The Corosync Cluster Engine

Steven C. Dake
Red Hat, Inc.
sdake@redhat.com

Christine Caulfield
Red Hat, Inc.
ccaulfie@redhat.com

Andrew Beekhof
Novell, Inc.
abeekhof@suse.de

Abstract

A common cluster infrastructure called the Corosync Cluster Engine is presented. The rationale for this effort as well as a history of the project are provided. The architecture is described in detail. The internal programming API is presented to provide developers with a basic understanding of the programming model to architecture mapping. Finally, examples of open source projects using the Corosync Cluster Engine are provided.

1 Introduction

The Corosync Cluster Engine [Corosync] Team has designed and implemented the Corosync Cluster Engine to meet logistical needs of the cluster community. Some members of the cluster developer community have strong desires to reduce technology and community fragmentation.

Technology fragmentation results in difficulty with interoperability. Different project clustering systems do not inter-operate well because they each make decisions regarding the state of the cluster in inconsistent ways. Each cluster software may take different approaches to managing failures, communicating, reading configuration files, determining cluster membership, or recovering from failures.

Community fragmentation results in dispersal of developer talent across many different projects. Most projects have a very small set of developers. These developers in the past have not worked on the same infrastructure but instead implement code with similar functionality. This software is then is deployed in various cluster systems and must be maintained and developed by individual projects.

The Corosync Cluster Engine resolves these issues by separating the core infrastructure from the cluster services. By making this abstraction, all cluster services can cooperate on decision making in the cluster. This abstraction also unifies the core code base under one open source group with the purpose to maintain, develop, and direct a reusable cluster infrastructure with an OSI-approved license.

2 History

The Corosync Cluster Engine was founded in January 2008 as a reduction of the OpenAIS project. The cluster infrastructure primitives are reduced from the Service Availability Forum Application Interface Specification APIs into a new project. This effort was spawned by various maintainers of cluster projects to improve interoperability and unify developer talent.

The OpenAIS project was founded in January 2002 to implement Service Availability Forum Application Interface Specification APIs [SaForumAIS]. These APIs are designed to provide an application framework for high availability using clustering techniques to reduce MTTR [Dake05]. During the development of OpenAIS, more development time was spent on the infrastructure than the APIs. As a result of the focus on the infrastructure, a completely reusable plug-in based Cluster Engine was created.

3 Architecture

3.1 Overview

Corosync Cluster Engine clusters are composed of processors connected by an interconnect. This paper defines an interconnect as a physical communication system which allows for multicast or broadcast operation to communicate packets of information. This paper defines a processor as a common computer, including a CPU, memory, network interface chip, physical storage and operating system such as Linux. This type of cluster is commonly referred to as a shared-nothing cluster.

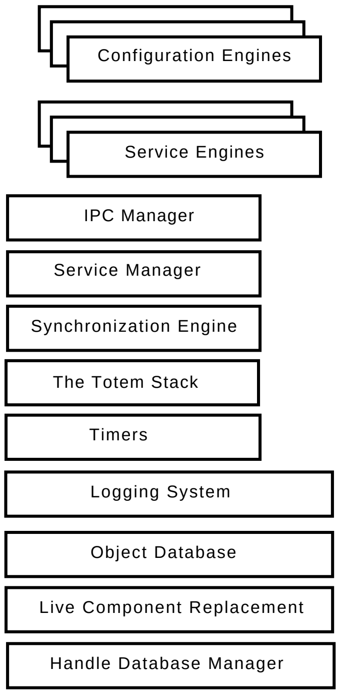

Figure 1: Corosync Cluster Engine Architecture

The Corosync Cluster Engine supports a fully componentized plug-in architecture. Every component of the Corosync Cluster Engine can be replaced by a different component providing the same functionality at processor start time.

Figure 1 depicts the architecture of the Corosync Cluster Engine process.

The subsections in this paper are organized by dependency, not importance. Every component used in the Corosync Cluster Engine is critical to creating a cluster software engine.

3.2 Handle Database Manager

The handle database manager provides a reference counting database that maps in O1 order a unique 64-bit handle identifier to a memory address. This mapping can then be used by libraries or other components of the Corosync Cluster Engine to map addresses to 64-bit values.

The handle database supports the creation and destruction of new entries in the database. Finally, mechanisms exist to obtain a reference to the object database entry and release the reference.

```
struct iface {
  void (*func1) (void);
  void (*func2) (void);
  void (*func3) (void);
};

/*
 * Reference version 0 of A and B interfaces
 */
res = lcr_ifact_reference (
  &a_ifact_handle_ver0,
  "A_iface1",
  0, /* version 0 */
  &a_iface_ver0_p,
  (void *)0xaaaa0000);

a_iface_ver0 = (struct iface *)a_iface_ver0_p;

res = lcr_ifact_reference (
  &b_ifact_handle_ver0,
  "B_iface1",
  0, /* version 0 */
  &b_iface_ver0_p,
  (void *)0xbbbb0000);

b_iface_ver0 = (struct iface *)b_iface_ver0_p;

a_iface_ver0->func1();
a_iface_ver0->func2();
a_iface_ver0->func3();

lcr_ifact_release (a_ifact_handle_ver0);

b_iface_ver0->func1();
b_iface_ver0->func2();
b_iface_ver0->func3();

lcr_ifact_release (b_ifact_handle_ver0);
```

Figure 2: Example of using multiple interfaces in one application

Garbage collection occurs automatically and a user-supplied callback may be called when the reference count for a handle reaches zero to execute destruction of the handle information.

3.3 Live Component Replacement

Live Component Replacement is the plug-in system used by the Corosync Cluster Engine. Every component in the engine is an LCR object which is loaded dynamically. LCR objects are designed to be replaceable at runtime, although this feature is not yet fully implemented.

The LCR plug-in system is different from all other plug-in systems in that a complete C interface is plugged into the process address space, instead of simply one function call. Figure 2 demonstrates the use of the LCR system.

LCR objects are linked statically or dynamically. When an interface is referenced, an internal storage area is checked to see if the object has been linked statically. If it has been linked statically, a reference will be given to the user. If it isn't found in the internal storage area, the `lcrso` directory on the storage medium will be scanned for a matching interface. If it is found it will be loaded and referenced to the user; otherwise, an error is returned.

The live component replacement plug-in system is used extensively throughout the Corosync Cluster Engine to provide dynamic run-time loading of interfaces.

3.4 Object Database

The object database provides an in-memory non-persistent storage mechanism for the configuration engines and service engines.

The object database is a collection of objects. Every object has a name and is stored in a tree-like structure. Every object has a parent. Within objects are key and value pairs which are unique to the object. Figure 3 depicts a partial object database layout.

The object database provides an API for the creation, deletion, and searching for objects. The database also provides mechanisms to read and write key and value pairs. Finally, the database provides a mechanism to find objects, iterate objects within a tree, and iterate keys within an object.

Objects have specific requirements. The object database allows multiple objects with the same name to be stored in the database with the same parent. Every object may contain key and value pairs. An object's key is unique and its value is a binary blob of data.

Because the object database is often used in parsing by the configuration engine, a special API is provided to automatically detect failures in the storing of keys and associated values within an object. On object creation, a list of valid keys for that object can be registered as well as a validation callback for each key. If the user of the API specifies an invalid key when modifying an object within the object database, the modification request will be rejected with an error. When the key is valid, before the key is modified, the validation callback is called. This validation callback verifies the contents of the value using the user-registered callback. If the callback returns an invalid value, the modification request is rejected.

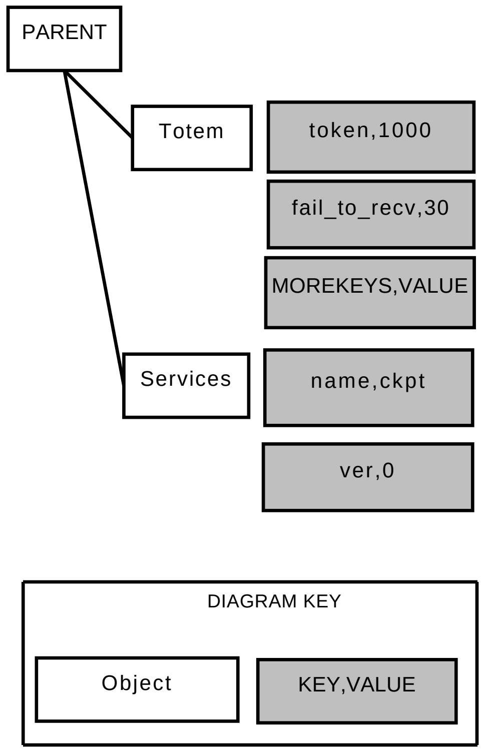

Figure 3: Typical Object Database Layout

3.5 Logging System

A common logging system is available to service engines as well as the rest of the Corosync Cluster Engine software stack. The logging system is completely non-blocking and uses a separate thread in the process address space to filter and output logging information. The logging system is a generically reusable library available to third-party processes as well as service engines. In the case that multiple service engines use the logging system, only one thread is created by the Corosync Cluster Engine.

The logging system supports logging with complete `printf()` style argument processing. Information may be printed to `stderr`, a file, and/or syslog.

A logging system may contain any number of components, called tags, which allow runtime filtering of debug messages and 8 levels of tracing messages to the logging output medium. Each tracing type may be separately filtered so specific trace numbers may be used for specific functionality.

A unique feature of the logging system is that a logging system and logging components are initialized through a constructor definition at the beginning of the C code for the file. The configuration options may also be changed at runtime. Additionally, the logging system supports the `fork()` system call.

3.6 Timers

Nearly every service engine requires the use of timers, so a timer system is provided. Time is represents in nanoseconds since the epoch, or January 1, 1970.

Timers may be set to expire at an absolute time. Another type of timer allows expiration in a certain number of nanoseconds into the future.

When a timer expires, it executes a callback registered at timer creation time to execute software code desired by the service engine designer.

3.7 The Totem Stack

The Totem Single Ring Ordering and Membership Protocol [Amir95] implements a totally ordered extended virtual synchrony communication model [Moser94]. Unlike many typical communication systems, the extended virtual synchrony model requires that every processor agrees upon the order of messages and membership changes, and that those messages are completely recovered.

A property of virtual synchrony, called agreed ordering, allows for simple state synchronization of cluster services. Because every node receives messages in the same order, processing of messages occur once the Totem protocol has ordered the message. This allows every node in the cluster to remain in synchronization when processor failure occurs or new processors are included in the membership.

One key feature of the Totem stack is that it supports the ability to communicate redundantly over multiple network interfaces. All data including the membership protocol is replicated over multiple network interfaces using the Totem Redundant Ring Protocol [Koch02].

Totem is implemented completely in userspace using user datagram protocol [Postel80] multicast. The protocol implementation can be configured to run within Internet Protocol version 4 [USC81] networks or Internet Protocol version 6 [Deering98] networks.

All communication may be, at user configuration, authenticated and encrypted using a private secret key stored securely on all nodes.

3.8 Configuration Engine

The Corosync Cluster Engine solves the issue of configuration file independence by providing the ability to load an application specific configuration engine. The configuration engine provides a method to read and write configuration files in an application specific way. These plug-ins configure the Corosync Cluster Engine as well as other components specific to an application plug-in.

In the event that the Corosync Cluster Engine executive is not running, the configuration engine can still be used by applications transparently to read and store configuration information.

3.9 Interprocess Communication Manager

The interprocess communication manager is responsible for receipt and transmission of IPC requests. The incoming IPC requests are routed via the service manager to the appropriate service engine plug-in. The service engine may send responses to a third-party process.

Every IPC connection is an abstraction of two file descriptors. One file descriptor is used for third-party process blocking request and response packets. The remaining file descriptor is used exclusively for non-blocking callback operations that should be executed by the third-party process. These two file descriptors are connected to each other during initialization of the IPC connection by the Interprocess Communication Manager.

3.10 Service Engine

A service engine is created by third parties to provide some form of cluster wide services. Some examples of these are the Service Availability Forum's Application Interface Specification checkpoint service, Pacemaker, or CMAN.

The service engine has a well defined live component replacement interface for run-time linking into the service manager. The service engine is responsible for providing a specific class of cluster service to a user via API or external control via the interprocess communication manager.

3.11 Service Manager

The service manager is responsible for loading and unloading plug-in service engines. It is also responsible for routing all requests to the service engines loaded in the Corosync Cluster Engine.

During Corosync Cluster Engine initialization, the configuration engine is loaded. The configuration engine then stores the list of service engines to load. Finally, the service manager loads every service engine.

Once the service manager loads a service, it is responsible for initializing the service engine. When the user requests an operation via the interprocess communication manager, that request is routed to the appropriate service engine by the service manager. The service manager is also responsible for sending membership changes to the service manager. A service engine replicates information via the low-level Totem Single Ring Protocol by transmitting messages. These transmitted messages are delivered via the service manager to a service engine. Finally, the service manager is responsible for routing synchronization activities with the synchronization engine.

3.12 Synchronization Engine

The synchronization engine is responsible for directing the recovery of all service engines after a failure or addition of a processor. A service engine may optionally use the synchronization engine, or set the synchronization engine functions to `NULL`, in which case they won't be used.

```
typedef uint64_t cpg_handle_t;

typedef enum {
  CPG_DISPATCH_ONE,
  CPG_DISPATCH_ALL,
  CPG_DISPATCH_BLOCKING
} cpg_dispatch_t;

typedef enum {
  CPG_TYPE_UNORDERED,
  CPG_TYPE_FIFO,
  CPG_TYPE_AGREED,
  CPG_TYPE_SAFE
} cpg_guarantee_t;

typedef enum {
  CPG_FLOW_CONTROL_DISABLED,
  CPG_FLOW_CONTROL_ENABLED
} cpg_flow_control_state_t;

typedef enum {
  CPG_OK = 1,
  CPG_ERR_LIBRARY = 2,
  CPG_ERR_TIMEOUT = 5,
  CPG_ERR_TRY_AGAIN = 6,
  CPG_ERR_INVALID_PARAM = 7,
  CPG_ERR_NO_MEMORY = 8,
  CPG_ERR_BAD_HANDLE = 9,
  CPG_ERR_ACCESS = 11,
  CPG_ERR_NOT_EXIST = 12,
  CPG_ERR_EXIST = 14,
  CPG_ERR_NOT_SUPPORTED = 20,
  CPG_ERR_SECURITY = 29,
  CPG_ERR_TOO_MANY_GROUPS=30
} cpg_error_t;

typedef enum {
  CPG_REASON_JOIN = 1,
  CPG_REASON_LEAVE = 2,
  CPG_REASON_NODEDOWN = 3,
  CPG_REASON_NODEUP = 4,
  CPG_REASON_PROCDOWN = 5
} cpg_reason_t;

struct cpg_address {
  uint32_t nodeid;
  uint32_t pid;
  uint32_t reason;
};

#define CPG_MAX_NAME_LENGTH 128

struct cpg_name {
  uint32_t length;
  char value[CPG_MAX_NAME_LENGTH];
};

#define CPG_MEMBERS_MAX 128
```

Figure 4. The Closed Process Group Interface Definitions

The synchronization engine has four states

- `sync_init`
- `sync_process`
- `sync_activate`
- `sync_abort`

The first step in the synchronization process for a service engine is initialization. The `sync_init` call in a service engine stores information for executing the recovery algorithm created by the service engine designer.

The `sync_process` is executed to process the recovery operation. Because the Totem protocol transmission queue may become full on the processor executing recovery, `sync_process` may have to return without completing by returning a negative value. If synchronization was completed, a value of zero should be returned.

If at any time during synchronization, a new processor joins the membership or a processor leaves the membership, the `sync_abort` call will be executed to reset any state created by `sync_init`.

After synchronization has completed on all nodes, `sync_activate` is called to activate the new data set for the service engine.

3.13 Default Service Engines

The Corosync Cluster Engine provides a few default service engines which are generically useful. Other default service engines will be provided in the future.

3.13.1 Closed Process Group Service Engine

The closed process group API and the associated service engine are responsible for providing closed process group messaging semantics. Closed process groups are a specialization of the process groups semantics [Birman93].

Any process may join a process group. A process is a system task with a process identifier, often called a PID. Once joined, a join message is sent to every process in the membership. The contents of the join message are the process ID of the process and the processor identifier that the joining process on which the process is running. When the process leaves the process group, either voluntarily, or as a result of failure, a leave message is sent to every remaining processor.

The closed process group service engine allows the transmission and delivery of messages among a collection of processors that have joined the process group.

The definitions in Figure 4 and API in Figure 5 are used to implement the closed process group system. At all times, the extended virtual synchrony messaging model is maintained by this service.

To join a process group, `cpg_join()` is used in a C program. The user passes the process group to join. To leave a process group, `cpg_leave()` is used. Failures automatically behave as if the process had executed a `cpg_leave()` function call. Messages are sent to every node in the process group using the C function `cpg_mcast()`.

Changes in the process membership and delivery of messages are executed using the `cpg_dispatch()` C function call. This function calls the `cpg_deliver_fn_t()` function to deliver messages and `cpg_confchg_fn_t()` to deliver membership changes. These functions are registered during initialization with the `cpg_initialize()` function call.

3.13.2 Configuration Database Service Engine

The configuration database service engine provides a C programming API to third-party processes to read and write configuration information in the object database. The API is essentially the same as that used in the object database.

The configuration database service C API may operate when the Corosync Cluster Engine is not running for configuration purposes. In this operational mode, a configuration engine is loaded and automatically used to read or write the object database after the user of the C API has made changes to the object database.

4 Library Programming Interface

4.1 Overview

The library programming interface is useful for third-party processes that wish to access a Corosync service

```
typedef void (*cpg_deliver_fn_t) (
  cpg_handle_t handle,
  struct cpg_name *group_name,
  uint32_t nodeid,
  uint32_t pid,
  void *msg,
  int msg_len);


typedef void (*cpg_confchg_fn_t) (
  cpg_handle_t handle,
  struct cpg_name *group_name,
  struct cpg_address *member_list,
         int member_list_entries,
  struct cpg_address *left_list, int
         left_list_entries,
  struct cpg_address *joined_list, int
         joined_list_entries);


typedef struct {
  cpg_deliver_fn_t cpg_deliver_fn;
  cpg_confchg_fn_t cpg_confchg_fn;
} cpg_callbacks_t;


cpg_error_t cpg_initialize (
  cpg_handle_t *handle,
  cpg_callbacks_t *callbacks);

cpg_error_t cpg_finalize (
  cpg_handle_t handle);

cpg_error_t cpg_fd_get (
  cpg_handle_t handle, int *fd);

cpg_error_t cpg_context_get (
  cpg_handle_t handle, void **context);

cpg_error_t cpg_context_set (
  cpg_handle_t handle, void *context);

cpg_error_t cpg_dispatch (
  cpg_handle_t handle, cpg_dispatch_t
               dispatch_types);

cpg_error_t cpg_join (
  cpg_handle_t handle,
  struct cpg_name *group);

cpg_error_t cpg_leave (
  cpg_handle_t handle,
  struct cpg_name *group);

cpg_error_t cpg_mcast_joined (
  cpg_handle_t handle,
  cpg_guarantee_t guarantee,
  struct iovec *iovec, int iov_len);
```

Figure 5: The Closed Process Group Interface API

engine. The library programming interface provides handle management and connection management with the hdb inline library and the cslib library.

4.2 Handle Database API

The handle database API, shown in Figure 6, is responsible for managing handles that map to memory blocks. Handle memory blocks are reference counted and the handle memory area is automatically freed when no user references the handle. The API is fully thread safe and may be used in multithreaded libraries.

When creating a handle database, the function `hdb_create()` should be used. When destroying a handle database, the function `hdb_destroy()` should be used.

To create a new entry in the handle database, use the function `hdb_handle_create()`. Once the handle is created, it will start with a reference count of 1. To reduce the reference count and free the handle, the function `hdb_handle_destroy()` should be executed.

Once a handle is created with `hdb_handle_create()`, it can be referenced with `hdb_handle_get()`. This function will retrieve the memory storage area relating to the handle specified by the user. When the library is done using the handle, `hdb_handle_put()` should be executed.

4.3 Corosync Library API

The Corosync Library API, defined in Figure 7, provides a mechanism for communicating with Corosync service engines. A library may connect with the Corosync Cluster Engine by using `cslib_service_connect()`. This function returns two file descriptors. One file descriptor is used for request and response messages. The remaining file descriptor is used for callback data that shouldn't block normal requests.

Once an IPC connection is made, a request message can be sent with `cslib_send()`. A response may be received with `cslib_recv()`. These functions generally shouldn't be used unless the size of the message to be received is variable length.

When the size of the message to be received is known, `cslib_send_recv()` should be used. This will send a request, and receive a response of a known size.

```
struct hdb_handle {
  int state;
  void *instance;
  int ref_count;
};

struct hdb_handle_database {
  unsigned int handle_count;
  struct hdb_handle *handles;
  unsigned int iterator;
  pthread_mutex_t mutex;
};

void hdb_create (
  struct hdb_handle_database
         *handle_database);

void hdb_destroy (
  struct hdb_handle_database
         *handle_database);

int hdb_handle_create (
  struct hdb_handle_database *handle_database,
  int instance_size,
  unsigned int *handle_id_out);

int hdb_handle_get (
  struct hdb_handle_database *handle_database,
  unsigned long long handle,
  void **instance);

void hdb_handle_put (
  struct hdb_handle_database *handle_database,
  unsigned long long handle);

void hdb_handle_destroy (
  struct hdb_handle_database *handle_database,
  unsigned long long handle);

void hdb_iterator_reset (
  struct hdb_handle_database
         *handle_database);

void hdb_iterator_next (
  struct hdb_handle_database *handle_database,
  void **instance,
  unsigned long long *handle);
```

Figure 6: The Handle Database API Definition

All of these functions handle recovery of message transmission on short reads or writes, or in the event of signals or other system errors that may occur.

Finally, it is useful to poll a file descriptor, especially in a dispatch routine. This can be achieved by using `cslib_poll()` which is similar to the poll system call

```
        cslib_service_connect (
          int *response_out,
          int *callback_out,
          unsigned int service);

        cslib_send (int s,
          const void *msg,
          size_t len);

        cslib_recv (int s,
          const void *sg,
          size_t len);

        cslib_send_recv (
          int s,
          struct iovec *iov,
          int iov_len,
          void *response,
          int response_len);

        cslib_poll (
          struct pollfd *ufds,
          unsigned int nfds,
          int timeout);
```

Figure 7: The Corosync Library API Definition

except it retries on signals and other errors which are recoverable.

5 Service Engine Programming Model and Interface

5.1 Overview

A service engine consists of a designer-supplied plug-in interface coupled with the implementation of functionality that uses Corosync Cluster Engine APIs.

A service engine designer implements the plug-in interface. This interface is a set of functions and data which are loaded dynamically. The service manager directs the service engine to execute functions. Some of the service engine functions then use four APIs which are registered with the service engine to execute the operations of the Corosync Cluster Engine.

5.2 Plug-In Interface

The full plug-in interface is a C structure depicted in Figure 8. The interface contains both data and function calls which are used by the service manager to direct the service engine plug-in.

The name field contains a character string which uniquely identifies the service engine name. This field is printed by the Corosync Cluster Engine to give status information to the user.

The id field contains a 16-bit unique identifier registered with the Corosync Cluster Engine. This unique identifier is used to route library and Totem requests to the proper service engine by the service manager.

When private data is needed to store state information, the interprocess communication manager allocates a block of memory of the size of the parameter private_data_size during initialization of the connection.

The exec_init_fn field is a function executed to initialize the service engine. The exec_exit_fn field is a function executed to request the service engine to shut down. When the administrator sends a SIGUSR2 signal to the Corosync Cluster Engine process, the state of the service engine is dumped to the logging system by the exec_dump_fn function.

The lib_init_fn field is a function executed when a new library connection is initiated to the service engine by the interprocess communication manager. The lib_exit_fn field is a function executed when the IPC connection is closed by the interprocess communication manager.

The main functionality of a service engine is managed by the service engine using the lib_engine and exec_engine parameters. These parameters contain arrays of functions which are executed by the service manager.

A service engine connection is routed to the proper lib_engine function by the service manager. When a library connection requests the service engine to execute functionality, the connection's id is used to identify the function in the array to execute. The lib_engine_count contains the number of entries in the lib_engine array.

The function then would generally use the various APIs available within the corosync_api_v1 structure to create timers, send Totem messages, or respond with a message using the interprocess communication manager.

When Totem messages are originated, they are delivered to the proper exec_engine function by the service manager to every processor in the cluster. The proper exec_engine function is called based upon the service id in the header of the function. The exec_engine_count contains the number of entries in the exec_engine array.

The design of a service engine should take advantage of the Totem ordering guarantees by executing most of the logic of a service engine in the exec_engine functions. These functions generally respond to the library request that originated the Totem message using the interprocess communication manager API.

5.3 Service Engine APIs

5.3.1 Overview

There are four sets of functionality within Corosync service engine APIs shown in Figure 9.

5.3.2 Timer API

The timer api allows a user-specified callback to be executed when a timer expires. Timers may either be defined as absolute or at some duration into the future.

The timer_add_duration() function is used to add a callback function that expires into a certain number of nanoseconds into the future. The timer_add_absolute() function is used to execute a callback at an absolute time as specified through the number of nanoseconds since the epoch.

If a timer has been added to the system, and later needs to be deleted before it expires, the designer can execute timer_delete() function to remove the timer.

Finally, a service engine can obtain the system time in nanoseconds since the epoch with the timer_get() function call.

5.3.3 Interprocess Communication Manager API

The Interprocess Communication Manager API includes functions to set and determine the source of messages, to obtain the IPC connection's private data store,

```
struct corosync_lib_handler {
  void (*lib_handler_fn) (void *conn, void *msg);
  int response_size;
  int response_id;
  enum corosync_flow_control flow_control;
};

struct corosync_exec_handler {
  void (*exec_handler_fn) (void *msg, unsigned int nodeid);
  void (*exec_endian_convert_fn) (void *msg);
};

struct corosync_service_engine {
  char *name;
  unsigned short id;
  unsigned int private_data_size;
  enum corosync_flow_control flow_control;
  int (*exec_init_fn) (struct objdb_iface_ver0 *, struct corosync_api_v1 *);
  int (*exec_exit_fn) (struct objdb_iface_ver0 *);
  void (*exec_dump_fn) (void);
  int (*lib_init_fn) (void *conn);
  int (*lib_exit_fn) (void *conn);
  struct corosync_lib_handler *lib_engine;
  int lib_service_count;
  struct corosync_exec_handler *exec_engine;
  int (*config_init_fn) (struct objdb_iface_ver0 *);
  int exec_service_count;
  void (*confchg_fn) (
    enum totem_configuration_type configuration_type,
    unsigned int *member_list, int member_list_entries,
    unsigned int *left_list, int left_list_entries,
    unsigned int *joined_list, int joined_list_entries,
    struct memb_ring_id *ring_id);
  void (*sync_init) (void);
  int (*sync_process) (void);
  void (*sync_activate) (void);
  void (*sync_abort) (void);
};

struct corosync_service_handler_iface_ver0 {
  struct corosync_service_handler *(*corosync_get_service_handler_ver0) (void);
};
```

Figure 8: The Service Engine Plug-In Interface

```
typedef void *corosync_timer_handle;

struct corosync_api_v1 {
  int (*timer_add_duration) (
    unsigned long long nanoseconds_in_future,
    void *data, void (*timer_nf) (void *data),
    corosync_api_handle_t *handle);

  int (*timer_add_absolute) (
    unsigned long long nanoseconds_from_epoch,
    void *data, void (*timer_fn) (void *data),
    corosync_timer_handle_t *handle)

  void (*timer_delete) (corosync_timer_handle_t timer_handle):

  unsigned long long (*timer_time_get) (void);

  void (*ipc_source_set) (mar_message_source_t *source, void *conn);

  int (*ipc_source_is_local) (mar_message_source_t *source);

  void *(*ipc_private_data_get) (void *conn);

  int (*ipc_response_send) (void *conn, void *msg, int mlen);

  int (*ipc_dispatch_send) (void *conn, void *msg, int mlen);

  void (*ipc_refcnt_inc) (void *conn);

  void (*ipc_refcnt_dec) (void *conn);

  void (*ipc_fc_create) (
    void *conn, unsigned int service, char *id, int id_len,
    void (*flow_control_state_set_fn)
      (void *context, enum corosync_flow_control_state flow_control_state_set),
    void *context);

  void (*ipc_fc_destroy) (
    void *conn, unsigned int service, unsigned char *id, int id_len);

  void (*ipc_fc_inc) (void *conn);

  void (*ipc_fc_dec) (void *conn);

  unsigned int (*totem_nodeid_get) (void);

  unsigned int (*totem_ring_reenable) (void);

  unsigned int (*totem_mcast) (struct iovec *iovec, int iov_len,
    unsigned int gaurantee);

  unsigned void (*error_memory_failure) (void);
};
```

Figure 9: The Service Engine APIs

and to send responses to either the response or dispatch socket descriptor. Messages are automatically delivered to the correct service engine depending upon parameters in the message header.

The `ipc_source_set()` will set a `mar_message_source_t` message structure with the node id and a unique identifier for the IPC connection. A service engine uses this function to uniquely identify the source of an IPC request. Later this `mar_message_source_t` structure is sent in a multicast message via Totem. Once this message is delivered, the Totem message handler then can respond to the ipc request by determining if the message was locally sent via `ipc_source_is_local()`.

Each IPC connection contains a private data area private to the IPC connection. This memory area is allocated on IPC initialization and is determined from the `private_data` field in the service engine definition. To obtain the private data, the function `ipc_private_data_get()` function is executed by the service engine designer.

Every IPC connection is actually two socket descriptors. One descriptor, called the response descriptor, is used for requests and responses to the library user. These requests are meant to block the third-party process using the Corosync Cluster Engine until a response is delivered. If the third-party process doesn't desire blocking behavior, but may want to execute a callback within a dispatch function, the service engine designer can use `ipc_dispatch_send()` instead.

There are other APIs which are useful to manage flow control, but they are complex to explain in a short paper. If a designer wants to use these APIs, they should consider viewing the Corosync Cluster Engine wiki or mailing list.

5.3.4 Totem API

The Totem API is extremely simple for service engines to use with only three API functions. These functions obtain the current node ID, allow a failed ring to be reenabled, and allow the multicast of a message. Conversely, most of the complexity of Totem is connected to the Corosync service engine interface and hidden from the user.

To obtain the current 32-bit node identifier, the function `totem_nodeid_get()` function can be called. This is useful when making comparisons of which node originated a message for service engines.

When Totem is configured for redundant ring operational mode, it is possible that an active ring may fail. When this happens, a service engine can execute `totem_ring_reenable()` via administrative operation to repair a failed redundant ring.

Service engines do a majority of their work by sending a multicast message and then executing some functionality based upon the multicasted message parameters. To multicast a message, an io vector is send via the `totem_mcast()` API. This message is then delivered to all nodes according to the extended virtual synchrony model.

5.3.5 Miscellaneous APIs

Currently many of the subsystems in the Corosync Cluster Engine are tolerant of failures to allocate memory. The exception to this rule may be the service engine implementations themselves. When a non-recoverable memory allocation failure occurs in a service engine, the api `error_memory_failure()` is called to notify the Corosync Cluster Engine that the service engine calling the function has had a memory malfunction.

In the future, the Corosync Cluster Engine designers intend to manage memory pools for service engines to avoid any out of memory conditions or memory process starvation.

6 Security Model

The Corosync Cluster Engine mitigates the following threats:

- Forged Totem messages intended to fault the Corosync Cluster Engine
- Monitoring of network data to capture sensitive cluster information
- Malformed IPC messages from unprivileged users intended to fault the Corosync Cluster Engine

The Corosync Cluster Engine mitigates those threats via two mechanisms:

- Authentication of Totem messages and IPC users
- Secrecy of Totem messages with the usage of encryption

7 Integration with Third Party Projects

7.1 OpenAIS

OpenAIS [OpenAIS] is an implementation of the Service Availability Forum's Application Interface Specification. The specification is a C API designed to improve availability by reducing the mean time to repair through redundancy.

Integration with OpenAIS was a simple task since a majority of the Corosync functionality was reduced from the OpenAIS code base. When OpenAIS was split into two projects, some of the internal interfaces used by plug-ins changed. The usage of these internal APIs were modified to the definitions described in this paper.

7.2 OpenClovis

OpenClovis [OpenClovis] is an implementation of the Service Availability Forum's Application Interface Specification. OpenClovis uses some portions of the Corosync services. Specifically, it uses the Totem protocol APIs to provide membership for its Cluster Membership API.

7.3 OCFS2

The OCFS2 [OCFS2] filesystem can use the closed process group api to communicate various pieces of state information about the mounted cluster. Further the CPG service is used for supporting Posix Locking because of the virtual synchrony feature of the closed process group service.

7.4 Pacemaker

Pacemaker [Pacemaker] is a scalable High-Availability cluster resource manager formerly part of Heartbeat [LinuxHA]. Pacemaker was first released as part of Heartbeat-2.0.0 in July 2005 and overcame the deficiencies of Heartbeat's previous cluster resource manager:

- Maximum of 2-nodes
- Highly coupled design and implementation
- Overly simplistic group-based resource model
- Inability to detect and recover from resource-level failures
- Pacemaker is now maintained independently of Heartbeat in order to support both the OpenAIS and Heartbeat cluster stacks equally.

Pacemaker functionality is broken into logically distinct pieces, each one being a separate process and able to be rewritten/replaced independently of the others:

- cib—Short for Cluster Information Base. Contains definitions of all cluster options, nodes, resources, their relationships to one another and current status. Synchronizes updates to all cluster nodes.
- lrmd—Short for Local Resource Management Daemon. Non-cluster aware daemon that presents a common interface to the supported resource types. Interacts directly with resource agents (scripts).
- pengine—Short for Policy Engine. Computes the next state of the cluster based on the current state and the configuration. Produces a transition graph contained a list of actions and dependencies.
- tengine—Short for Transition Engine. Coordinates the execution of the transition graph produced by the Policy Engine.
- crmd—Short for Cluster Resource Management Daemon. Largely a message broker for the PE, TE, and LRM. Also elects a leader to co- ordinate the activities of the cluster.

The Pacemaker design of one process per feature presented an interesting challenge when integrating with Corosync which uses plug-ins/service engines to expand its functionality. To simplify the task of porting to the Corosync Cluster Engine, a small plug-in was created to provide the services traditionally delivered by Heartbeat.

At startup, the Pacemaker service engine spawns Pacemaker processes and respawns them in the event of failure. Cluster-aware components connect to the plug-in using the interprocess communication manager. Those applications can then send and receive cluster messages, query the current membership information, and receive updates.

The Pacemaker components use the Pacemaker service engine features indirectly via an informal API which is used to hide details of the chosen cluster stack. The abstraction layer can automatically determine the operational stack and chose the correct implementation at runtime by checking the runtime environment. Once the Pacemaker service engine and abstraction layer were functional, Pacemaker was made stack independent, as shown in Figure 10, with little effort.

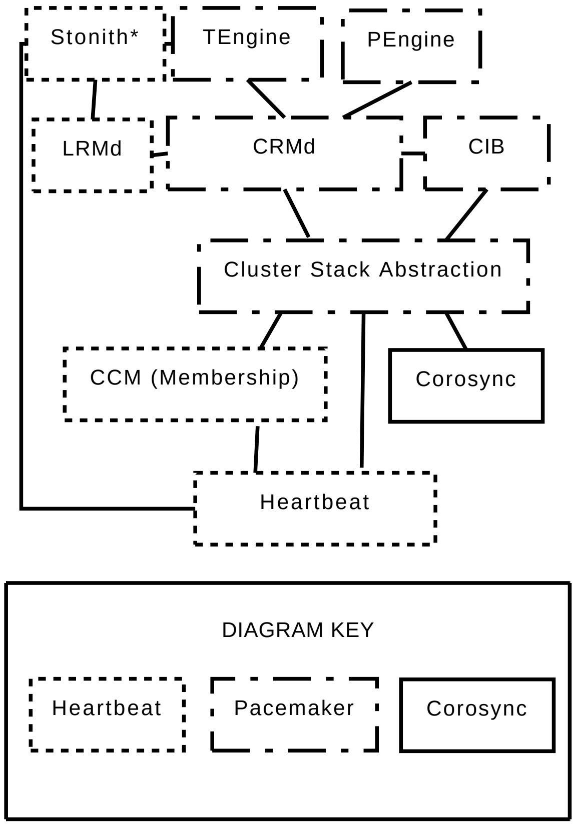

Figure 10. Pacemaker Dual Stack Architecture

Pacemaker components exchange messages consisting mostly of compressed XML-formatted strings. Representing the payload as XML is not efficient, but the format's verboseness means it compresses well, and complex objects are easily unpackable by numerous custom and standard libraries.

In order to accommodate Pacemaker, the Corosync Cluster Engine designers added ordered service engine shutdown. When an administrator or another service engine triggers a shutdown of the Corosync Cluster Engine, the service engines clean up and exit gracefully. This allows the Pacemaker service engine to organize for resources on a node be to migrated away gracefully and eventually stop its child processes before the Corosync Service Engine process exits.

7.5 Red Hat Cluster Suite

The Red Hat Cluster Suite [RHCS] version 3 uses the Corosync Cluster Engine. The Red Hat Cluster Suite uses a service engine called CMAN to provide services to other Red Hat Cluster Suite services.

Quorum is the main function of the CMAN service and is a strong dependency in all of the Red Hat Cluster Suite software stack. Quorum ensures that the cluster is operating consistently with more then half of the nodes operational. Without quorum, filesystems such as the Global File System can lead to data corruption.

The Quorum disk software communicates with the CMAN service via an API. The quorum disk software provides extra voting information to help the infrastructure identify when quorum has been met for special criteria.

Red Hat Cluster Suite uses a distributed XML-based configuration system called CCS. CMAN provides a configuration engine which reads Red Hat Cluster Suite specific configuration format files and stores them within the object database. This configuration plug-in overrides the default parsing of the `/etc/corosync/corosync.conf` configuration file format.

The libcman library provides backwards compatibility with the cman-kernel in Red Hat Enterprise Linux 4. This backwards compatibility is used by a few applications such as CCS, CLVMD, and rgmanager.

Red Hat Cluster Suite, and more specifically the Global File System component, makes use of the Closed Process Groups interface that is standardized within the CPG interface included in the Corosync Cluster Engine.

8 Future Work

The Corosync Cluster Engine Team intends to improve the scalability of the engine. Currently, the engine has been used in a physical 60 node cluster. The engine has been tested in a 128 node virtualized environment.

While these environments demonstrated the Corosync Cluster Engine works properly at large processor counts, the team wants to improve scalability to even larger processor counts and reduce latency while improving throughput.

The Corosync Cluster Engine designers desire to add a generically useful quorum plug-in engine so that any project may define its own quorum system.

Finally, the team wishes to add a generic fencing engine and mechanism for multiple plug-in services to determine how to fence cooperatively.

9 Conclusion

This paper has presented a strong rationale for using the Corosync Cluster Engine and demonstated the design is generically useful for a variety of third-party cluster projects. This paper has also presented the current architecture and plug-in developer application programming interfaces. Finally, this paper has presented a brief overview of some of our future work.

References

[Corosync] The Corosync Cluster Engine Community. *The Corosync Cluster Engine*, `http://www.corosync.org`

[Amir95] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. *The Totem Single-Ring Ordering and Membership Protocol*, ACM Transactions On Computer Systems, 13(4), pp. 311–342, November, 1995. `http://www.cs.jhu.edu/~yairamir/archive.html`

[Moser94] L.E. Moser, Y. Amir, P.M. Melliar-Smith, and D.A. Agarwal. *Extended Virtual Synchrony*, ACM Transactions on Computer Systems 13(4):311-342, November, 1995. Proceedings of DCS, pp. 56–65, 1994.

[Dake05] S. Dake and M. Huth. *Implementing High Availability Using the SA Forum AIS Specification*, Embedded Systems Conference, 2005.

[SaForumAIS] Service Availability Forum. *The Service Availability Forum Application Interface Specification*, `http://www.saforum.org/specification/download`

[Birman93] K.P. Birman. *The Process Group Approach to Reliable Distributed Computing*, Communications of the ACM 36(12): 36-56, 103, 1993.

[Koch02] R.R. Koth, L.E. Moser, and P. M. Melliar-Smith. *The Totem Redundant Ring Protocol*, ICDCS 2002:598-607.

[Postel80] J. Postel. *User Datagram protocol*, Darpa Internet Program RFC 768, August 1980.

[USC81] University of Southern California. *Internet Protocol*, Darpa Internet Program RFC 791, September 1981.

[Deering98] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*, IETF Network Working Group, December 1998.

[OpenAIS] The OpenAIS Community. *The OpenAIS Standards Based Cluster Framework*, `http://www.openais.org`

[OpenClovis] The OpenClovis Company. *OpenClovis*, `http://www.openclovis.org`

[OCFS2] The Oracle Cluster File System Community. *The Oracle Cluster Filesystem*, `http://oss.oracle.com/projects/ocfs2`

[Pacemaker] The Pacemaker Community. *Pacemaker*, `http://www.clusterlabs.org`

[LinuxHA] The Linux-HA Community. *Linux-HA*, `http://www.linux-ha.org`

[RHCS] Red Hat Cluster Suite. *The Linux Cluster Community Project*, `http://sources.redhat.com/cluster/wiki`

Proceedings of the Linux Symposium

Volume One

July 23rd–26th, 2008
Ottawa, Ontario
Canada