Project 2

Inter-Process Communication Services

6210: Advanced Operating Systems

Spring 2018

Submitted by:

Assil Ksiksi (gtID: 903237927)

Submitted on: March 3rd 2018

Table of Contents

Overview

In this report, we present the design and implementation of an IPC client library and accompanying server called *GTIPC*.

The goal of this project is to implement a low-level IPC client/server protocol and interface targeting the Unix platform.

The main requirements for this project are:

1. Implement a blocking IPC client that can communicate with some server process.
2. Implement a non-blocking/asynchronous IPC client that performs the same tasks.
3. Design a client API and library that can be used in a generic C/C++ Unix application.
4. Present a sample application that utilizes the aforementioned client API.

GTIPC High-Level Design

The system design is presented in the following figure:

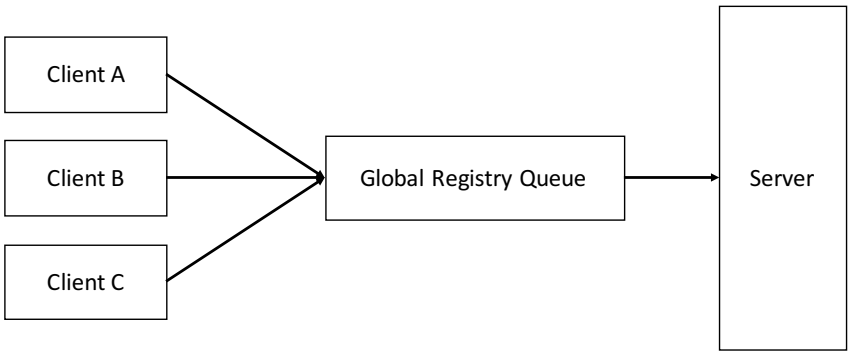

Figure 1: High level IPC system design

GTIPC allows multiple clients – three in the figure – to send requests and pass arguments to a server process. Clients register themselves with the server through a **global registry queue**.

Figure 2 depicts the interface between a *single* client and the server process. The client creates two POSIX message queues and a dynamic POSIX shared memory segment. The names of the queues and shared memory are passed to the server via the global registry.

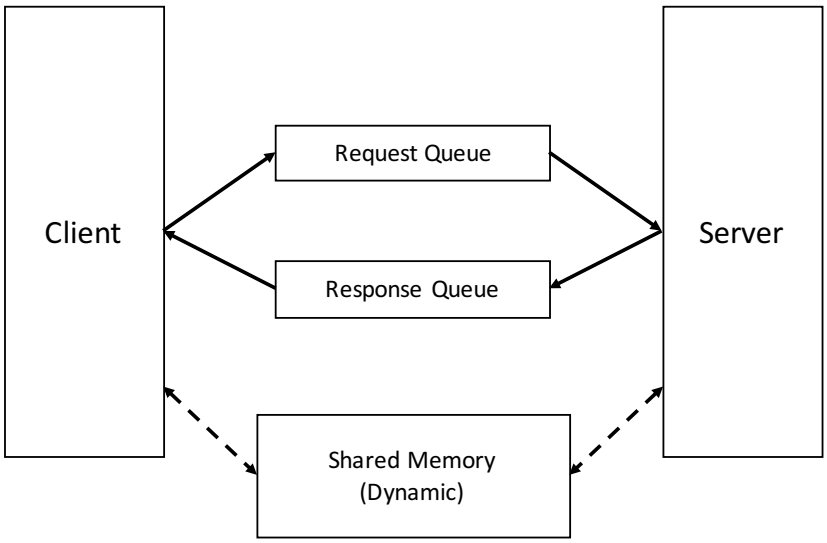

Figure 2: Single client and server interface

GTIPC Services

The GTIPC API interface provides clients with three IPC services:

1. A service that can multiply two 32-bit integers (*MUL* service).
2. A service that generates four pseudorandom 32-bit integers (*RAND* service).
3. A service that creates and/or appends a line of text to a given file path (*FILE* service).

The above services are requested by the client and executed by the server, which implies that the actual implementations of the services reside *only* in the server process and can therefore only be reached via IPC.

GTIPC Client Library

Figure 3 shows a slightly detailed overview of the GTIPC client library.

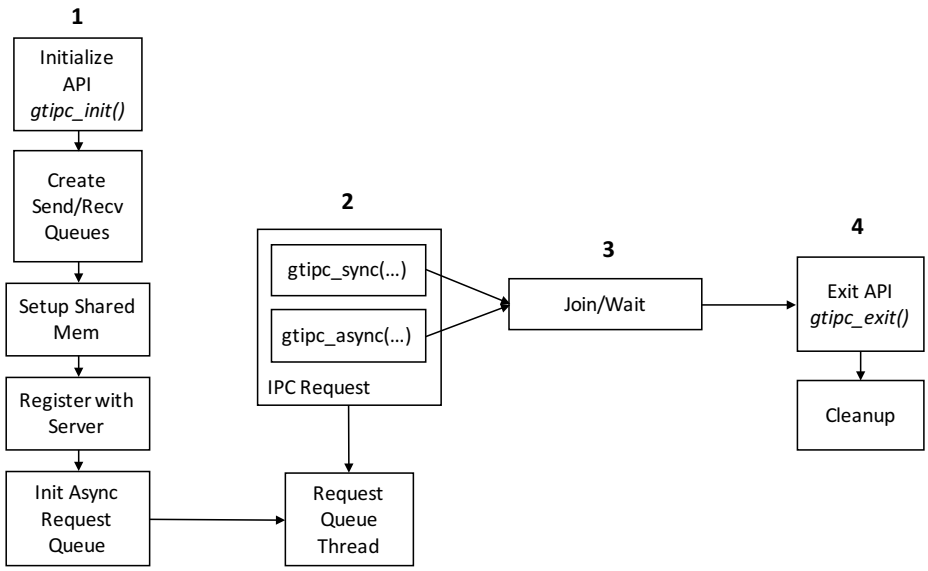

Figure 3: Overview of the GTIPC client library

The entry point for the library is *gtipc_init()*. Once the library is initialized, a request queue thread is spawned. The client application performs IPC requests by calling either *gtipc_sync()* or *gtipc_async()*. The client can join any group of async requests. Shutdown and cleanup for the library takes place in *gtipc_exit()*.

Usage

To bundle the GTIPC library with your own application, you simply need to link in *libgtipc.a* and include the header *"gtipc/api.h"*. Refer to the bundled README and Makefile for more information.

Synchronous Interface

```
int gtipc_sync(gtipc_arg *arg, gtipc_service service, gtipc_request_prio prio,
               gtipc_arg *out)
```

Takes an argument, a service identifier, a request priority, and provides the output argument in *out. Blocking call. Returns 0 if no error occurred, or an error number from one of those listed in the header file otherwise.

Asynchronous Interface

```
int gtipc_async(gtipc_arg *arg, gtipc_service service, gtipc_request_prio prio,
                gtipc_request_key *key)
```

Takes an argument, a service identifier, a request priority, and provides the a unqiue key that allows the caller to retrieve the output argument at a later time. Non-blocking call. Returns 0 if no error occurred, or an error number from one of those listed in the header file otherwise.

```
int gtipc_async_wait(gtipc_request_key key, gtipc_arg *arg)
```

Given a request key (as provided by *gtipc_async*), blocks until the corresponding result is ready and writes the output to *arg.

```
int gtipc_async_join(gtipc_request_key *keys, gtipc_arg *args, int size)
```

Given an array of request keys, blocks until *all* results are ready and writes the outputs the provided *args array. This is done by iterating over each request key and checking whether or not the request is done processing. The function only when all requests have been marked as done.

GTIPC Arguments

Arguments are passed to the client library as *gtipc_arg* objects. A *gtipc_arg* is a **union** containing either a *gtipc_mul_arg*, a *gtipc_rand_arg*, and a *gtipc_file_arg*. See *gtipc/types.h* for the structure of the underlying arguments.

```
/**
 * Generic argument to GTIPC API
 *
 * Only one of the members should be set!
 */
typedef union __gtipc_arg {
    gtipc_mul_arg mul;
    gtipc_rand_arg rand;
    gtipc_file_arg file;
} gtipc_arg;
```

Figure 4: GTIPC library argument structure

We used a union to make it easier to extend the library as services are added or removed.

Please note that arguments in the GTIPC API are used both to pass inputs from client to server as well as retrieve results from the server at the client side.

Client Initialization

As noted above, the main entry point for the client library is *gtipc_init()*, so this function **must** be called before the IPC library can be used in an application.

The init function performs the following steps:

1. Obtains a handle to the global registry queue.
2. Creates the send and receive POSIX message queues (*mq_open()*).
3. Creates a POSIX shared memory object (*shm_open()*) and maps it to the client's address space (*mmap()*).
4. Sends the register message via the global registry queue (see Figure 5).

5. Initializes the async request queue and spawns the request queue handler thread which waits for requests to be pushed to the request queue.

Client Registry

GTIPC clients send **registry messages** (Figure 5) to let the server know that the client exists and to provide the server with pertinent information for IPC.

```
typedef struct __gtipc_registry {
    // Register or unregister current client
    gtipc_registry_cmd cmd;

    // Client's PID
    pid_t pid;

    // Send and receive queue names
    char send_queue_name[100];
    char recv_queue_name[100];

    // Shared memory name
    char shm_name[100];
} gtipc_registry;
```

Figure 5: GTIPC registry message

GTIPC Request

The send message queue – i.e., from client to server – uses ***gtipc_request*** as an entry format (Figure 6). Each request represents a service call initiated by the client application through the GTIPC library.

```
/**
 * IPC request from client to server to initiate a service.
 */
typedef struct __gtipc_request {
    gtipc_service service;    // Requested IPC service
    gtipc_request_prio prio;  // Request priority
    int request_id;           // ID for current request
    int entry_idx;            // Index in shared mem for request's gtipc_shared_entry
    int pid;                  // Client PID
} gtipc_request;
```

Figure 6: GTIPC request format (sent from client to server)

The *prio* field allows library users to control the relative priority of messages delivered on the message queue. On the other hand, *entry_idx* is an index relative to the base of the shared memory segment and is used to lookup arguments and write results for a particular request. The remaining fields should be self-explanatory.

Library Message Queues

The client library manages two POSIX message queues: a send queue and a receive queue. The send queue is to send request (*gtipc_request*) information to the server via IPC, while the receive queue is used to get notifications from the server.

Due to the size limitation of POSIX message queues in userland (max size is 10!), we implemented a request queue that feeds into the POSIX send message queue (Figure 7).

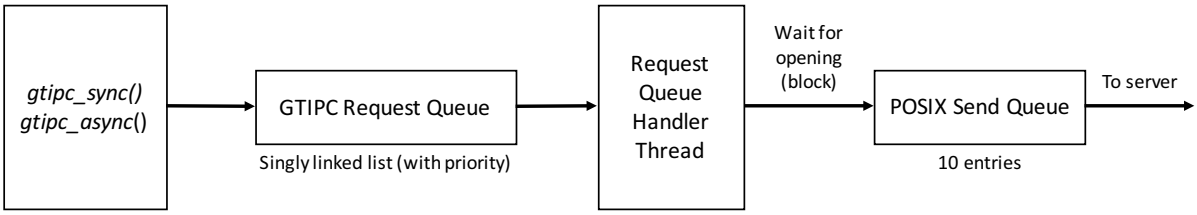

Figure 7: Client library request queue management

Whenever a call to either the sync or async library functions is made, the request object is buffered until a spot is available on the POSIX send queue. A background thread blocks on *mq_send()* and pulls in requests from the buffer queue whenever a spot opens up. In general, this approach allows for fast asynchronous request dispatch, regardless of the underlying POSIX queue size.

Library Shared Memory

The client library allocates a region of shared memory – initially 1024 *gtipc_shared_entry* objects – that is used to transmit arguments and results between the client and server. Note that each client shares its own dedicated region of shared memory with the server.

A *gtipc_shared_entry* is structured as follows:

```
/**
 * Entry in the shared memory segment accessible by both client and server.
 * Used to pass requests and arguments back and forth.
 */
typedef struct __gtipc_shared_entry {
    int used;
    int done;
    gtipc_arg arg;
    pthread_mutex_t mutex;
} gtipc_shared_entry;
```

Figure 8: Structure of gtipc_shared_entry in POSIX shared memory

Each shared entry maintains two flags: *used* and *done*. When the shared memory is first initialized, all entries have *used* set to 0, indicating that the entries are free to be used. The *done* flag is used by the server to indicate that the accompanying *gtipc_arg* has been updated with the result.

A mutex (*pthread_mutex_t*) is included with each shared entry to coordinate access between client and server. The mutex is set to *PTHREAD_PROCESS_SHARED* to provide entry access synchronization across different threads **and** processes. The mutex is acquired whenever *used* or *done* is read or updated.

We saw above that each *gtipc_request* object contains a field *entry_idx* which is used to look up the request's arguments in shared memory. The *entry_idx* is simply an integer offset to the location of the request's *gtipc_shared_entry* relative to the base of the POSIX shared memory segment.

Resizing Shared Memory

The client library reallocates shared memory based on underlying demand. In our implementation, once the shared memory is half full, the client and server coordinate with each other to create a **new** shared memory segment and copy over the old data to the new

segment (see: *resize_shm_object()*). Making sure that all worker threads on the server were paused and that the server memory is synchronized was fairly challenging, but I got it done!

GTIPC Server Process

The GTIPC server is responsible for registering and unregistering GTIPC clients and, more importantly, serving client requests. Figure 9 depicts a high-level flow of the server and the components involved in its operation.

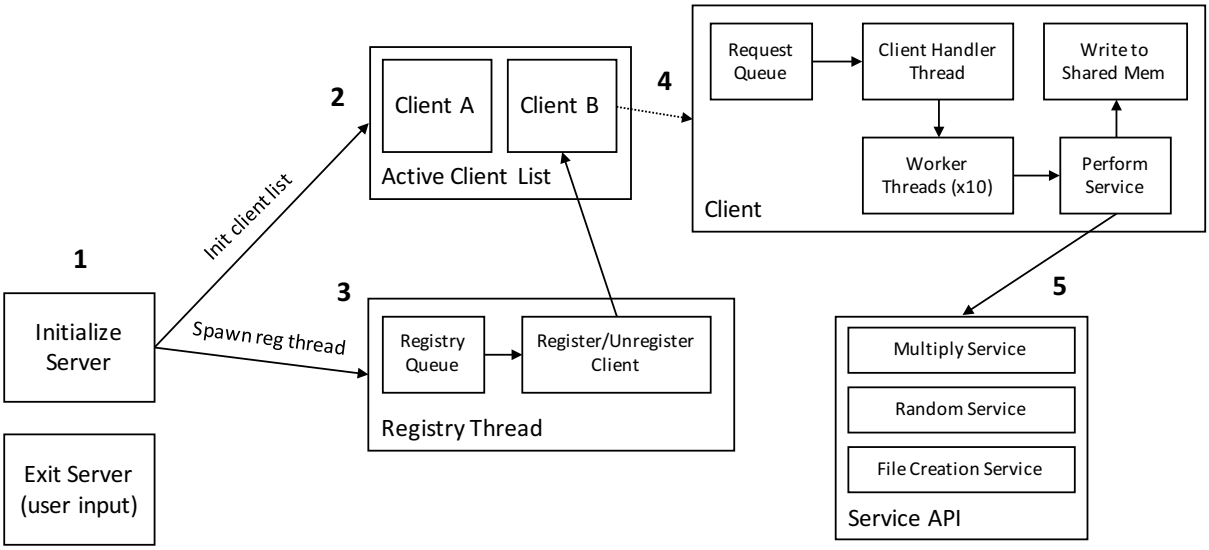

Figure 9: GTIPC server process design and operation

The server first performs initialization, which roughly involves the following steps:

1. Create the global registry POSIX message queue.
2. Spawn the registry handler background thread.
3. Wait for user to exit.

Upon server exit, the inverse process is performed to cleanup resources.

Client Registration

The registry handler thread waits for new messages to arrive on the registry queue and either registers or unregisters the client.

When a new client requests to be registered with the server, the following steps are performed on the registry thread:

1. Open send and receive queues based on names provided by client registry message.
2. Open shared memory object and map it.
3. Create a new *client* object that stores all information about the client.
4. Append the client to active client list (global doubly linked list).
5. Initialize the client's handler and worker thread pool (10 threads).

When a client is unregistered, all resources are cleaned up and the client's entry in the client list is removed.

Client Worker Threads

Each client has its own dedicated pool of threads which all listen on the client's send POSIX message queue. The worker threads increment a count of how many requests have been completed and synchronize access using a mutex.

Once the service has been performed (e.g., generating 4 random integers), the relevant worker thread writes the result back to shared memory and set the *done* flag to 1.

Sample Application

We wrote a sample application (*sample/client.c*). that uses the client library to perform a sequence of asynchronous and synchronous API requests to the server process. You can build the sample by running *make client*.

Recall that our library provides three API services:

1. *MUL*: multiplies two 32-bit integers
2. *RAND*: generates 4 pseudorandom 32-bit numbers
3. *FILE*: creates and/or appends to a file

The application performs the following (in exact sequence):

1. Initialize the GTIPC library
2. 1024 asynchronous requests to the *MUL* service (triggers shared memory resize)
3. 128 synchronous (blocking) requests to the *RAND* service
4. Spawns 10 background threads that **each** perform (in parallel):
    a. An asynchronous request to *RAND*
    b. An asynchronous request to the *FILE* service
    c. A synchronous request to *RAND*
    d. Join both of the asynchronous requests
5. 1024 asynchronous requests to the *MUL* service (back on main thread)
6. Join all 2048 asynchronous requests
7. Exit the GTIPC library

Results: Sample Application

We compiled and ran the sample application described above in two different testing environments and recorded the time. All times are **real** (via *CLOCK_REALTIME*) and not CPU times.

Environments:

- Ubuntu (v4.10 kernel), GCC 5.4.0, 2 CPUs
- AdvOS VM (v3.13 kernel), GCC 4.8.4, 3 CPUs

Ubuntu

1. Time to *dispatch* first 1024 async requests: **21299 usecs**
2. Time to *complete* 128 sync and 2048 async requests: **898769 usecs**

AdvOS VM

1. Time to *dispatch* first 1024 async requests: **59563 usecs**
2. Time to *complete* 128 sync and 2048 async requests: **1 sec and 626884 usecs**

Conclusion

In this report, we presented our design and implementation of an IPC client library and server that allows generic applications to perform services on a different process. The project involved using various POSIX IPC mechanisms, namely message queues and shared memory objects.