The Corosync High Performance Shared Memory IPC Reusable C Library

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

Abstract

The Corosync coroipc reusable C libraries providing high performance client server communication are presented. The rationale for this effort is provided. An overview of the coroipc features are given. The programming API is described in enough detail to provide developers with a complete understanding of how to develop a client server application. Finally performance results are provided.

1 Introduction

The Corosync Cluster Engine project was created in July 2008 to address the needs of the Linux clustering community. As part of this effort, the project implemented and qualified a high performance client server interprocess communication system called coroipc.

Throughout the history of client server applications, every project implemented a unique IPC system. These IPC systems each contain a unique set of defects, performance characteristics, security model, thread safety, and portability support. After developing coroipc, the Corosync community determined coroipc could be modified to be reusable by third party client server applications.

By making coroipc reusable, coroipc enables consuming projects to focus on their strengths. Further by centralizing development effort on one IPC system, a larger community of experienced designers can provide support for that IPC system. Finally since coroipc is built into a significant portion of the Linux community's cluster infrastructure, it provides a perfect environment for ensuring the software has a sanitary design model and is defect free.

2 Features

2.1 Security

The coroipcs library provides a mechanism to ensure only users with specific user id or group id access the IPC system, and by inference, the server. This is enforced on all platforms which support the ability to retrieve the uid or gid of a connecting socket from a platform-specific system call.

2.2 High Performance

The coroipc client and server use almost exclusively the `mmap()` system call to map shared memory. As a result, in most cases there is no copy into the kernel, or from the kernel to userspace for communications. Notification of new messages occurs through a system V semaphore.

2.3 Portability

The coroipc system is dependent upon a Posix API, a coherent `mmap()` system call, and system V semaphores. Nearly all modern Posix platforms provide these features. The coroipc system has been ported and tested on Linux, BSD, Darwin, and Solaris.

2.4 Thread Safety

The coroipc client library is thread safe and requires no special attention by the client library users to ensure thread safety. Thread safety is implemented using reference counting on the identifier used for a client IPC connection. The reference counting critical sections are protected by spinlocks on platforms which support them, or a mutex on platforms without spinlocks.

2.5 Zero Copy

Clients may allocate a zero copy buffer which removes one copy from client requests. Allocating a zero copy buffer is an expensive operation and is reserved for buffers with a consistent size which are consistently reused.

2.6 Support for External Poll Systems

The coroipc server allows the server developer to use customized polling mechanisms. Currently there are no examples of using third party polling systems beyond the coropoll API provided with the software. We expect a glib example to be available in the community.

2.7 Asynchronous Client Delivery

The coroipc client blocks when the waiting for a server response. If the server takes long periods to process requests, it may prefer to issue an asynchronous response to unblock the client. The coroipc system supports the delivery of these messages through a special channel called the dispatch channel.

3 Architecture Overview

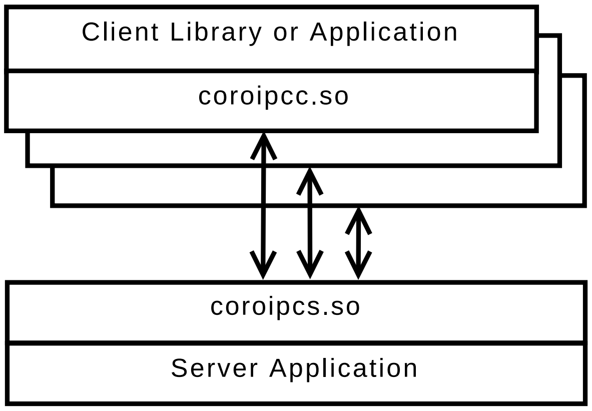

Figure 1: Example client-server application

The coroipc system is composed of two major components. The client component is composed of a client header file called coroipcc.h and client shared library called coroipcc.so. The server component is composed of a server header file called coroipcs.h and server shared library called coroipcs.so. Figure 1 an example client server application with multiple clients communicating to one server.

The global header file `coroipc_types.h` is shown in Listing 1. Every request message sent by library coroipcc clients should begin with a `coroipc_request_header_t`. The `size` parameter should be set to the size of the message and the `id` parameter should be set to the message identifier.

The server coroipcs handlers should format a message with a header of `coroipc_response_header_t`. The coroipcc clients should expect to receive a message with the `coroipc_response_header_t` header.

```
typedef struct {
        int size;
        int id;
} coroipc_request_header_t;

typedef struct {
        int size;
        int id;
        cs_error_t error;
} coroipc_response_header_t;
```

Listing 1: The coroipcc Types Definition

4 coroipcc

The coroipcc library provides lifecycle operations, dispatch operations, request and reply operations, and zero copy buffer operations. The full API is shown in Listing 2.

4.1 Lifecycle Operations

Clients connect to servers using the `coripcc_service_connect()` API. When a client connects, the client and server both `mmap()` several files into memory shared by the client and server. Finally a semaphore set is created to provide signalling between client and server of new messages.

Several files are mapped using the `mmap()` system call into the address space of both the client and server. The first of these files is the control buffer which is used internally for communication between the client and server. A unique file is also mapped for client to server requests and server to client responses. Finally an

```
extern cs_error_t
coroipcc_service_connect (const char *socket_name, unsigned int service,
        size_t request_size, size_t respnse_size, size_t dispatch_size,
        hdb_handle_t *handle);

extern cs_error_t
coroipcc_service_disconnect (hdb_handle_t handle);

extern cs_error_t
coroipcc_fd_get (hdb_handle_t handle, int *fd);

extern cs_error_t
coroipcc_dispatch_get (hdb_handle_t handle, void **buf, int timeout);

extern cs_error_t
coroipcc_dispatch_put (hdb_handle_t handle);

extern cs_error_t
coroipcc_dispatch_flow_control_get (hdb_handle_t handle,
        unsigned int *flow_control_state);

extern cs_error_t
coroipcc_msg_send_reply_receive (hdb_handle_t handle, const struct iovec *iov,
        unsigned int iov_len, void *res_msg, size_t res_len);

extern cs_error_t
coroipcc_msg_send_reply_receive_in_buf_get (hdb_handle_t handle,
        const struct iovec *iov, unsigned int iov_len, void **res_msg);

extern cs_error_t
coroipcc_msg_send_reply_receive_in_buf_put (hdb_handle_t handle);

extern cs_error_t
coroipcc_zcb_alloc (hdb_handle_t handle, void **buffer, size_t size,
        size_t header_size);

extern cs_error_t
coroipcc_zcb_free (hdb_handle_t handle, void *buffer);

extern cs_error_t
coroipcc_zcb_msg_send_reply_receive (hdb_handle_t handle, void *msg,
        void *res_msg, size_t res_len);
```

Listing 2: The coroipcc C API

asynchronous dispatch buffer is mapped twice to avoid copies during dispatch operations.

Clients may disconnect via the `coroipcc_service_disconnect()` API. A disconnect doesn't actually occur until all references of the ipc connection have been released.

4.2 Dispatch Operations

Client server applications may desire asynchronous communication. In coroipc, these are called dispatch operations.

To determine when a dispatch operation is available, the `poll()` system call should be used on a file descriptor obtained with `coroipcc_fd_get()`.

To retrieve the current dispatch buffer contents, the `coroipcc_dispatch_get()` API is called. The dispatch buffer is implemented internally as a circular buffer. To avoid copies, the operating system virtual memory system is used to provide a circular buffer mapping. The `coroipcc_dispatch_get()` operation pins the dispatch message. It can be then be released with `coroipcc_dispatch_put()`.

The design of coroipcc requires there is only one thread of execution which executes a `coroipcc_dispatch_get()` and `coroipcc_dispatch_put()` operation. This model is consistent with the user of coroipc providing an API to handle asynchronous dispatching of events and using the internal coroipcc dispatch operation functions.

The `coroipcc_fd_get()` API does not have to be used for those client applications which don't need to multiplex input/output operations in the client. Instead `coroipcc_dispatch_get()` may be used directly and will use semaphores to avoid busy spins.

4.3 Request and Reply Operations

The coroipcc library provides message request and reply operations to allow requests to be sent synchronously to a server and a reply to be received from the server. The common API is `coroipcc_msg_send_reply_receive()` which copies the response into a user supplied buffer. The remaining two APIs allow zero copy reading of the response buffer by pinning the response buffer into memory. Pinning is done via `coroipcc_msg_send_reply_receive_in_buf_get()` and an unpin operation occurs via `coroipcc_msg_send_reply_receive_in_buf_put()`.

4.4 Zero copy buffer operations

To provide zero copy requests, the client must allocate memory in both the client and server and share it via `mmap()`. The client requests the server allocate this shared memory via `coroipcc_zcb_alloc()` and free the memory via `coroipcc_zcb_free()`. Since these operations are expensive, they should be rarely done and zero copy buffering should only be used on often reused buffer areas. To send a request and receive a reply, the API `coroipcc_zcb_msg_send_reply_receive()` is used.

5 coroipcs

Servers link with the coroipcs library and include the coroipcs.h file to access coroipcs services. The coroipcs library includes lifecycle operations, response operations, and integration with third party polling systems.

5.1 Lifecycle Operations

The coroipcs system is initialized by `coroipcs_init()` and exited by `coroipcs_exit()`. The initialization is defined by the structure `coroipcs_init_state` shown in Listing 3. This structure includes many user provided function parameters. These routines include scheduling policy, memory mangement, serialization, flow control, security, custom poll handler control, and functions to retrieve operations of the user service.

5.1.1 Scheduling Policy

The coroipcs threads may be scheduled at Posix scheduling policies rather then the default scheduler. The `policy` parameter to `coroipcs_init` controls the policy and the `sched_param` parameter controls the parameters related to the policy.

5.1.2 Memory Management

Many servers provide their own memory allocation. In that case, the internal use of `malloc()` and `free()` can be overridden with user defined functions.

5.1.3 Serialization

If the backend function handlers are not thread safe, the user may provide a `serialize_lock()` function that is executed when the service function callbacks are called and `serialize_unlock()` function that is executed when the service function callback is done with execution. This acts to serialize input into the service so no extra mutual exclusion is needed. If high concurrency is desired, these functions can be defined to NULL and will not be used. Instead the user should provide finer grained locking within their callbacks.

5.1.4 Flow Control

Two functions are provided to provide flow control into the server handler callbacks determined by the `handler_fn_get()` callback.

```
typedef int (*coroipcs_init_fn_lvalue) (void *conn);
typedef int (*coroipcs_exit_fn_lvalue) (void *conn);
typedef void (*coroipcs_handler_fn_lvalue) (void *conn, const void *msg);

struct coroipcs_init_state {
  const char *socket_name;
  int sched_policy;
  const struct sched_param *sched_param;
  void *(*malloc) (size_t size);
  void (*free) (void *ptr);
  void (*log_printf) ( const char *format, ...) __attribute__((format(printf, 1, 2)));
  int (*service_available) (unsigned int service);
  int (*private_data_size_get) (unsigned int service);
  int (*security_valid)(int uid, int gid);
  void (*serialize_lock)(void);
  void (*serialize_unlock)(void);
  int (*sending_allowed) (unsigned int service, unsigned int id, const void *msg,
    void *sending_allowed_private_data);
  void (*sending_allowed_release) (void *sending_allowed_private_data);
  void (*poll_accept_add) (int fd);
  void (*poll_dispatch_add) (int fd, void *context);
  void (*poll_dispatch_modify) (int fd, int events);
  void (*poll_dispatch_destroy) (int fd, void *context);
  void (*fatal_error) (const char *error_msg);
  coroipcs_init_fn_lvalue (*init_fn_get) (unsigned int service);
  coroipcs_exit_fn_lvalue (*exit_fn_get) (unsigned int service);
  coroipcs_handler_fn_lvalue (*handler_fn_get) (unsigned int service, unsigned int id);
};
```

Listing 3: The init state structure

The `sending_allowed()` function determines if an IPC message may be delivered to the server. If it returns the value 1, the coroipcs library will deliver the IPC message to the appropriate server handler.

After an IPC message is delivered, the `sending_allowed_release()` callback is executed.

It is often helpful to store some private information for these two functions to share their operating state. A 64 byte parameter `sending_allowed_private_data` is passed to both functions to store this operational state. The use of this private data is optional and invisible to the coroipcs library.

5.1.5 Security

The `security_valid()` function is called by coroipcs when a new IPC connection is made to the system. The uid and gid are passed as parameters to this function. The function should return 1 if the uid or gid are valid users of the coroipcs application, otherwise it should return 0.

5.1.6 Poll Handling

The `poll_dispatch_add()` call is executed when a dispatch routine is required to be added to the poll loop. The `poll_dispatch_modify()` is used modify the events on the existing file descriptor. The `poll_dispatch_destroy()` removes the fd from the polling system.

5.1.7 Function Retrieval

The coroipcs system works by retrieving a function from user defined selectors and executing those functions when the appropriate action is requested by the ipc client library. The `init_fn_get()` function is called to retrieve the initialization function for the service. When the ipc connection disconnects, the `exit_fn_get()` function is called to retrieve the exit function for the ipc connection. Finally `handler_fn_get()` is used to retrieve the appropriate IPC handler.

```
extern void coroipcs_ipc_init (struct coroipcs_init_state *init_state);

extern void
*coroipcs_private_data_get (void *conn);

extern int
coroipcs_response_send (void *conn, const void *msg, size_t mlen);

extern int
coroipcs_response_iov_send (void *conn, const struct iovec *iov, unsigned int iov_len);

extern int
coroipcs_dispatch_send (void *conn, const void *msg, size_t mlen);

extern int
coroipcs_dispatch_iov_send (void *conn, const struct iovec *iov, unsigned int iov_len);

extern void
coroipcs_refcount_inc (void *conn);

extern void
coroipcs_refcount_dec (void *conn);

extern void
coroipcs_ipc_exit (void);

extern int
coroipcs_handler_accept (int fd, int revent, void *context);

extern int coroipcs_handler_dispatch (int fd, int revent, void *context);
```

Listing 4: The coroipcs API

5.2 Response Handling

The IPC services on delivery of a message can respond via the APIs shown in Listing 4. More specifically a regular response can be sent via `coroipcs_response_send()` or via iovectors with `coroipcs_response_iov_send()`. To send to the dispatch output channel, `coroipcs_dispatch_send()` can be used and and iovector version is also available with `coroipcs_dispatch_iov_send()`.

5.3 Custom Poll Handling

The init functions specified in the structure `coroipcs_init_state` for `poll_accept_add()` and `poll_dispatch_add()` should register callbacks with the poll system which then call the external coroipcs APIs `coroipcs_handler_accept()` and `coroipcs_handler_dispatch()` respectively. The purpose of the external APIs is to translate whatever API the application uses for poll into function calls the coroipcs system can understand.

6 Performance

The coroipcs system is designed for high concurrency operation on multiple processors. Each IPC connection is represented in the OS by a separate scheduling entity to allow multi-threaded server designs. As a result, coroipcs should better be able to utilize the operating system scheduling features to achieve higher concurrency then single threaded server applications.

The throughput in megabytes per second for message sizes ranging from 1000 to 500,000 in 1000 byte increments is shown in Figure 2. As can be seen from the Figure 2, newer processor designs have higher total throughput of up to 30 GB/sec for larger message sizes. Older processor designs reach maximum throughput of 6 GB/sec for larger message sizes. The dropoff for very large message sizes on 4 client Nehalem processors is

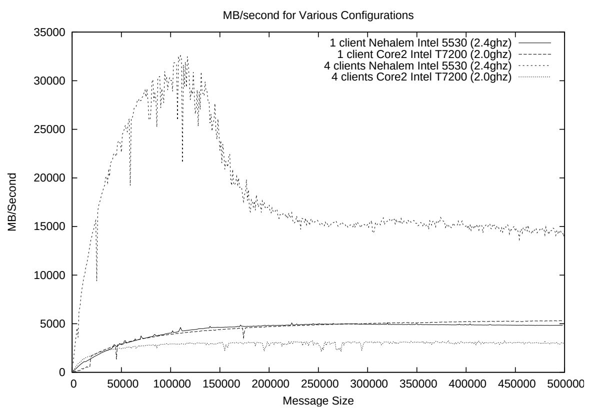

Figure 2: MB/Sec Throughput

unexplained but may be a result of cache behavior of the processor.

Transactions per second is shown in Figure 3. Nahalem with 4 clients in this graph shows very good results of one million transactions per second while a single client shows results of 100,000 transactions per second. As the size of the message increases, more time is spent within the `memcpy()` C library function resulting in lower overall transaction rates.

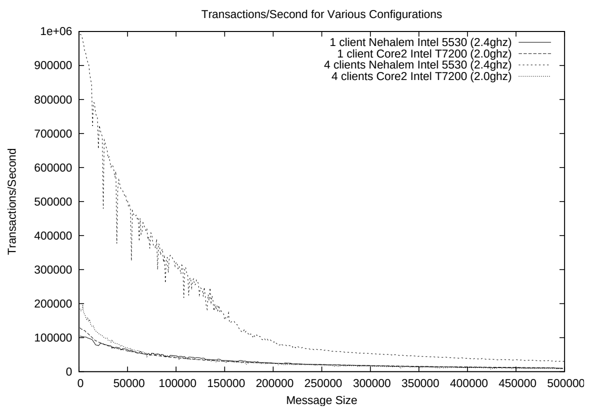

Figure 3: Transactions/Sec Throughput

7 Future Work

One area for future development is the tracking and notification of buffer lengths to prevent a blocked client from triggering server memory pressure. There are many choices for how this could be done and remains a further area of investigation.

Currently when a coroipcc request is made, a mutex is taken on the shared memory area responsible for requests and responses. This blocks other requests on the same handle instance from proceeding until a response is made. To improve concurrency, we plan to investigate removal of the mutex requirement by allowing multiple requests and responses to be mapped into the shared memory segment for multithreaded high concurrency applications.

Since our implementation just concluded, we have not had a thourough chance to optimize small message sizes for maximum MB/sec and transactions/sec throughput. We intend to further analyze and characterize the performance of coroipc to find hot spots within the implementation and make improvements where possible.

8 Conclusion

The coroipc system is a reusable C library that meets the general needs of many client server applications. It is portable to most Posix platforms, provides a sanitary security model, and is thread safe for both clients and servers. While improvements can be made with performance, the performance of the initial implementation is very good for many workload combinations. Our initial requirements of an IPC system are met satisfactorily and we expect future work to provide improved performance and usability.

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada