High level design of Colibri configuration caching

by Valery V. Vorotyntsev <valery_vorotyntsev@xyratex.com>

Date: 2011-11-29
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 design document.]

This document presents a high level design (HLD) of Colibri configuration caching. 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.]

Configuration information of a Colibri cluster (node data, device data, filesystem tuning parameters, etc.[1]) is stored in a database, which is maintained by a dedicated management service --- configuration server. Other services and clients access configuration information by using API of configuration client library.

Configuration caching provides a higher level interface to the configuration database, convenient to use by upper layers. The implementation maintains data structures in memory, fetching them from the configuration server if necessary. Configuration caches are maintained in management-, metadata- and io-services, and in clients.

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.]

- Colibri **configuration** is part of C2 cluster meta-data.
- **Configuration database** is a central repository of C2 configuration.
- **Confd** (configuration server) is a management service that provides configuration clients with information obtained from configuration database.
- **Confc** (configuration client library, configuration client) is a library that provides configuration consumers with interfaces to query C2 configuration.
- **Configuration consumer** is any software that uses confc API to access C2 configuration.
- **Configuration cache** is configuration data stored in node's memory. Confc library maintains such a cache and provides configuration consumers with access to its data. Confd also uses configuration cache for faster retrieval of information requested by configuration clients.
- **Configuration object** is a data structure that contains configuration information. There are several types of configuration objects: profile, service, node, etc.

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.conf.async]** Configuration querying should be a non-blocking operation.
- **[r.conf.cache]** Configuration information is stored (cached) in management-, metadata-, and io- services, and on clients.
- **[r.conf.cache.in-memory]** Implementation maintains configuration structures in memory, fetching them from the management service if necessary.
- **[r.conf.cache.resource-management]** Configuration caches should be integrated with the resource manager[4].

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.]

This design assumes Colibri configuration to be read-only. Implementation of writable configuration is postponed.

A typical use case is when a client (confc) requests configuration from a server (confd). The latter most likely already has all configuration in memory: even large configuration data base is very small compared with other meta-data.

Simplistic variant of configuration server always load the entire database into the cache. This considerably simplifies the locking model (reader-writer) and configuration request processing.

4. Functional specification

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

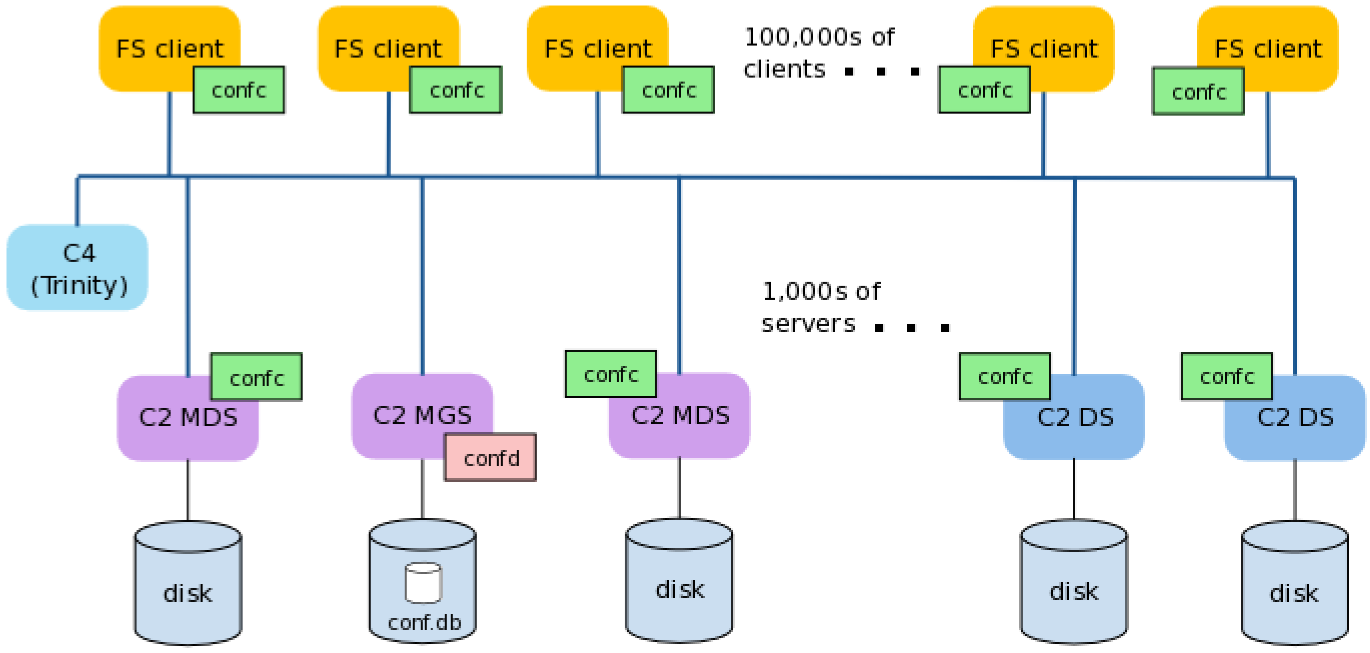

Fig. 1. Colibri cluster with configuration modules[1]

Configuration of a Colibri cluster is stored in the configuration database ("conf.db" on Fig. 1). Colibri services and filesystem clients --- configuration consumers --- have no access to this database. In order to work with configuration information, they use API and data structures provided by confc library.

Confc's obtain configuration data from configuration server (confd); only the latter is supposed to work with configuration database directly.

The following figure shows interfaces of confc and confd modules.

[1] Source file: conf-cluster.dia

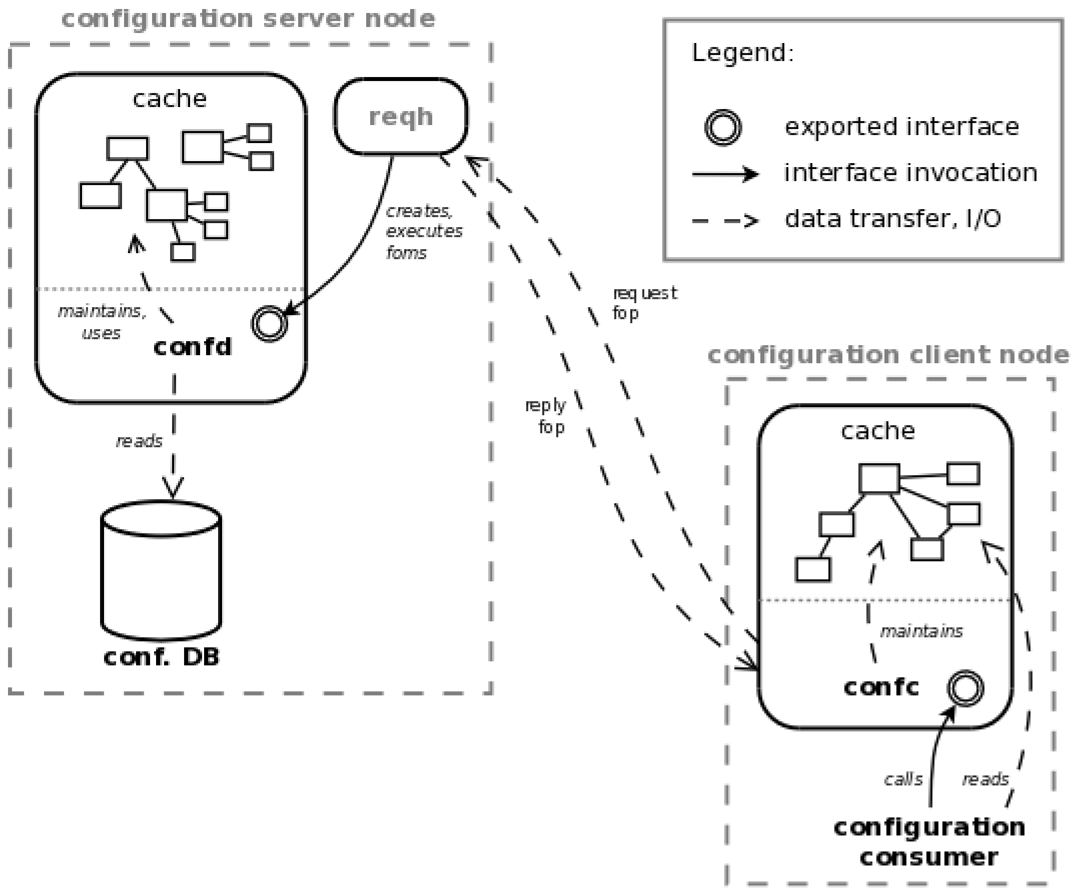

Fig. 2. Configuration interfaces

Configuration consumer (the bottom right corner of the diagram) accesses configuration information, which is stored in the configuration cache. The cache is maintained by confc library. If the data needed by a consumer is not cached, confc will fetch this data from confd.

Confd has its own configuration cache. If the data requested by a confc is missing from this cache, confd gets the information from the configuration database.

4.1. Configuration data model

Configuration database consists of tables, each table being a set of {key, value} pairs. The schema of configuration database is documented in [2].

Confd and confc organize configuration data as a directed acyclic graph (DAG) with vertices being configuration objects and edges being relations between objects. This DAG is shown on Fig. 3.

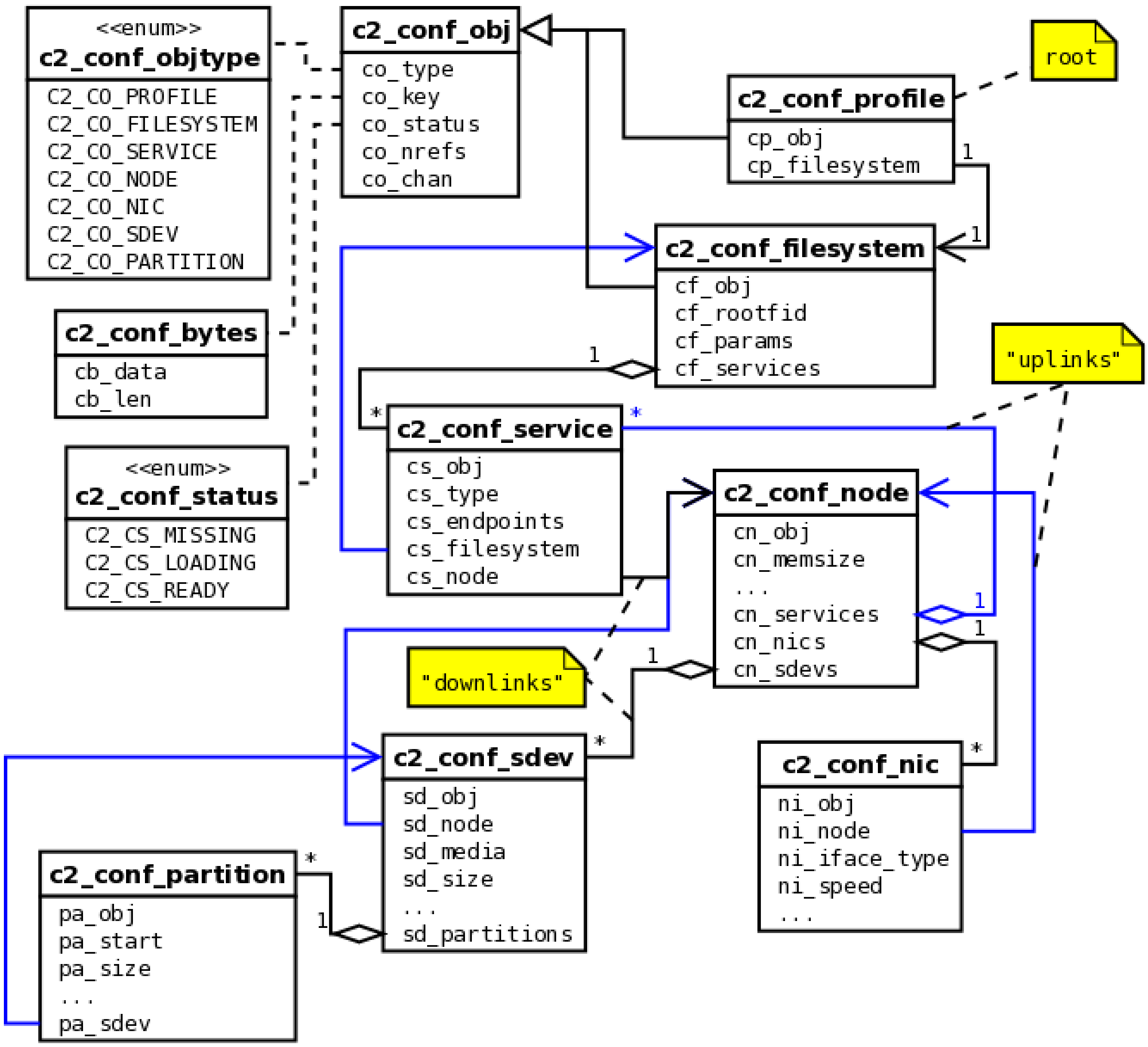

Fig. 3. Configuration objects

Note how c2_conf_profile and c2_conf_filesystem "inherit" from c2_conf_obj. The rest of configuration object types (c2_conf_service, c2_conf_node, etc.) also inherit from c2_conf_obj; the corresponding inheritance lines are not shown on the diagram as they would add too much noise to it.

Profile object is the root of configuration data provided by confc. To access other configuration objects, a consumer follows the links (relations), "descending" from the profile object.

```
profile
 \_ filesystem
     \_ service
         \_ node
             \_ nic
             \_ storage device
                 \_ partition
```

Relation is a pointer from one configuration object to another configuration object or to a collection of objects. In the former case it is *one-to-one* relation, in the latter case it is *one-to-many*.

Some relations are explicitly specified in corresponding records of the configuration database (e.g., a record of "profiles" table contains name of the filesystem associated with this profile). Other relations are deduced by the confd (e.g., the list of services that belong given filesystem is obtained by scanning "services" table and selecting entries with particular value of 'filesystem' field).

Table 1. Relations of configuration objects

Configuration object	**Relations specified in the DB record**	**Relations deduced by scanning other DB tables**
profile	.filesystem	--
filesystem	--	.services
service	.filesystem, .node	--
node	--	.services, .nics, .sdevs
nic	.node	--
storage device (sdev)	.node	.partitions
partition	.storage_device	--

Relation is a *downlink* if its destination is located further from the root of configuration DAG than the origin. Relation is an *uplink* if its destination is closer to the root than the origin (uplinks are shown in blue colour on Fig. 3).

Configuration object is a *stub* if its status (`.*_obj.co_status` subfield) is not equal to `C2_CS_READY`. Stubs contain no meaningful configuration data apart from object's type and key.

Configuration object is *pinned* if its reference counter (`.*_obj.co_nrefs` subfield) is non-zero. When a configuration consumer wants to use an object, it pins it in order protect existence of the object in the cache. Pinning of an object makes confc library request a corresponding distributed lock (resource) from the resource manager.

4.2. Path

Imagine a sequence of downlinks and keys for which the following is true:

- the first element (if any) is a downlink;
- a one-to-many downlink is either followed by a key or is the last element;
- a key is preceded by a one-to-many downlink.

Such a sequence R and a configuration object X represent a *path* to configuration object Y if Y can be reached by starting at X and following all relations from R sequentially. X object is called path *origin*, elements of R are path *components*.

Confc library uses `c2_confc_path` data structure to represent a path. The members of this structure are:

- *p_origin* --- path origin (`struct c2_conf_obj *`). `NULL` for absolute path;
- *p_comps* --- array of components. A component is either a downlink, represented by a type of target object (`enum c2_conf_objtype`), or a key.

Examples (pseudocode):

- `{ NULL, [FILESYSTEM, SERVICE, "foo", NODE, NIC, "bar"] }` --- absolute path (origin =

NULL) to the NIC with key “bar” of the node that hosts service “foo”;

- { node_obj, [SDEV, “baz”, PARTITION] } --- relative path to a list of partitions that belong “baz” storage device of a given node.

4.3. Subroutines

- c2_confc_init()
  Initiates configuration client, creates the root configuration object.
  Arguments:
  - *profile* --- name of profile to be used by this confc;
  - *confd_addr* --- address of confd end point;
  - *sm_group* --- state machine group (struct c2_sm_group *) that will be associated with configuration cache.

- c2_confc_open()
  Requests an asynchronous opening of a configuration object. Initiates retrieval of configuration data from the confd, if the data needed to fulfill this request is missing from configuration cache.
  Arguments:
  - *path* --- path to configuration object (see section 4.2). The caller must guarantee existence and immutability of *path* until the state machine, embedded in *ctx* argument, terminates or fails;
  - *ctx* --- fetch context (struct c2_confc_fetchctx *) containing:
    - state machine (struct c2_sm);
    - FOP (struct c2_fop);
    - asynchronous system trap (struct c2_sm_ast) that will be posted to confc’s state machine group when a response from confd arrives;
    - resulting pointer (void *) that will be set to the address of requested configuration object iff the state machine terminates successfully. Otherwise the value is NULL;
    - errno.

- c2_confc_open_sync()
  Synchronous variant of c2_confc_open(). Returns a pointer to requested configuration object or NULL in case of error.
  Argument: *path* --- path to configuration object.

- c2_confc_close()
  Closes a configuration object opened with c2_confc_open() or c2_confc_open_sync().

- c2_confc_diropen()
  Requests an asynchronous opening of a collection of configuration objects. Initiates retrieval of configuration data from the confd, if the data needed to fulfill this request is missing from configuration cache.
  Arguments:
  - *path* --- path to collection of configuration objects (see section 4.2). The caller must guarantee existence and immutability of *path* until the state machine, embedded in *ctx* argument, terminates or fails;
  - *ctx* --- fetch context (the structure is described above). Its ‘resulting pointer’ member will be set to non-NULL opaque value iff the state machine terminates successfully. This value is an argument for c2_confc_dirnext() and c2_confc_dirclose() functions.

- c2_confc_diropen_sync()
  Synchronous variant of c2_confc_diropen(). Returns an opaque pointer to be passed to c2_confc_dirnext(). Returns NULL in case of error.

- c2_confc_dirnext()
  Returns next element in a collection of configuration objects.
  Argument: *dir* --- opaque pointer obtained from a fetch context (see *ctx* argument of c2_confc_diropen()).

- c2_confc_dirclose()
  Closes a collection of configuration objects opened with c2_confc_diropen() or c2_confc_diropen_sync().
  Argument: *dir* --- opaque pointer obtained from a fetch context.

- c2_confc_fini()
  Terminating routine: destroys configuration cache, freeing allocated memory.

4.4. FOP types

Confc requests configuration information by sending c2_conf_fetch FOP to confd. This FOP contains the path to the requested configuration object/directory (see 4.2. Path). Note that the path in FOP may be shorter then the path originally specified in c2_confc_*open*() call: if some of the objects are already present in confc cache, there is no reason to re-fetch them from confd.

Confd replies to c2_conf_fetch with c2_conf_fetch_resp FOP, containing:
- status of the retrieval operation (0 = success, -Exxx = failure);
- array (SEQUENCE in .ff terms) of configuration object *descriptors*.

If the last past component, specified in c2_conf_fetch, denotes a *directory* (i.e., a collection of configuration objects), then confd's reply must include descriptors of all the configuration objects of this directory. For example, if a path targets a collection of partitions, then c2_conf_fetch_resp should describe every partition of the targeted collection.

Note that in the future, configuration data will be transferred from confd to confc using RPC bulk interfaces. Current implementation embeds configuration information in a response FOP and uses encoding and decoding functions generated by fop2c from .ff description.

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.]

Confd service is parametrized by address of the end point, associated with this service, and path to the configuration database. Confd state machine, created by the request handler[3], obtains requested configuration (either from configuration cache or from configuration database; in the latter case the data gets added to the cache), generates a response FOP (c2_conf_fetch_resp) populated with configuration information, and sends it back to confc.

Confc is parametrized by name of profile, address of confd end point, and state machine group. When c2_confc_fetch() function is called, confc checks whether all of the requested configuration data is available in cache. If something is missing, confc sends request to the confd. When response arrives, confc updates the configuration cache.

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.conf.async]**: `c2_confc_fetch()` is an asynchronous non-blocking call. Confc state machine keeps the information about the progress of the query.
- **[i.conf.cache]**: confc components, used by Colibri services and filesystem clients, maintain the caches of configuration data.
- **[i.conf.cache.in-memory]**: a confc obtains configuration from the confd and builds in-memory data structure in the address space of configuration consumer.
- **[r.conf.cache.resource-management]**: prior to pinning configuration object or modifying configuration cache, configuration module (confd or confc) requests appropriate resource from the resource manager.

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.]

- **[configuration.schema]** The confd is aware of the schema of configuration database. The actual schema is defined by another task[2] and is beyond the scope of this document.
- **[confc.startup]** Parameters, needed by confc startup procedure, are provided by calling code (presumably the request handler or the service startup logic).

5.3. Security model

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

No security model is defined.

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.]

The implementation must address the following requirements:

- **[r.conf.confc.kernel]** Confc library must be implemented for the kernel.
- **[r.conf.confc.user]** Confc library must be implement for user space.
- **[r.conf.confd]** Confd service must be implemented for user space.
- **[r.conf.cache.data-model]** The implementation should organize configuration information as outlined in section 4.1. The same data structures should be used for confc and confd caches, if possible. Configuration structures must be kept in memory.
- **[r.conf.cache.pinning]** Pinning of an object protects existence of this object in the cache. Pinned object can be moved from stub condition to “ready”.
- **[r.conf.cache.unique-objects]** Configuration cache must not contain multiple objects with the same identity (*identity* of a configuration object is a tuple of type and key).

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.]

Confd and confc modules define state machines for asynchronous non-blocking processing of configuration requests. State diagrams of such machines are shown below.

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.]

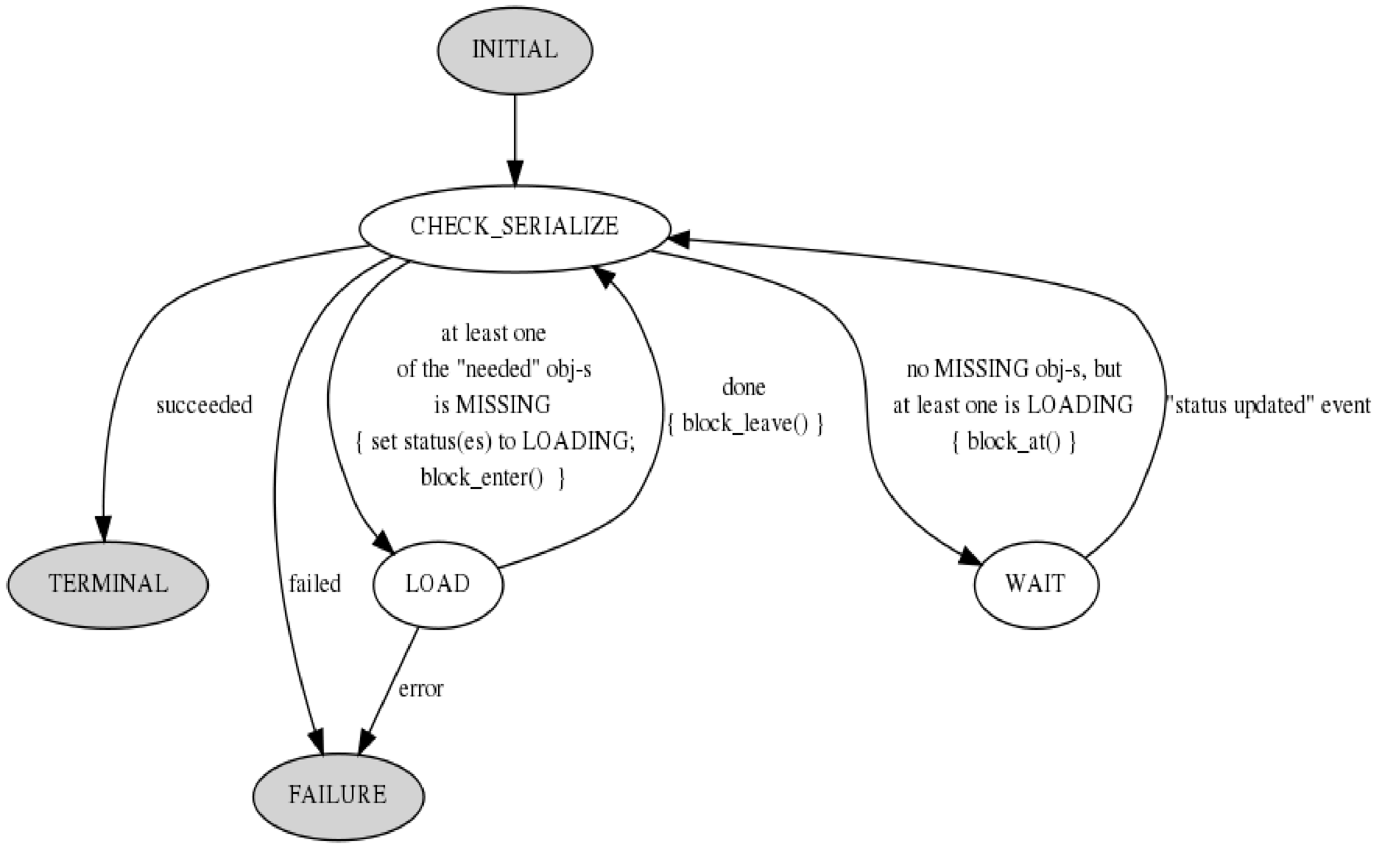

Fig. 4. FOM states (confd side)

While a confd state machine is in CHECK_SERIALIZE state, it keeps configuration cache locked.

LOAD state unlocks the cache, fetches missing objects from the configuration database, and falls back to CHECK_SERIALIZE. After configuration object is successfully loaded from the database, its status is set to `C2_CS_READY` and its channel (`c2_conf_obj::co_chan`) is broadcasted.

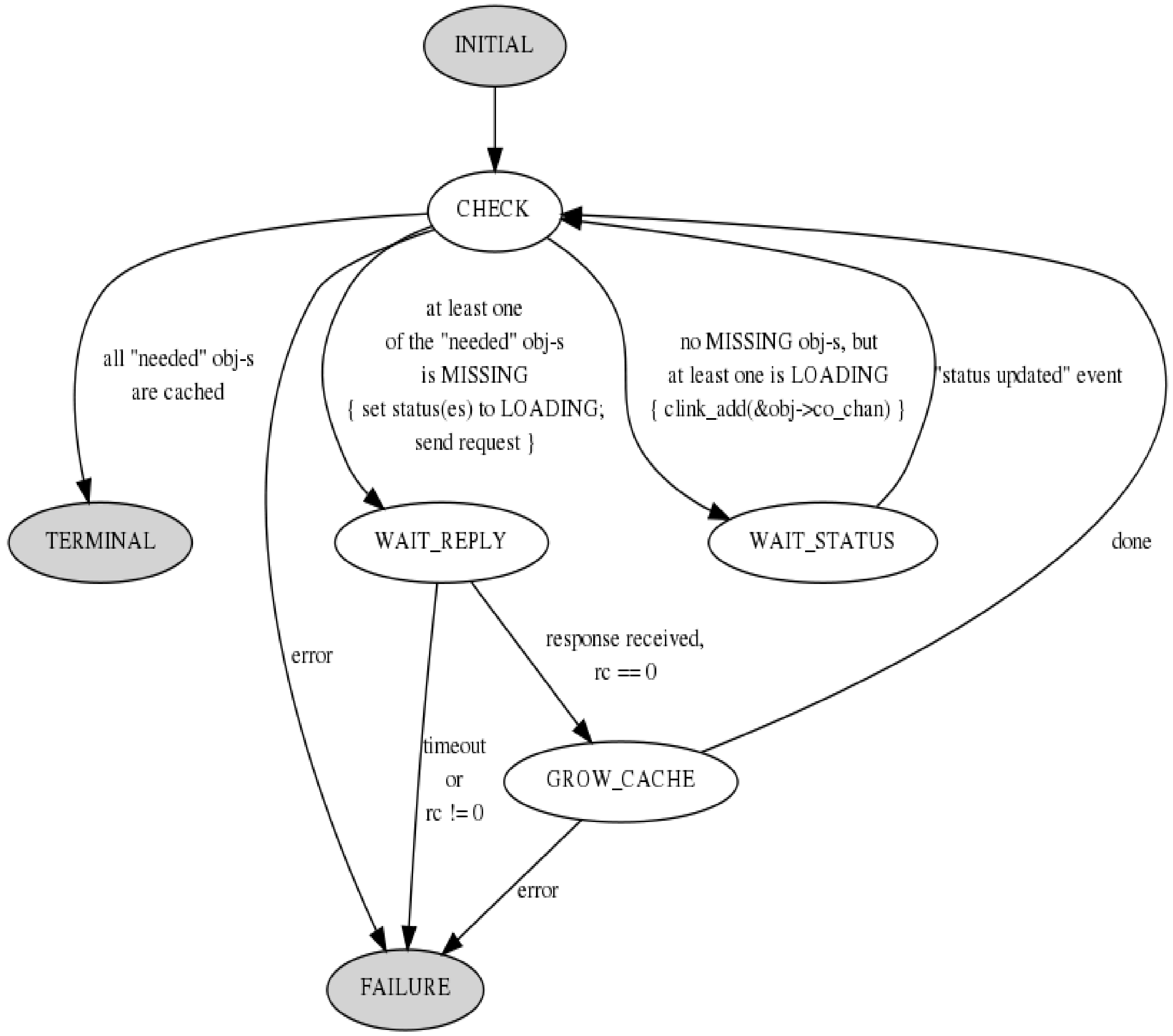
INITIAL
CHECK
all "needed" obj-s are cached
at least one of the "needed" obj-s is MISSING { set status(es) to LOADING; send request }
no MISSING obj-s, but at least one is LOADING { clink_add(&obj->co_chan) }
"status updated" event
TERMINAL
WAIT_REPLY
WAIT_STATUS
done
error
response received, rc == 0
timeout or rc != 0
GROW_CACHE
error
FAILURE

Fig. 5. States of a confc state machine

Configuration cache is associated with a state machine group (c2_sm_group). While a confc state machine is in CHECK state, it keeps the state machine group locked.

GROW_CACHE state releases state machine lock and performs the following actions for every configuration descriptor decoded from confd's response (c2_conf_fetch_resp):

- lock the state machine group;
- make sure that target object of every relation, mentioned by the descriptor, is present in cache (stubs are created for absent target objects);
- if an object with described identity (type and key) already exists in cache and its status is C2_CS_READY then compare existing object with the one received from confd, reporting inequality by means of ADDB API;
- if an object with described identify already exists and is a stub then fill it with the received configuration data, change its status to C2_CS_READY, and announce status update on object's channel;
- unlock the state machine group.

6.2. State invariants

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

If a confc state machine in GROW_CACHE state, while trying to add an object, finds that the object with this key already exists, then either the existing object is a stub or new and existing objects are equal.

This invariant is also applicable to LOAD state of a confd state machine.

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.]

6.3.a. confd

Several confd state machines (FOMs) can work with configuration cache --- read from it and add new objects to it --- concurrently.

A confd state machine keeps configuration cache locked while examining its completeness (i.e., checking whether it has enough data to fulfill confc's request) and serializing configuration data into response FOP. Another confd state machine is unable to enter CHECK_SERIALIZE state until the mutex, embedded into configuration cache data structure (`c2_conf_cache::cc_lock`), is released.

The same lock is used to prevent concurrent modifications to configuration DAG. A confd state machine must hold `c2_conf_cache::cc_lock` in order to add new configuration object to the cache or fill a stub object with configuration data.

6.3.b. confc

Configuration cache at confc side is shared by configuration consumers that read from it and confc state machines that traverse the cache and add new objects to it. Consumers and state machines can work with configuration cache concurrently.

Implementation of confc serializes state transitions by means of a state machine group, associated with configuration cache. It is up to upper layers to decide whether this group will be used exclusively for confc or for some other state machines as well.

The state machine group is locked by a confc state machine running in CHECK state. The lock (`c2_sm_group::s_lock`) is also acquired by a confc state machine for time interval needed to append new configuration object to the cache.

Configuration consumers must pin the objects they are using (and not to forget to unpin them afterwards, of course). Pinned configuration objects cannot be deleted by confc library but those that have zero references may be deleted at any moment (e.g., when another consumer thread calls `c2_confc_fini()`).

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

1) Getting data on the filesystem.

1. Configuration consumer initializes statically defined or dynamically allocated path object. The 'origin' member is NULL and the only element of 'components' is C2_CO_FILESYSTEM.
2. The consumer initializes fetch context object.
3. The consumer registers a clink on the channel of the state machine embedded in the fetch context.
4. The consumer calls c2_confc_open(). The arguments are path and context objects.
5. The consumer awaits completion of the state machine.
   NOTE: If a consumer wants to synchronously wait for state machine's completion, steps 2--5 are substituted with a single call to c2_confc_open_sync().
6. The 'resulting pointer' member of the fetch context object (or, in synchronous case, the value returned by c2_confc_open_sync()) points to the requested configuration object (filesystem) or is NULL in case of error.
7. The consumer gets configuration of a filesystem by reading fields of the filesystem configuration object.
8. The consumer calls c2_confc_close() passing pointer to the filesystem object as an argument.

2) Getting configuration of a service of specific type.

1. Configuration consumer initializes path object: origin = NULL, components = [C2_CO_FILESYSTEM, C2_CO_SERVICE].
2. The consumer sets up fetch context and calls asynchronous c2_confc_diropen() or just calls synchronous c2_confc_diropen_sync().
3. State machine has completed. The consumer obtains a pointer to the "directory" of services objects.
4. The consumer calls c2_confc_dirnext() function in a loop. The argument is the directory pointer obtained in previous step. Returned value is a pointer to service configuration object. The consumer loops until the function returns NULL; the end of services list has been reached at this point.
   - Loop body: if service type is the wanted one, use the service configuration object.
5. Close the collection of services: c2_confc_dirclose().

3) List the devices used by a specific service on a specific node.

Configuration consumer loops over filesystem's services (scenario 2 describes the prelude). Loop body:
1. If service type is not the wanted one, skip this iteration.
2. Open configuration object that represents hosting node of current service. Path object being passed to c2_confc_open() or c2_confc_open_sync() originates at the service object and has only one path component --- C2_CO_NODE.
3. If node's key differs from the wanted value, close the node object (c2_confc_close()) and go to the next iteration.
4. Initiate path object: origin = node, components = [C2_CO_NIC]. Use c2_confc_diropen() or c2_confc_diropen_sync() to open the collection of node's network interface cards.
5. Use c2_confc_dirnext() to iterate over NICs and use their configuration.
6. Close the collection of NICs with c2_confc_dirclose().
7. Iteration over node's storage devices is similar (path.components should be [C2_CO_SDEV] though).

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.]

1. Invalid path.

   Invalid path is specified to any of `c2_confc_*open*()` calls. When a state machine completes, the 'resulting pointer' member of a fetch context remains unset (is `NULL`), the 'errno' member is set to non-zero value (e.g., `ENOENT`). Error is logged via ADDB.

2. No response from confd.

   Confc sends configuration request to confd. No response, timeout occurs. Resulting pointer = `NULL`, errno = `ETIMEDOUT`. Error is logged via ADDB.

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.]

Present design assumes that there is exactly one configuration server per cluster. The problem is that however fast a confd is, the amount of data it can serve per fixed time is limited (the same goes for the number of requests it can process). Given big enough number of confc's willing to obtain configuration simultaneously (e.g., when a cluster is starting), RPC timeouts are inevitable.

One of the possible solutions (which will probably be employed anyway, in one form or another) is to make configuration consumers keep on re-fetching the configuration data until a request succeeds or a maximum number of retries is reached. Another solution is to replicate the configuration database and have as many confd's as there are replicas. Both solutions are beyond the scope of this document.

Of course, once a confc has configuration data cached it will not need to fetch the same data from confd again (at least not until configuration is updated, which is not going to happen in this incarnation of C2 configuration system).

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.]

In the presence of resource manager, the invariant mentioned in section 6.2 is maintained by the resource manager. In the absence of resource manager, the invariant is maintained by the "cache is read-append-only" property.

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.]

Rejected:

- `c2_confc_lookup()` function that returns configuration record given name of configuration database table and record's key;
- YAML for configuration data serialization.
  (Rationale: YAML is not convenient to be parsed in the kernel. It is much simpler to use encoding and decoding functions generated by fop2c from .ff description.)

Postponed:

- writable configuration;
- configuration auto-discovery;
- using RPC bulk interfaces for transmitting configuration data.

Pointer to the root configuration object cannot be embedded in `c2_reqh` structure, because filesystem clients do not have request handler (yet).

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

None

9.1.2. Persistent storage

None

9.1.3. Core

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

- Nikita mentioned some change to `c2_sm_timedwait()` that may possibly be needed.
- 'core/cfg/cfg.h' is expected to be moved to 'core/conf/schema.h'. Its data structures may be renamed.

9.2. Installation

[How the component is delivered and installed.]

- Configuration database is created with 'yaml2db' utility (the product of 'configuration.devenum' component).
- Confd is a management service, started by request handler. It is installed together with Colibri.
- Confc is a library that configuration consumers link with.

10. References

[1] Configuration one-pager
[2] HLD of configuration.schema
[3] HLD of request handler
[4] HLD of resource management interfaces
[5] configuration.caching drafts

11. Inspection process data

[The tables below are filled in by design inspectors and reviewers. Measurements and defects are transferred to the appropriate project data-bases as necessary.]