible for correct Element ID generation:
  - ID is unique (see **[r.m0.spiel.element-id]**) across the entire cluster
  - Particular Element ID embeds element type ID
- **[r.m0.spiel.element-types]** Element types to be supported so far:

- Regular:
    - Filesystem
    - Node
    - Process
    - Service
    - Device
    - Pool
    - Rack
    - Enclosure
    - Controller
    - Pool version
- V-objects
    - Rack_v
    - Enclosure_v
    - Controller_v

- **[r.m0.spiel.conf-actions]** Currently, actions to be supported, per element:
  - Add
  - Delete
- **[r.m0.spiel.conf-validation]** The validity of configuration composed by a series of Spiel calls is to be tested only outside of Spiel, e.g. on **confd** side, but not in Spiel calls themselves
  - However, some level of trivial on-the-run validation is required anyway, e.g. m0_fid embedded type must match with the processed call semantically, and similar aspects.
- **[r.m0.spiel.environment]** Spiel must always make sure it runs on a consistently made list of confd servers, i.e. the passed list of confd endpoints is identical to the list of confd servers in every existent database replica
- **[r.m0.spiel.cmd-iface]** Spiel API allows changing cluster state via special operation requests to dedicated Mero service. Spiel API user is unaware of any underlying transport details.
- **[r.m0.spiel.cmd-iface-sync]** Spiel command interface is synchronous.
- **[r.m0.spiel.cmd-iface-ids]**Command action is applicable only to objects currently present in configuration database and identified by element ID.
- **[r.m0.spiel.cmd-explicit]**A configuration change, when occurs, does not imply any automatic command execution. Any required command must be issued explicitly.
- **[r.m0.spiel.supported-commands]** Commands supported so far:
  - Service
    - Initialize
    - Start
    - Stop
    - Health
    - Quiesce

- Device
  - Attach
  - Detach
  - Format
- Process
  - Stop
  - Reconfigure
  - Health
  - Quiesce
  - List services
- Pool
  - Start rebalance
  - Quiesce rebalance
  - Start repair
  - Quiesce repair

Supplementary requirements

- Resource Manager
  - **[r.m0.rm.rw-lock]** Need to introduce an additional type of shared resource Read/Write Lock
    - Read lock acquisition is required to let every confc instance access/update local cache
    - Write lock is to be acquired to start updating configuration database instances in a consistent and non-conflicting manner
    - Write lock acquisition has to invalidate already provided read locks and this way to force confc instances to initiate version re-election and updating local caches
  - **[r.m0.rm.failover]** If RM service serving configuration RW lock fails, new RM service should be chosen to serve configuration RW lock resource.
- Configuration database
  - **[r.m0.conf.replicas]** Cluster must run having more than one database instances (replicas) available at any moment, and served by a single configuration server each

- **[r.m0.conf.consistency]** Configuration database replicas at any given moment are allowed to be inconsistent under condition that one particular version number reaches a quorum across all the cluster

- **[r.m0.conf.quorum]** The number of configuration servers running simultaneously in the cluster must meet the following requirement:

  $N_{confd} = Q + A$

  where Q is a quorum number to be reached during version negotiation,
  and A is a number meeting the condition: $A < Q$
  Note: The simplest case is $A = Q - 1$, which gives $N_{confd} = 2Q - 1$

- **[r.m0.conf.transaction]** Configuration database version needs to be distributed to all known confd servers in transactional manner. Spiel client must be able distribute the version normal way, i.e. reaching quorum on the cluster, as well as forcible way, when version is uploaded to as many confd servers as possible at the moment. In the latter case the forcible version upload must allow multiple attempts for the same transaction dataset.

- **[r.m0.conf.attributes]** Configuration database version needs to be identified by the following attributes:

  - Version number, incremented from update to update
  - Transaction id, generated uniquely across the entire cluster
  - Epoch (to be decided later)

- Configuration server (confd)

  - **[r.m0.confd.update-protocol]** Two-phase database update protocol, LOAD (long but non-blocking) preceding FLIP (short and blocking) need to be supported to minimize blocking effect on the entire cluster

- Configuration client (confc)

  - **[r.m0.confc.rw-lock]** The new lock is to be adopted by confc implementation where applicable

  - **[r.m0.confc.quorum]** Client needs to poll all known configuration servers to find out a configuration version reaching quorum Q in the cluster right now. The quorum version must be the only one used in process of serving configuration requests from Mero modules

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.]

Spiel roughly falls into 2 types of interfaces: Configuration management interface (CMI) and Command interface (CI). Logically these interfaces provide different modes of operation.

CMI calls series must explicitly form a transaction. CI calls does not require any transaction mechanism.

4. Functional specification

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

4.1. Configuration database

Configuration database format remains as is. But a new configuration path must be introduced to represent current database attributes:

- Version number
- Quorum Q
- Epoch (??)

TBD: A possible option would be storing the attributes as a metadata, i.e. not in configuration database but somewhere outside the one. In this case, an extra call to confd must be introduced to let version attributes be communicated between confc and confd.

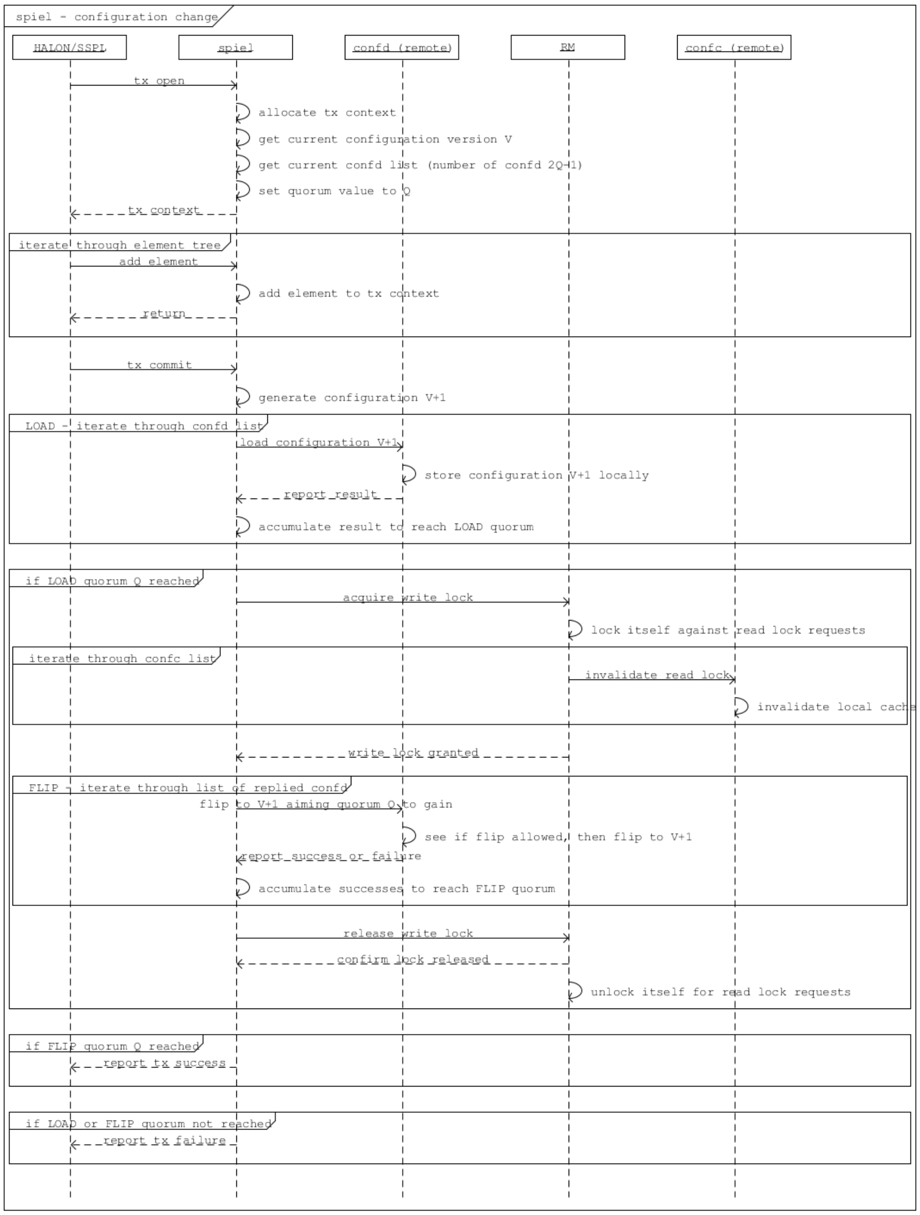
Configuration change

Spiel client is able to compose new database from scratch using information on the cluster configuration stored/collected outside of Mero.

Note: Collecting configuration information is out of Spiel responsibility either.

[i.m0.conf.transaction] The whole process is guided by Spiel library and done in transactional manner. Spiel client is responsible for explicit transaction opening, composing new configuration database tree and committing transaction. Transaction commit opaquely for Spiel client does all the required communication, version negotiation and distribution.

The same opened transaction may be committed either normal way or forcibly. Normal way implies that to succeed the transaction dataset must be uploaded to a number of confd servers reaching quorum or greater than that. Forcible transaction committing succeeds always no matter how many successful uploads were actually done. The same transaction dataset being repeatedly committed is uploaded to all known confd servers currently available in the environment.



TBD: Potential issues identified to the moment:

- Read lock invalidation is too slow or fails with some credit borrower
    - This may block write lock acquisition
- Write lock acquired but never released due to unrecoverable communication failure or Spiel client death
    - This may cause all the cluster to lock out, as no confc is going to be able to complete reading configuration data

Note:

[i.m0.spiel.environment] To comply with **[r.m0.spiel.environment]** requirement the transaction must end with spiel making sure its confd list it was initialised with is identical to the list of confd contained in the newly distributed quorum version. Or transaction is to make sure at the very beginning it starts with the list of confd identical to the latest known list of ones.

Otherwise the requirement seems excessive, and should not be imposed at all, having only Spiel confd initialisation list to rely on.

4.2. Configuration client

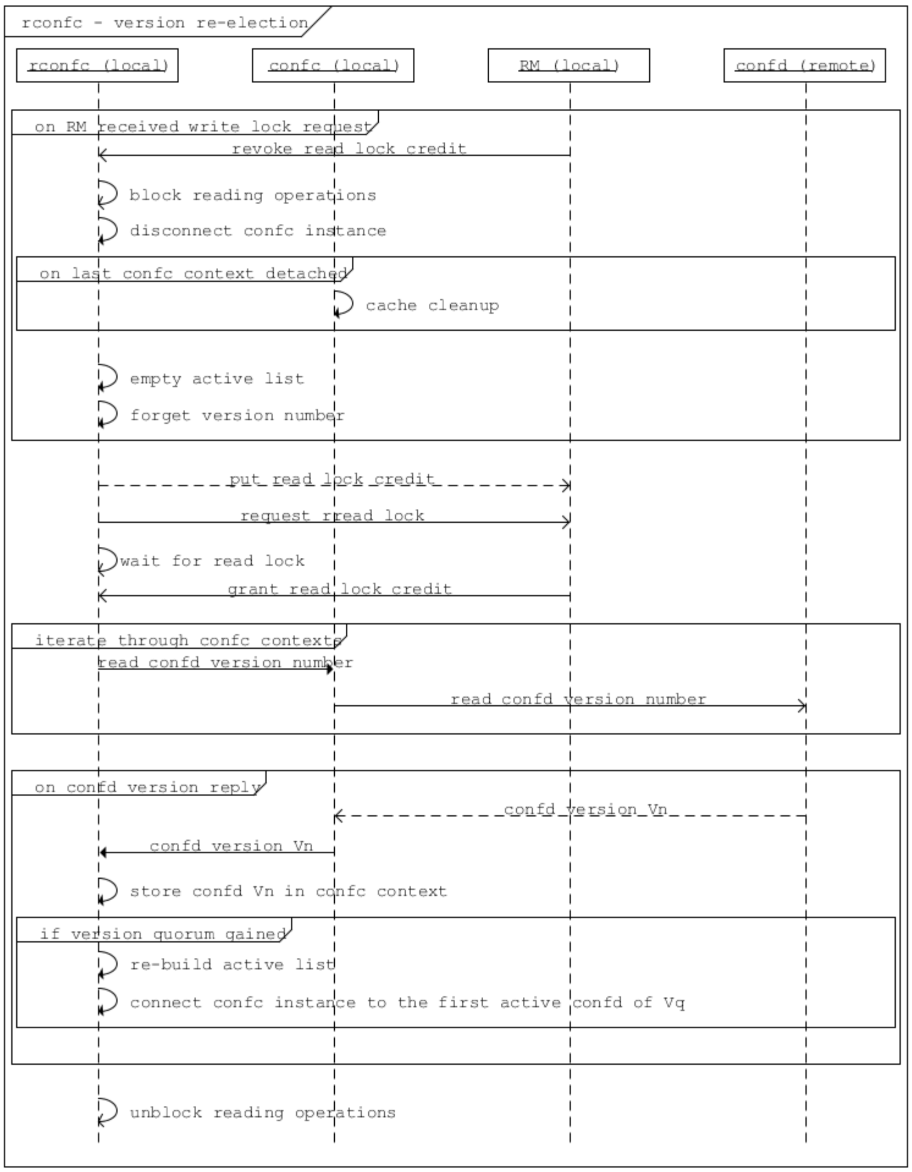
Cache refresh

Roughly, refreshing looks like gathering current version numbers all the known confd servers currently run, deciding on what version number reaches the quorum, and further loading requested configuration data from the confd server currently assigned to be active.

Note: Read lock acquisition is not a standalone call, but in accordance with current RM design it is hidden inside every 'get' operation issued to confc cache. The diagram just highlights the fact of mandatory lock acquisition to let operation complete.

Version re-election

Re-election comes in effect either on client's very start, or when read lock revocation detected.


rconfc - version re-election
rconfc (local)
confc (local)
RM (local)
confd (remote)
on RM received write lock request
revoke read lock credit
block reading operations
disconnect confc instance
on last confc context detached
cache cleanup
empty active list
forget version number
put read lock credit
request rread lock
wait for read lock
grant read lock credit
iterate through confc contexts
read confd version number
read confd version number
on confd version reply
confd version Vn
confd version Vn
store confd Vn in confc context
if version quorum gained
re-build active list
connect confc instance to the first active confd of Vq
unblock reading operations

Configuration client emits asynchronous requests to all currently known confd servers aiming to collect their configuration database version numbers. When some version reaches a quorum, it becomes a working (active) version until next re-election.

TBD: In case the required quorum was not reached, the node appears inoperable having no active confc context to communicate over. This sort of ambiguity must be solved somehow, or reporting to HA may take place as a fallback policy.

Note: Ideally, the process of version re-election must end with rconfc updating its list of known confd getting the one from the newly elected version Vq, as the list may appear changed.

TBD: Quorum number Q must be re-calculated either, immediately or on next re-election start.

RCONFC - Redundant configuration client

To work in cluster with multiple configuration servers the consumer (Mero module) makes use of redundant configuration client (rconfc). The client, being aware of the multiple confd instances, is able to poll all the confd servers, find out the version number of configuration databases the servers run and decide on the version to be used for reading. The decision is made when some version reaches a quorum. The rconfc carries out the whole process of version election providing communication with confd instances.

When appropriate version elected, rconfc acquires read lock from RM. Successful acquisition of the lock indicates that no configuration change is currently in progress. The lock remains granted until the next configuration change is initiated by some Spiel client.

> **Note:**
>
> Rconfc provides no special API for reading configuration data. Instead, it exposes a standard confc instance the consumer is to deal with standard way. When rconfc initialisation succeeded, the confc instance is properly set up and connected to one of confd servers running the elected version.

When read lock is acquired, the rconfc sets up its confc by connecting it to a confd instance running the version that reached quorum. Rconfc remains in control of all confc contexts initialisations, blocking them when the read lock is being revoked.

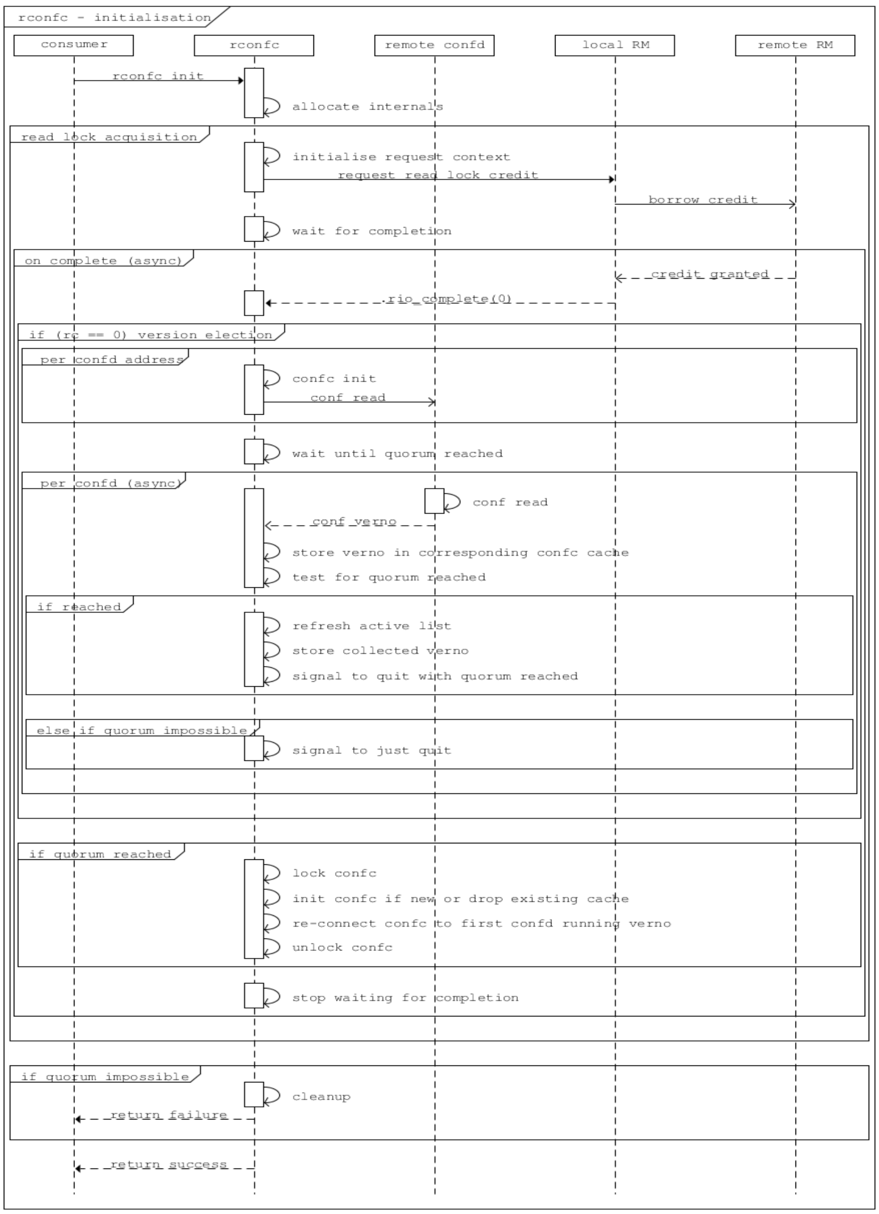
Having confc context initialised, consumer is allowed to conduct reading operations until the context finalisation. Rconfc is not to interfere into reading itself. When confc cache cleanup is needed, rconfc waits for all configuration objects to be properly closed and all configuration contexts to be detached from the corresponding confc instance. During this waiting rconfc is to prevent all new contexts from initialisation, and therefore, attaching to the controlled confc.

Commented [1]: Point for discussion: Important to mention here that with the suggested design a consumer behavior may provoke deadlocks in rconfc: in case new confc context is to initialise while having some contexts still not finited, and the code flow is linear, consumer appears deadlocked being able neither to initialise new context nor release already inited ones.

Note:

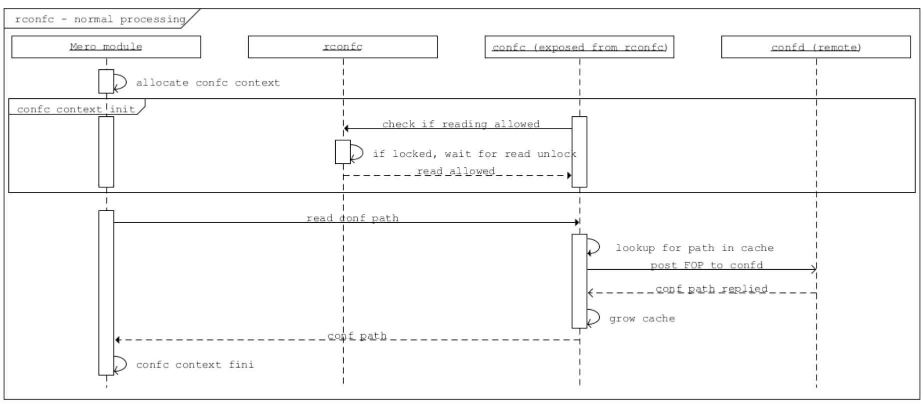
Confc instances participating in version election are never blocked by rconfc.

RCONFC Initialisation



RCONFC initialisation starts with allocation of internal structures. Mainly those are quorum calculation context and read lock context. Read lock is requested on the next step, and once successfully acquired, is continued by version election. On this step all known confd servers are polled for configuration version number. Based on the replied values a decision is made on the version number the consumer is going to read from in subsequent operations. With this version number a list of active confd servers is built, and the confc instance, hosted and exposed by the rconfc instance, is connected to one of those servers. Being successfully connected it appears ready for reading configuration data.

Normal processing

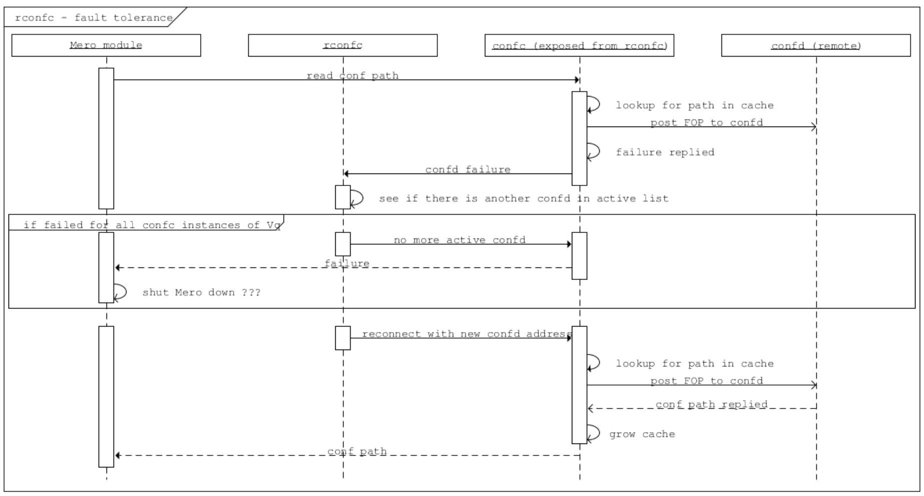


After successful rconfc initialisation the consumer (Mero module) operates with configuration client exposed from rconfc. The client reads the requested path usual way starting with initialising configuration context.. Internally the operation starts with confc referring to rconfc if read operation allowed at the moment, then when rconfc is not locked, the context appears initialised, and consumer goes to reading configuration path from the confc's cache. In case the path is not in the cache, the corresponding confd is requested for the data.

In case when rconfc is locked for reading due to read lock revocation occurred because of configuration change and/or new version election, the configuration context initialisation appears blocked until the moment of rconfc unlock. Multiple configuration context initialisations done simultaneously on the same confc will be waiting for the same rconfc unlock.

Fault tolerance

Robustness is based on the configuration redundancy and provided by the ability of rconfc to switch its confc dynamically among confd instances without need in invalidating any currently cached data, because each time the switch is done to confd running the same configuration version.



Read operation from selected confd may fail on some reason. In this case configuration client must be passed by rconfc through the list of confd addresses pointing to the same quorum version Vq of configuration database and tried to request the respective confd servers one by one until getting to success or end of the list.

In the latter case the call returns failure to Mero module originated the call.

> **Note:** It is up to module what scenario to choose to properly react to the failure.
>
> It might decide to wait and re-try to accomplish the call, or notify HA about the failure(s) and do nothing, or initiate the caller's Mero node shutdown, etc.

Resilience against top-level RM death

Requirements:

As long as RCONFC workflow relies on proper handling (getting/releasing) Conf Read Lock between client and remote RM, in case of remote RM's death there must be a way for RCONFC instance for the following:

- detect the fact of RM death
- properly handle (drop in timely fashion) all previously borrowed resource credits on client side
- initiate RCONFC restart routine
- in the course of the latter one, discover an endpoint of top-level *online* RM newly designated by HA as well as refresh the set of CONFD services currently expected to be up and running
- based on the endpoints obtained from cluster, conduct RCONFC restart and re-borrow the required credits

Assumptions:

- Two-tier RM architecture is in effect. (is it? does it matter???)
- Conf database:
    - profile includes a set of CONFD + top-level RM services sharing same endpoints with similar services in all other profiles.
        - The point is: no matter what profile client runs in, the set of CONFDs+RMs endpoints remains the same, and the same *online* top-level RM endpoint is announced among profiles' clients.
- RM service:
    - no matter local or remote one, is aware of HA notification mechanism, and smart enough to handle on its own any node death that may affect the balance of credits/loans.
        - Therefore, RM client side is responsible for a limited number of aspects to take into consideration, and must not care about remote service side.
    - running top-level, is present in configuration on every node running CONFD service, this way making the cluster run redundant set of top-level RM services remaining in *transient* state all but one chosen by HA and promoted to *online* state.
    - serves simultaneously all requests for all known resource types
        - this may be a subject for change in future design when RM service may be configured for serving some particular resource type(s) requests only
- RM client:
    - initially is unaware of any CONFD/RM endpoints
    - on (re)initialisation, is responsible for discovering from cluster:
        - the set of CONFD endpoints to connect its conf clients to
        - current online RM and guaranteeing this only RM is to be used until getting explicit death notification

    - is expected to be aware of HA notification mechanism and subscribe to HA notifications whenever required
- HA service:
    - is responsible for nodes' HA status delivery to every part constituting Mero cluster. This implies, clients are to be notified the same way as the servers currently are.
    - is responsible for detection of current top-level RM death, and designating in place of just died one a new one from the rest of *transient* RMs.
    - notifies all known cluster nodes, currently reachable ones, about death of particular **process** as well as all its **services**.
        - It is up to HA about the logic behind the notion of 'dead process'
            - This aspect may be a subject for future change, as conflict between 'dead service' and 'dead process' may cause some ambiguity in understanding and evaluating HA state for particular conf object. *The simplest resolution of the ambiguity might be considering 'process is dead' when having at least one 'service is dead' state.*
        - The process fid is included into notification vector
            - This may require current HA state update mechanism to be revised in the aspect of updating **process** conf object status
        - Process' service fids are included into notification vector
    - is responsible for announcing a new epoch (when? under what circumstances?) to disable processing requests came from the past

Questions, points for consideration:

- Should *transient* RM care about its current HA state and actively reject client requests?
- Should formerly *online* RM care about its current HA state and start actively reject client requests once being announced dead?
    - **Note:** HA may decide to announce dead a node that actually is alive but suffered from degradation in performance or network environment.
- Conf database versions are expected to be governed based on the same set of CONFD service endpoints, no matter what profile conf clients run in. So should be with top-level RM service used for Conf Read-Write Locks. But this scheme may appear inapplicable when we are talking about other resource types.
    - So the question of **resource-type dependent RM service/endpoint** still requires for additional consideration.
    - So does the question of relationship between **RM service** and **conf profile**.

RCONFC initialisation details

HA session is established when module setup takes place. The session is kept in globally accessible HA context along with the list of confc instances. Client is allowed to add an arbitrary confc instance to the list, and further HA acceptance routine is to apply standard way the received change vector to every instance in the list.

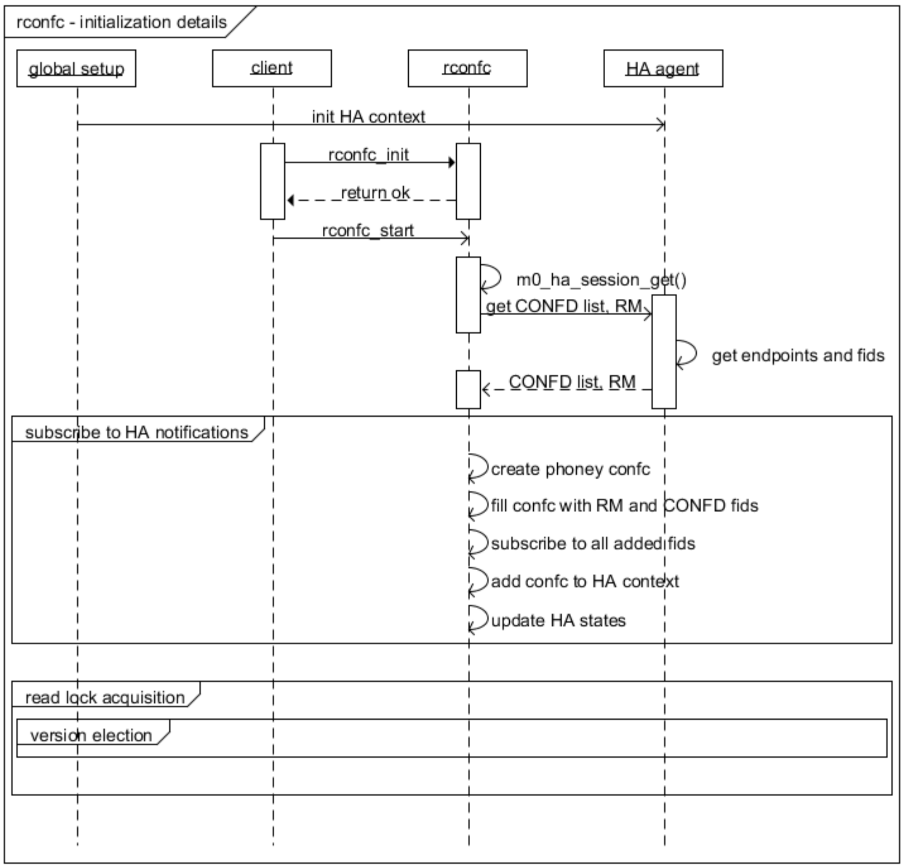
When RCONFC gets started, it obtains an already established session to HA and sends request about information regarding current set of CONFD servers and top-level RM server. The returned information is expected to include the mentioned servers' endpoints as well as fids.

RCONFC stores the server endpoints internally to use those in subsequent calls. And received service fids are put into special 'phoney' instance of **confc**, while the **confc** is added to HA context. RCONFC subscribes to all **confc** objects placed into the **confc** instance cache.

Note: The **confc** cache is to be artificially filled with the fids of interest, but not standard ways. The reason is that at the moment RCONFC has no quorum, therefore, does not know what CONFD server to read from.

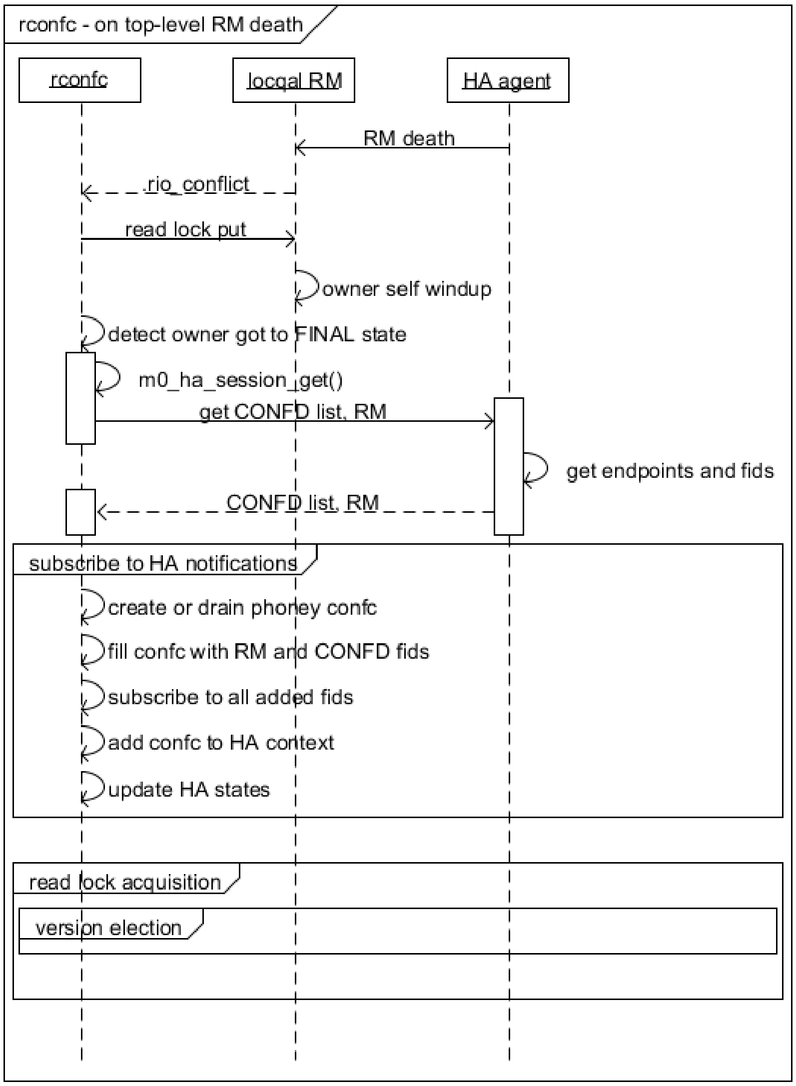
Local RM is to use the said **confc** instance for its owner's subscription as well.

With the subscription successfully done RCONFC continues its start routine with getting read lock and starting version election.



RCONFC behavior on top-level RM death

When HA notification about top-level RM death comes, local RM performs standard processing for creditor's death case. That implies, RCNOFC's resource conflict callback is called, which makes RCONFC put the held read lock. At the time RCONFC detects that its credit owner object got to final state, and after that RCONFC invokes 'start' routine, i.e. queries HA about current CONFD and top-level RM servers, re-subscribes in accordance with the most recent information and goes on with read lock acquisition and configuration version election.



Formatted: Font: (Default) Arial, (Asian) Arial, 11 pt, Font color: Black, Complex Script Font: Arial, 11 pt

4.3. Configuration server

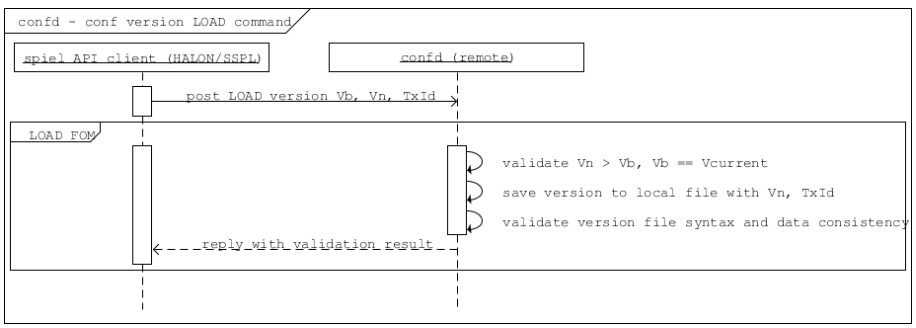
Configuration server remains operating as before, i.e. on locally stored configuration database snapshot. However, the server is able to update the database file being explicitly instructed to do that.

Configuration files are placed on server side as files in IO STOB domain. Folder STOB domain equal folder current configure FID file consists of two version number - old, new and TX ID.

The update procedure is implemented by introducing two-phase protocol.

Phase 1: LOAD

The command semantic implies the new database file is to be just locally stored without any interference with currently active version.



where:

Vb	– base version number, the number recently read by confc when reaching quorum
Vn	– new version number, the number generated by Spiel on transaction start, expected to be Vmax + 1 (maximum number reported by confc contexts plus one)
Vcurrent	– current version number stored on confd, normally equal to Vb

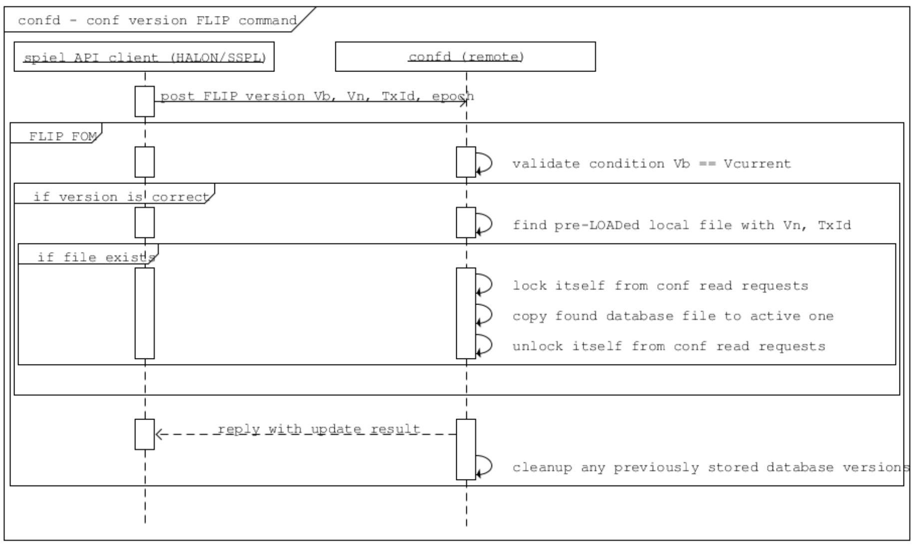
Though, minimal database validation is required before saving it to local file.

During this command execution the server is able to serve any read requests from clients.

TBD: Any dead-end situations like being lack of drive free space, etc. are to be resolved some way, most probably by reporting to HA about being unable to operate any further, and possibly confd leaving cluster.

Phase 2: FLIP

The command explicitly instructs to find a previously loaded particular version and put it in effect.



4.4. Resource Manager

Resource manager is used to control access to the contents of confd configuration database replicas

There are mainly three types of configuration database users that operate concurrently: confc, confd, spiel client. Confc makes read-only requests to confd. Confd maintains configuration database and handles read/write requests to it. Spiel client issues both read and write requests to confd.

In order to serialize access to confd configuration database new read/write lock (**RW lock**) resource type is introduced. One RW lock is used to protect an access for all confd configuration databases.

RW Lock

The distributed RW lock is implemented over Mero resource manager. Distributed RW Lock allows concurrent access for read-only operations, while write operations require exclusive access. Confc acquires read lock and holds it to protect its local database cache. Confd does not acquire RW lock by itself, but confd clients acquire RW lock instead. Spiel client acquire read lock on transaction opening to reach the quorum and acquires write lock in order to accomplish FLIP operation.

RM service failover

RM service controlling access to configuration database is essential for proper functioning of the whole cluster and becomes single point of failure. The following algorithm addresses this issue:

1. There are N_{rms} RM services starting automatically at cluster startup. $N_{rms}= N_{confd}$.
2. Only one RM service is active at any point in time.
3. There is no synchronisation between RM services. While one is active, tracking credits for RW lock, others are idle and do nothing.
4. HA is responsible for making decision about active RM service failure.
5. When active RM service fails, HA chooses new active RM service and notifies all nodes about the switch to new RM service.
6. On reception of "RM service switch" HA notification, every node drops cached credits for configuration RW lock and reacquires them using new RM address.

HA decision about RM service failure is based on notifications that are sent by mero nodes encountering RPC timeouts during interaction with RM service. The decision about the next RM instance to be used is done by HA. This decision is provided through general HA object state notifications interface. In order to switch to new version consistently across the cluster rm services states are maintained the following way:

1. Initially states of all, but one, RM instances are set to TRANSIENT, the remaining instance being ONLINE.
2. Then, each state transition notification fop, which brings an RM service to FAILED state, simultaneously moves another RM instance to ONLINE state, maintaining an invariant that no more than one RM instance is ONLINE at any time.

If all RMs are FAILED, then it is a system failure.

Since currently active RM address should be known prior to accessing configuration database, it is provided explicitly to configuration database users (command-line parameter for m0d, initialisation parameter for spiel).

Spiel is initialised with active RM service address and then maintains this address internally, changing it on HA updates if necessary.

HA guarantees that it doesn't change configuration database until all notifications about switching to new RM service are replied. That prevents situation when configuration database changes and some part of clients works with it through new RM service and other part works through RM service considered failed. If some node doesn't reply for switch notification, then it should be considered FAILED by HA.

Commented [2]: What exactly is this about? Rconfc clients?
Will there be any issue with canceling credits when there is no more top-most RM to complete the cancellation?

Commented [3]: Yes, it is about rconfc. About cancelling credits more research is needed.

Commented [4]: Again this seems true for rconfc clients, right?

Commented [5]: rconfc and spiel.

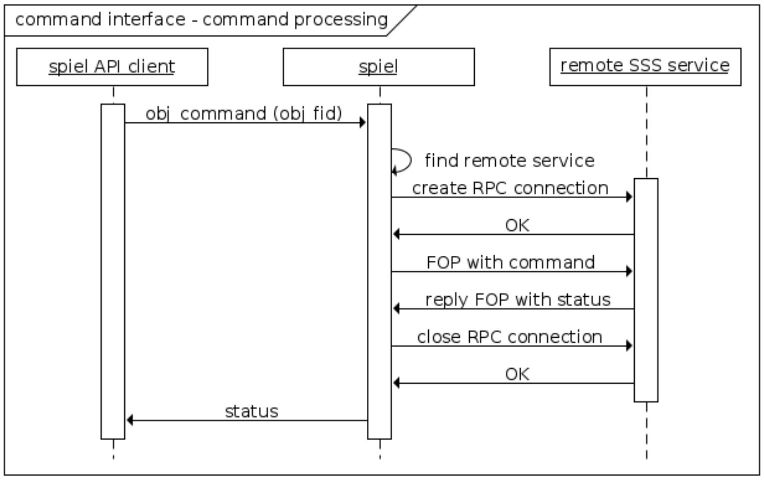
Commented [6]: This may be a cause for other sort of cluster freeze. I believe, HA should not be stuck on some unreasonably long waiting, and claim unresponsive nodes being offline from now on.

Commented [7]: Yes, I think HA should do exactly this. Updated.

4.5. Command interface

4.5.1. Common part

Command interface provides an ability to change cluster state by applying actions to the objects stored in configuration database. Command interface performs actions by sending operation requests (FOPs) to the dedicated service on remote node and waiting for reply synchronously. Dedicated service RPC endpoint is determined using the information presented in the configuration database. Start-stop service (SSS) service will play role of dedicated service in current implementation.



Note: Some commands return not only status, but some other information as well. For example "list services" command for process object returns status of the command and list of services.

4.5.2. Device commands

Device command: attach, detach and format. Target service for these commands - SSS service.

Current devices context was loaded at start Mero instance or before Load/Flip command. Device command change current status Disk on Pool machine.

Attach

Search disk item in current Pool machine. Set disk status to Online uses Pool machine API (State Transit function). If call State Transit is success then create STOB if STOB not exist yet.

Detach

Search disk item in current Pool machine. Set disk status to Offline uses Pool machine API (State Transit function). If call State Transit is success then free STOB.

Format

[This section will be written later]

4.5.3. Process reconfig command

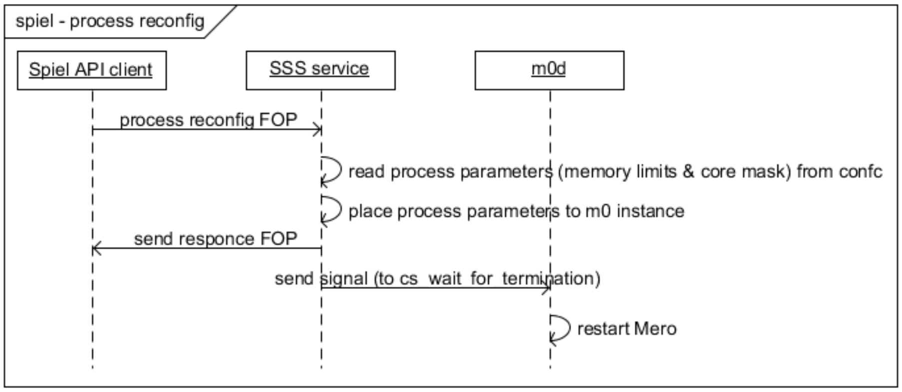
Reconfig command applies mero process configuration parameters stored in configuration database. There are two parameters for now: memory limits and processor cores mask.

Memory limits can be applied easily using *setrlimit* Linux API call in *m0_init* before initialisation of all subsystems.

Core mask specifies on which processors (cores) localities should be running. Core mask is applied through restarting localities. Localities is a part of FOM domain, which is initialised during mero initialisation (*m0_init*). So, in order to restart localities the whole mero instance should be re-initialised. That involves stopping all running services, mero instance reinitialisation (*m0_fini/m0_init*) and starting basic services again. Mero library user is responsible for proper mero reinitialisation. Reconfig command will be supported only by *m0d* mero user for now. In order to notify *m0d* that mero should be re-initialised UNIX signal is used. After reception of this signal *m0d* finalizes Mero instance and start it again.

Note that unit test can't check process reconfiguration since UT framework behaves differently than *m0d*.

Reply to for process reconfig command is sent to spiel user before actual reconfiguration is done. The reason is that during reconfiguration RPC infrastructure for sending reply is lost. Therefore reply is sent before running services are stopped.



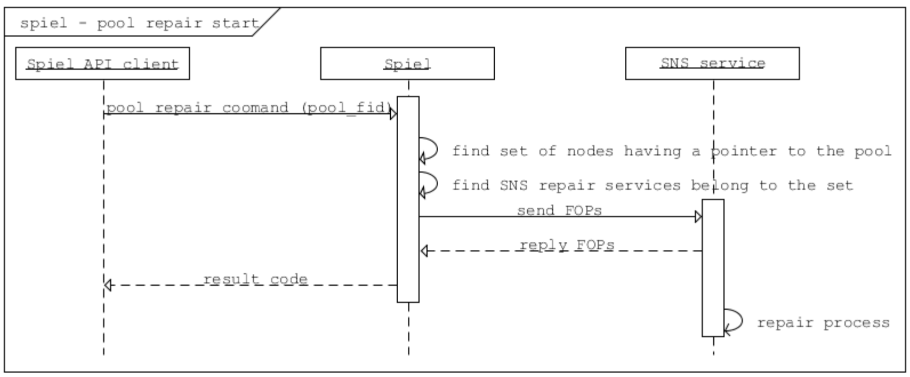
4.5.4. Pool commands

Pool repair start command

This command starts SNS repair processes on nodes related to the pool.

The algorithm is:

1. Find all nodes related to a pool in cluster configuration. A configuration node has pointer to the appropriate configuration pool. Therefore, nodes can be found by the pool fid.
2. Find SNS repair services, that belong to the nodes. Endpoints of a SNS repair service and the corresponding ioservice are the same. Thus, it suffices to find endpoints of ioservices.
3. Send a fop with REPAIR opcode to every service.
4. Once the fop received, SNS repair service sends reply fop immediately and start repair process. Spiel client is able to check status of the running repair process with "pool repair status" command.
5. Return success result code to Spiel client if every service replies with success result code or return error code if one replies with error code.



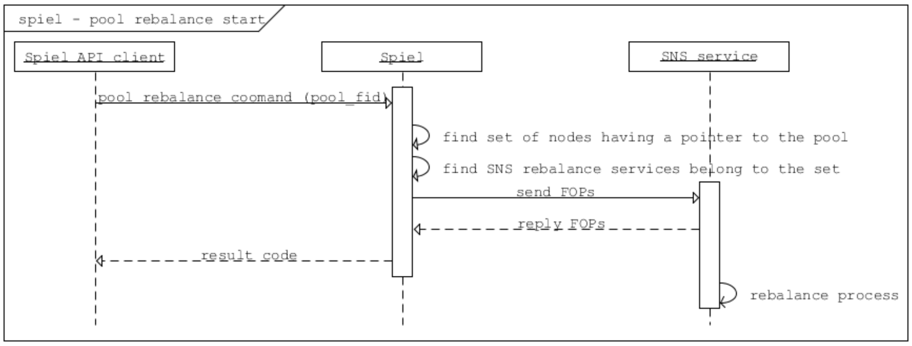
Pool rebalance start command

This command starts SNS rebalance processes on nodes related to the pool.

The algorithm is similar to SNS repair:

1. Find all nodes related to a pool in cluster configuration. A configuration node has pointer to the appropriate configuration pool. Therefore, nodes can be found by the pool fid.
2. Find SNS rebalance services, that belong to the nodes. Endpoints of a SNS rebalance service and the corresponding ioservice are the same. Thus, it suffices to find endpoints of ioservices.
3. Send a fop with REBALANCE opcode to every service.

4. Once the fop received, SNS repair service sends reply fop immediately and start rebalance process. Spiel client is able to check status of the running rebalance process with “pool rebalance status” command.
5. Return success result code to Spiel client if every service replies with success result code or return error code if one replies with error code.



Pool repair quiesce command

This command pauses SNS repair processes on nodes related to the pool.

Note: Currently, functionality of SNS repair pause or resume is not implemented. Therefore, the Spiel function returns -ENOSYS.

Pool rebalance quiesce command

This command pauses SNS rebalance processes on nodes related to the pool.

Note: Currently, functionality of SNS rebalance pause or resume is not implemented. Therefore, the Spiel function returns -ENOSYS.

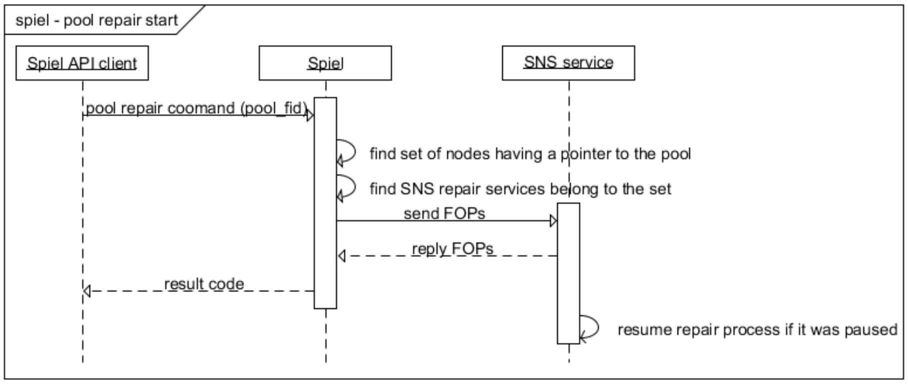
Pool repair continue command

This command resumes SNS repair process which was paused on nodes related to the pool.

The algorithm is:

1. Find all nodes related to a pool in cluster configuration. A configuration node has pointer to the appropriate configuration pool. Therefore, nodes can be found by the pool fid.
2. Find SNS repair services, that belong to the nodes. Endpoints of a SNS repair service and the corresponding ioservice are the same. Thus, it suffices to find endpoints of ioservices.
3. Send a fop with CONTINUE opcode to every service.

4. Once the fop received, SNS repair service sends reply fop immediately and resumes repair process.
5. Return success result code to Spiel client if every service replies with success result code or return error code if one replies with error code.



Pool rebalance continue command

This command resumes SNS rebalance process which was paused on nodes related to the pool, but SNS rebalance services imply instead SNS repair..

Pool repair status command

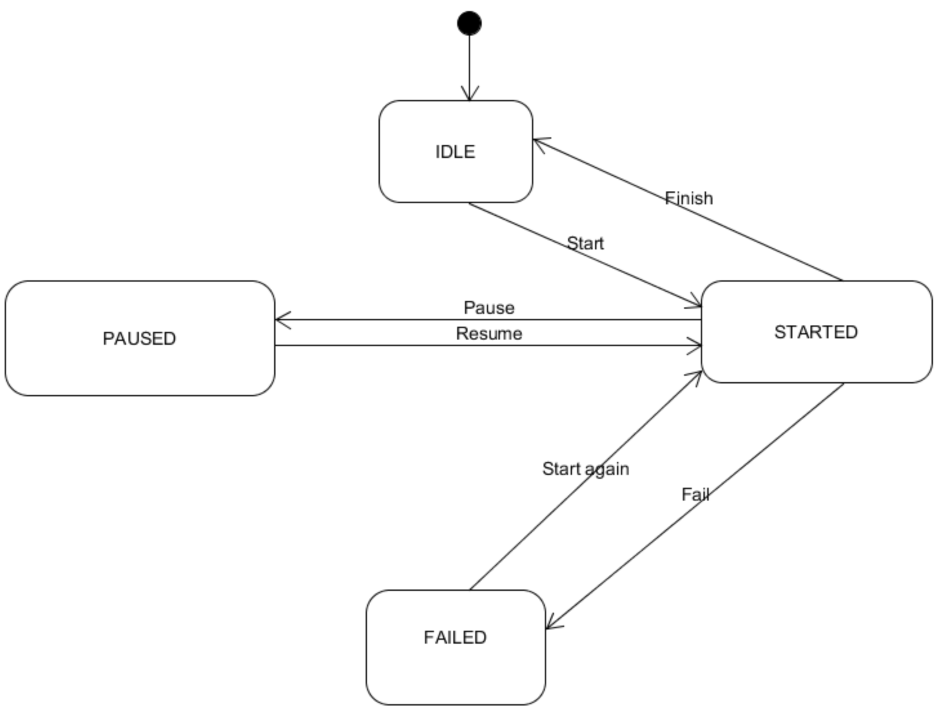
This command polls the progress of current repair process on nodes related to the pool. SNS service reply consists of two values: state of current repair process and progress in percent or number of copied bytes/total bytes or error code if repair was failed.

Commented [8]: Point to discussion

SNS repair may be in the following states: IDLE, STARTED, PAUSED, FAILED.

- The service is considered IDLE, if no running repair at the moment.
- It is STARTED, if repair is running.
- It is PAUSED, if repair was paused.
- It is FAILED if an error occurred during the repair;

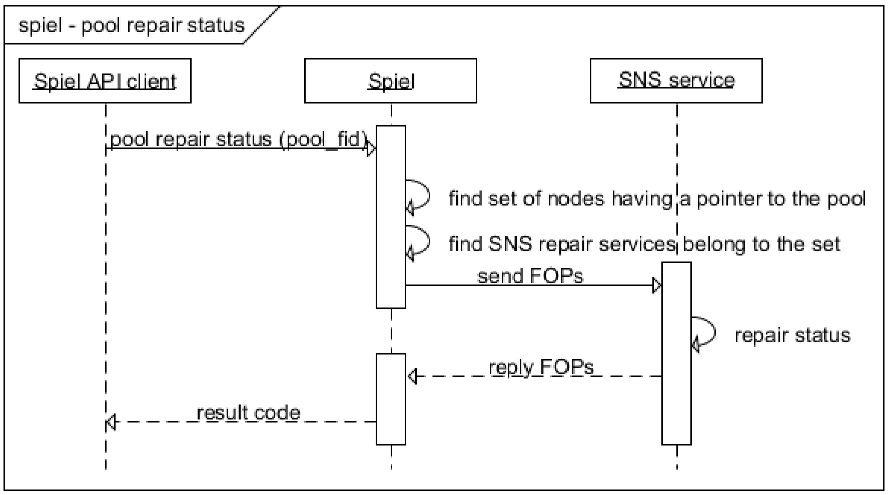
State diagram for repair status:



The algorithm is:

1. Find all nodes related to a pool in cluster configuration. A configuration node has pointer to the appropriate configuration pool. Therefore, nodes can be found by the pool fid.
2. Find SNS repair services, that belong to the nodes. Endpoints of a SNS repair service and the corresponding ioservice are the same. Thus, it suffices to find endpoints of ioservices.
3. Send a fop with STATUS opcode to every service.

4. Return success result code to Spiel client if every service replies with the progress of current repair process if it is happening.



Pool rebalance status command

This command polls the progress of current rebalance process on nodes related to the pool.

The algorithm is the same as for SNS repair, but SNS rebalance services imply instead SNS repair.

4.5.5. File System commands

This section describes commands specific to FS object.

Get FS free/total space

This command is intended to report free/total space the distributed file system provides across all its nodes. The counters are to be of **uint64_t** size. So far, only **ioservice** and **mdservice** are going to report the space counters. The counters are used as those are at the moment of querying, constituting no transaction of any sort.

To obtain the space sizes Spiel user has to always make an explicit call, as no size change notification mechanism is expected so far. Schematically execution is done as follows:

- Spiel user calls API entry providing it with FS fid

- Spiel finds FS object with the fid in configuration profile and iterates through FS node objects
- Per node, iterates through process objects
- Per process,
    - Spiel makes a call for process health
        - On correspondent SSS service side:
            - It iterates through m0 list of BE domain segments
                - Per segment
                    - Information is obtained from segment’s BE allocator object
                        - total size
                        - free size
                    - Free/total size is added to respective accumulator values
            - If IO service is up and running, it iterates through m0 list of storage devices
                - Per device object:
                    - Stob domain object is fetched, the one device operates on
                        - Information is extracted from domain’s allocator object:
                            - free blocks
                            - block size
                            - total size
                    - Total size is added to respective accumulator value unconditionally
                    - Free size is added to respective accumulator value only in case the storage device is found in the list of pool devices stored in m0 pool machine state object, and corresponding pool device state is “on-line”
            - The final accumulator values are included into service status reply
    - The call returns process status reply including free/total size values
    - Spiel accumulates the size values collected per process call
- Resultant free/total size values are returned to Spiel user

4.6. System Tests

This section defines scenarios for system tests for Spiel functionality.

Tests are to cover two major areas:

- Distributed confd management.

- Control commands.

Confd#1: normal case

- All confd instances get the same initial DB.
- Start all mero nodes.
- Validate all confc consumers are connected to confd, and loaded the same DB version.

Confd#2: split versions, with quorum

- First N confd instances get initial DB with version v1
- Remaining N + 1 confd instances get initial DB with version v2.
- Start all mero nodes.
- Validate all confc consumers are connected to confd, and loaded DB version v2.

Confd#3: start with broken quorum and then restore it

- First N confd instances get initial DB with version v1.
- Next N confd instances get initial DB with version v2.
- Remaining 1 confd instance gets initial DB with version v3.
    - Note: there's no "winner", no quorum in this setup.
- Start all mero nodes.
- Expected behavior:
    - No crashes, but
    - Cluster is not functioning.
        - Note: This behavior needs discussion; as of now behavior for this situation is not defined.
- Use spiel client to apply new DB version, v4.
    - Expected to succeed.
- Validate that all confc consumers are now unlocked, services are started, and mero cluster is ready to process requests (probably the easiest way would be to feed some I/O requests to mero and make they succeeded).

Confd#4: concurrent DB updates

- Launch N spiel clients (processes or threads).
- Each client, in a loop, tries to upload new DB version to confd.
    - This has to happen simultaneously over all clients, with minimal delays, in attempt to make them send actual LOAD/FLIP commands at the same time.
    - Run a loop for some prolonged time, to increase the chances of getting the conflict.
    - Note: This is non-deterministic test, but there does not seem to be a way within current system test framework to make this test deterministic.
- Use (N+1)th spiel client to monitor confd state. Expectations are:
    - confd remains "in quorum" (means the algo is stable against load)
    - confd version increases (means that at least some of those parallel clients succeed in updating the db -- no permanent dead-lock situations occur).

Cmd#1: control commands

- Start mero cluster.
- Use spiel commands to validate health of all services and processes.
- Restart M services.
- Reconfig N processes. (Make sure that there are untouched services, untouched processes, restarted services within untouched processes and vice versa, and restarted services within reconfigured processes.)
- Use spiel commands to validate health of all services and processes.
- Perform some I/O on the mero fs, make sure it all succeeds (thus validating that cluster is truly alive, and all services and processes are truly OK).
- Use spiel commands to validate health of all services and processes.

Cmd#2: FS stats commands

- Start mero cluster.
- Use spiel commands to validate health of all services and processes.
- Test IO operation effects:
  - Get file system stats.
  - Create new file, inflate it by a predefined number of bytes.
  - Get file system stats, make sure free space decreased, total space remained.
  - Delete the new file.
  - Get file system stats, make sure free space returned to original value, total space remained.
- Test file system repair effects:
  - Provoke file system error.
  - Start repair process
  - While repairing, get file system stats, make sure free space decreased.
  - When repair completed, get file system stats, make sure free space recovered.
- Test reconfigure effects:
  - Detach some device, make sure free space decreased, total space decreased.
  - Attach the device back, make sure free space recovered, total space recovered.
  - Format device.
  - When formatted, make sure free space increased.
- Stop mero cluster.

Commented [9]: Not sure if detach/attach actions required here. Need to update when more details on storage device formatting available.

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.]

[i.m0.spiel]

The Spiel library purpose, behavior patterns and mechanisms are explicitly and extensively described by the current design document.

[i.m0.spiel.concurrency-and-consistency]

The concurrent and consistent configuration change distribution is described in Configuration change, Cache refresh, Version re-election sections.

[i.m0.spiel.transaction], [i.m0.conf.transaction]

The transaction basic flow is described in Configuration change section.

[i.m0.spiel.conf-db-abstraction]

The configuration database abstraction approach is described in Normal processing, Fault tolerance sections.

[i.m0.spiel.element-id-generation]

Element ID is provided by Spiel API user by design.

[i.m0.spiel.element-types]

The supported element types coverage is subject for DLD.

[i.m0.spiel.conf-actions]

The supported configuration actions is subject for DLD.

[i.m0.spiel.conf-validation]

The configuration validation is described in Phase 1: LOAD section.

[i.m0.spiel.environment]

See final note in Configuration change section.

[i.m0.spiel.cmd-iface]

Command interface control flow is described in Command interface section. Implementation details will be covered with Command interface DLD.

[i.m0.spiel.cmd-iface-sync]

Spiel command interface is synchronous. See Command interface section.

[i.m0.spiel.cmd-iface-ids]

Every command interface API function has element ID as parameter.

[i.m0.spiel.cmd-explicit]

Configuration database change doesn't trigger any commands execution in the cluster (starting services, attaching new devices, etc.) These commands are issued explicitly by Spiel API user. See Phase 1: LOAD, Phase 2: FLIP sections, where on configuration change no automatic command execution occurs.

[i.m0.spiel.supported-commands]

The supported commands list is subject for DLD.

[i.m0.rm.rw-lock]

The RW lock usage is described in Cache refresh, Version re-election sections.

[i.m0.rm.failover]

Failover is provided by means described in RM service failover section.

[i.m0.conf.replicas]

The use of configuration database replicas is described in Cache refresh, Version re-election, Phase 1: LOAD, Phase 2: FLIP sections.

[i.m0.conf.consistency]

The issue of providing consistent use of potentially inconsistent replicas is described in Cache refresh, Version re-election, Phase 1: LOAD, Phase 2: FLIP sections.

[i.m0.conf.delivery]

TBD

[i.m0.conf.quorum]

The process of reaching quorum is described in Cache refresh, Version re-election sections.

[i.m0.conf.attributes]

The use of configuration database attributes is described in Cache refresh, Version re-election sections.

[i.m0.confd.update-protocol]

The two-phase update protocol is described in Phase 1: LOAD, Phase 2: FLIP sections.

[i.m0.confc.rw-lock]

<..>

[i.m0.confc.quorum]

The process of reaching quorum in configuration client is described in Version re-election section.

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.]

5.3. Security model

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

5.4. 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.]

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.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.]

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.]

8.2. 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.]

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] <..>