Component based software design

Linber

RPC/IPC framework

Falzone Giovanni

jointly M.Sc Embedded Computing Systems

Sant'Anna School of Advanced Studies

University of Pisa

September 7, 2019

Contents

1 Introduction

Linber is a kernel driver module that allow the user to register and request a service exchanging the payload between two applications, *Server worker* and *Client*.
The Server application can register a service and define a number of worker threads waiting to serve the incoming requests.
During the **Service registration** it also define the *period* and the *budget* used to set the *Rreal Time* policy when the request have timing constraints; the budget from an application point of view is considered as the number of request that a worker can serve in the same period.

The Client application **request a service** specifying a relative deadline, it can choose between different type of request:

- Blocking request with payload in buffer
- Blocking request with payload in shared memory
- Non blocking request with payload in buffer
- Non blocking request with payload in shared memory

Also the Server worker can choose between both a response in buffer or in shared memory.

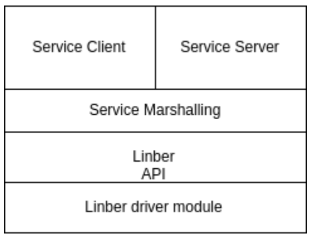

Figure 1: linber layers

2 Implementation

The module is realized as a driver module and uses IOCTL to comunicate with user space, it uses 0xFF as identifier of the ioctl call and a sequence number in the range [0, 10].
The module mantains a list of registered services, each service is also a node in a *RBtree* sorted using the service uri, the service uri is an unique identifier of the service.
The operations that manage the service list are protected using a module mutex to avoid inconsistencies.

Each service mantains an *RBtree* of **waiting requests** and a list of **completed requests**. Every service worker is associated with a request that represents the serving request for that worker. The operations that operate on the lists and the *Rbtree* are protected using a service mutex to avoind inconsistencies.

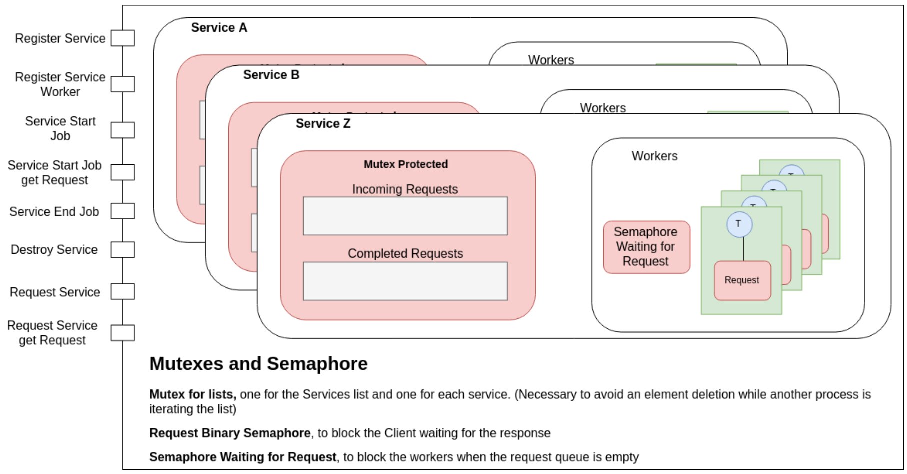

Figure 2: component view

2.1 Request

A request can be in one of the following states:

- **Waiting**
- **Serving**
- **Completed**

A *Waiting* request is stored in the service incoming *RBtree* that is sorted using the absolute deadline of the request. Each node in the *RBtree* can have multiple request organized in a FIFO queue, we can have multiple requests with the same absolute deadline.

Once a request arrives, the workers are notified of this event using a semaphore where they can block when there aren't waiting requests. The *Client* that sent the request is blocked on a binary semaphore associated with the request until the request is completed or aborted. If the *Client* used a non blocking request then can retrieve the response later using an unique *token* and the absolute deadline received with the request.

When a blocked *Worker* is notified of a new request, it pick-up the oldest one among the requests that have the shortest absolute deadline and store it in his serving request slot. Once the *Worker* completed the request then move the request in the *Completed* FIFO queue and notify the *Client* that the request is completed using the request semaphore.

2.1.1 Non Blocking request

In the case of a non blocking request case the *Client* is not blocked in the request semaphore and return the control to the user. Once the user want to retrieve the response it calls the *get response* API function, this will start a procedure to find the request in the service. The request after the non blocking request call can be in one of the three different states:

- **Waiting**, the request is still in the incoming *RBtree*, we search it using the absolute deadline and the unique token, then the *Client* blocks on the request
- **Serving**, one of the workers taken it and is serving it, we search among all the workers, check with the token and block the *Client* in the request
- **Completed**, the request is in the Completed FIFO queue, we search it among all of them and then return the response and the control to the *Client*

2.2 How the data are exchanged

The framework allow the client and the server to exchange the request and the response using shared memory or kernel memory, for each of them there are dedicated API function.

In the case of kernel memory, every message is copied from the user space into the kernel space and linked in the request; let's consider the example of a Client and a Worker exchanging request and response using the kernel memory:

1. Request call from the user, the request is copied into the kernel space
2. Worker wake-up to serve, the request is copied from kernel space into the worker's user space
3. Worker served the request, the response is copied from user space into kernel space
4. Client wake-up, the response is copied into the Client's user space.

In the following fig:3 is reported the sequence call for a blocking request that is using the kernel memory to exchange the payload. Notice that when the Client wake-up it knows the dimension of the response, allocate the buffer into the user space and then invoke another ioctl call to get a copy of the response into the buffer.

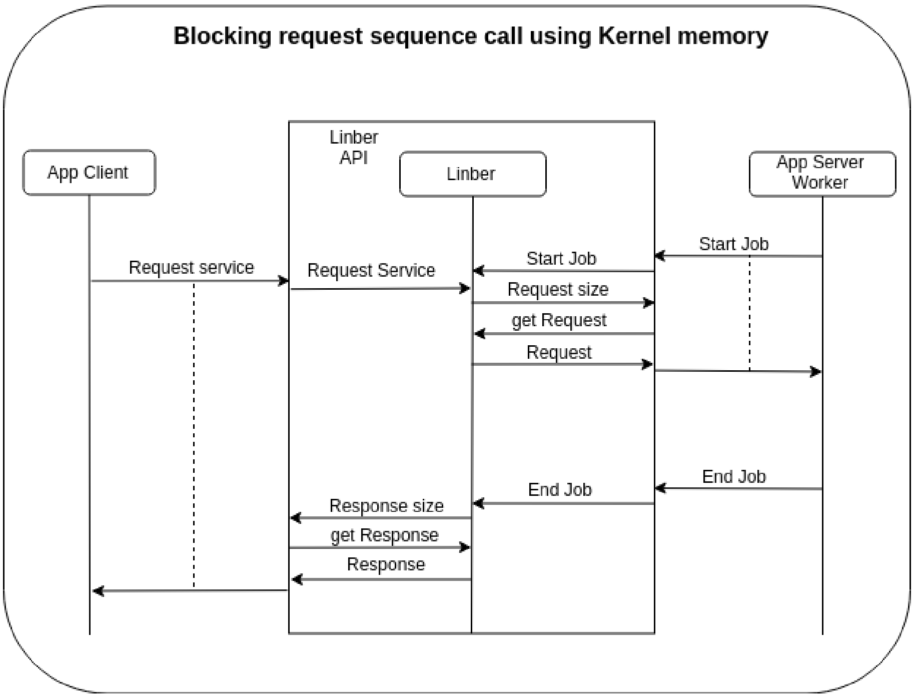

Figure 3: Sequence call of a blocking request

The latter situation where it is needed another iotcl call is not necessary in the case of Shared memory because the applications are exchanging the keys of shared memory and not the payload, but they need to use the syscall to manage the sahred memory.

2.3 Serving policy and Scheduling

When the *Client* requests a service specify a relative deadline, if this is set to zero then the absolute deadline for the request is set to the maximum value $2^{64} - 1$ and will be served only if the *RBtree* of the waiting request is empty or there aren't requests with timing constraints.

Every time a worker pick-up a waiting request, it looks at the absolute deadline and choose one of the following policies.

- **Real time**
- **Best effort**

2.3.1 Best effort

The best effort policy is obtained setting the worker's scheduler to SCHED_FIFO with a priority equal to 99, that is the maximum for a *Real Time* task scheduled under SCHED_FIFO or SCHED_RR.
The worker choose this policy if one of the following conditions are valid:

- the absolute deadline is equal to the maximum value, $2^{64} - 1$
- the deadline is expired, $absolutedeadline > now$

2.3.2 Real Time policy

The *Real time policy* is obtained setting the worker's scheduler to SCHED_DEADLINE with period and budget defined during the service registration.
The relative deadline is computed as $rel\_deadline = (now - absolute\_deadline)$, if the relative deadline is greater then the service period then is set to the service period.

3 Performance evaluation

The module have been tested measuring the elapsed time of a blocking request, the same request have been executed for 1000 times and the elapsed times collected to evaluate the performance of the framework. The request deadline have been set to 0, in order to force the worker to execute with SCHED_FIFO with maximum priority. The server have only one worker and an execution time equal to 0, it only allocate the space needed for the response that is equal to the request' size.

3.1 Environment preconditions

In order to mantain as stable as possible the cpu frequency and to avoid interferece during the test, the following configurations have been applied on the testing machine:

- Disable turbo boost
- Limits the maximum and minimum P-State to the same value without turbo boost
- Set the governor policy to performance for all the cpus
- Set to maximum the budget for RT tasks
- Execute the client with a nice value of -20

The hardware used for the tests

```
Intel(R) Core(TM) i7-8550U CPU running @1.60Ghz
L1d cache: 32K L1i cache: 32K
L2 cache: 256K
L3 cache: 8192K
```

The performance evaluation test have been done starting with a payload of 1 byte and multiplying its size by a factor of 2 until reaching 1 Mbyte when using kernel memory and 1GB when using shared memory.

All of the following figures and more are located in *testtest_ efficiencyplots*, the stored values in two csv files located in *testtest_ efficiencyplots*, there is also a matlab script that generates and save the figures starting from these csv files. Executing the client_eff_mem and client_eff_shm the two csv files will be substituted with the new results, remember to execute the client that use the shared memory with the server that use shared memory because the kernel allocation is limited to 4MB.

In the following figure is reported the average execution for the cases using shared memory and the cases using kernel memory.

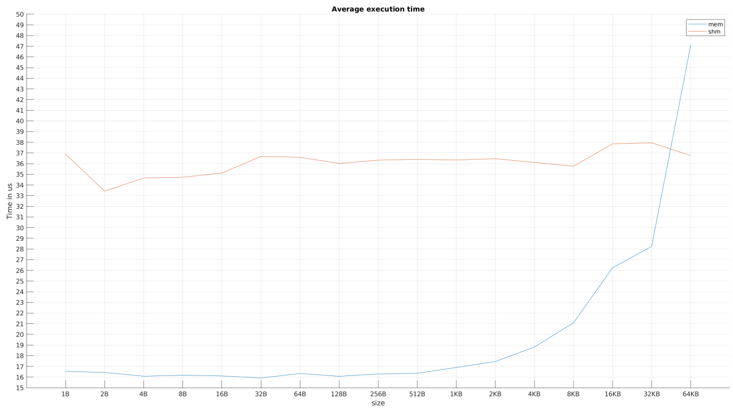

Figure 4: Average Execution time comparison

3.2 Results obtained using kernel memory

In figure 5 is reported for each size the maximum, average and minimum execution time in microseconds obtained over the 1000 iterations; in figure 6 a zoom of the same result.

In figure 7 is reported the probability distribution for each size in a logarithmic scale, this figure allow to see how with a small payload under 4KB the variability of the execution time is smaller and how it become more unpredictable increasing the size. Then in 8 and 9 are reported the cases with a smaller payload size.

Finally the last figure 10 is a plot of the execution time for each size and iteration, it is reported in a logarithmic scale. There is the case of 32KB that has a step where the execution time is reduced by $10us$ for more then 500 iterations.

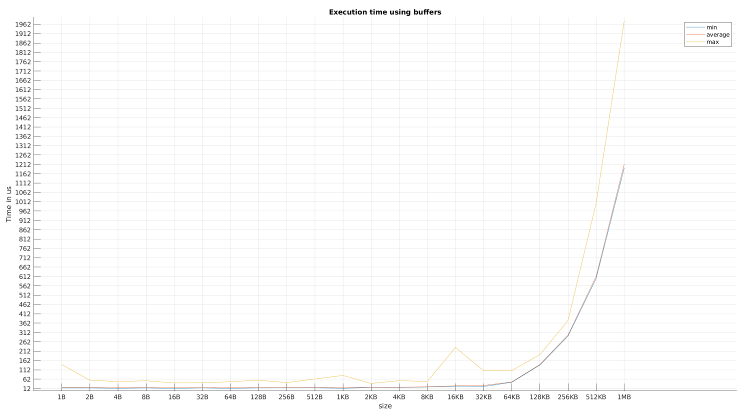

Figure 5: execution time results

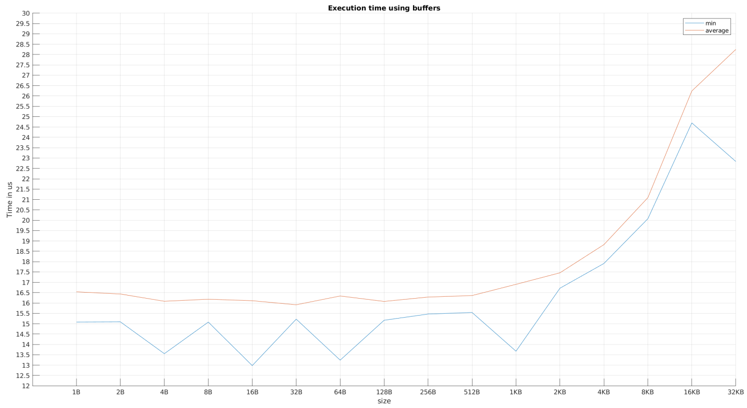

Figure 6: execution time results zoom

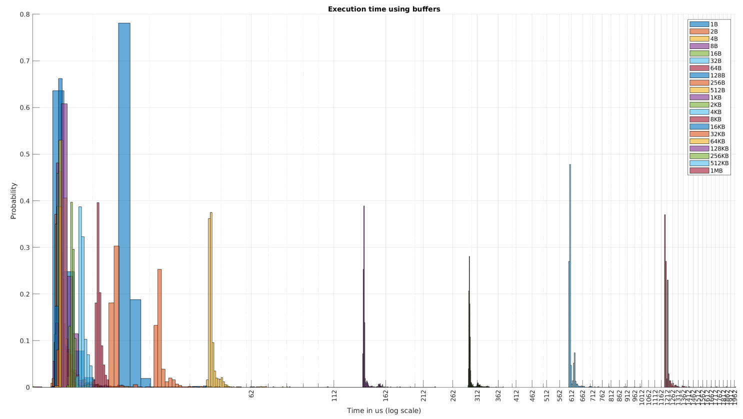

Figure 7: probablity distribution for every size

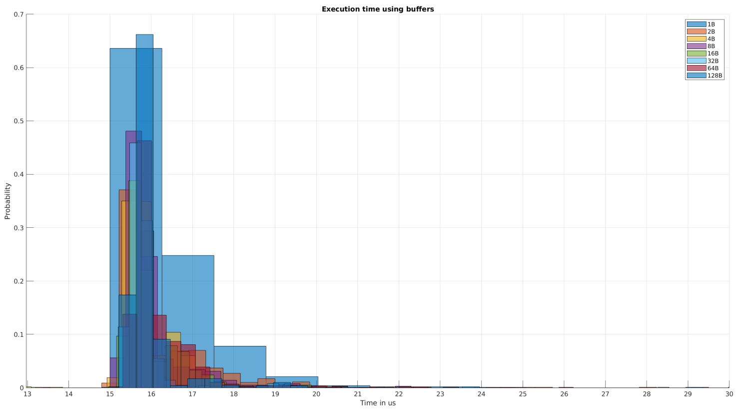

Figure 8: probablity distribution small payload

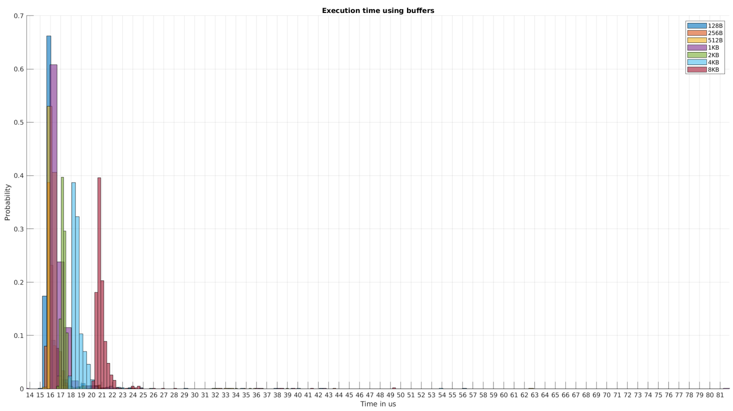

Figure 9: probablity distribution small payload

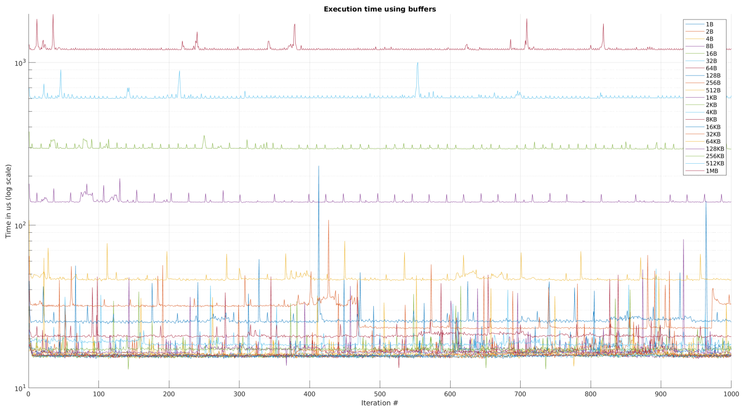

Figure 10: execution time, all cases all iterations

3.3 Results obtained using shared memory to exchange data

The results are flatter respect to the other case because while using memory we have to copy from and into the kernel, here we copy just the shared memory key but we have to pay in overhead due to the syscalls needed to manage the shared memory. In the figure13 are reported the histograms related to all the cases, we can notice that the variability of the results is smaller for all the cases.

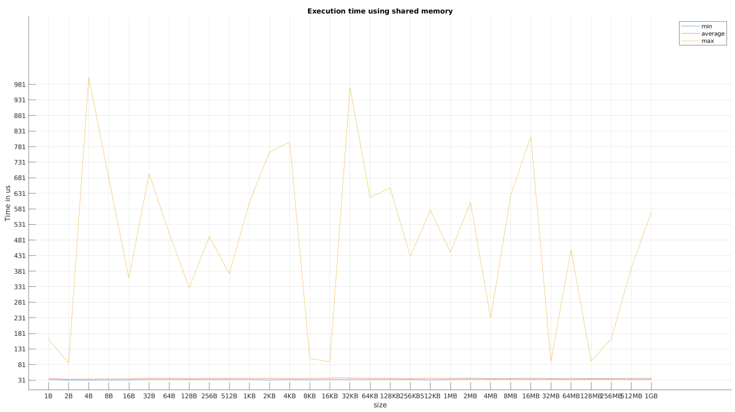

Figure 11: Execution time, minimum average and maximum

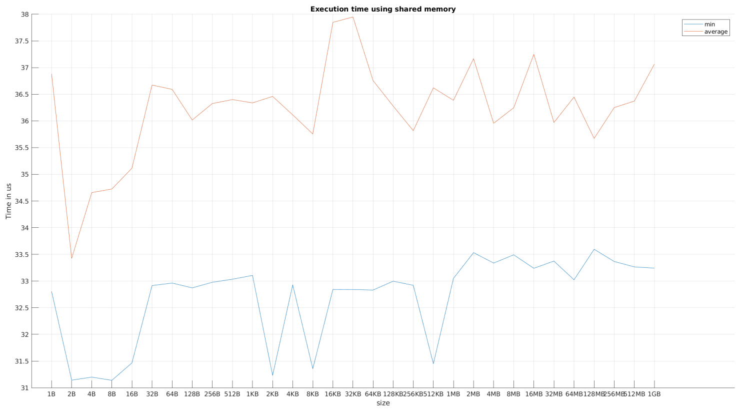

Figure 12: Execution time, minimum and average for all cases

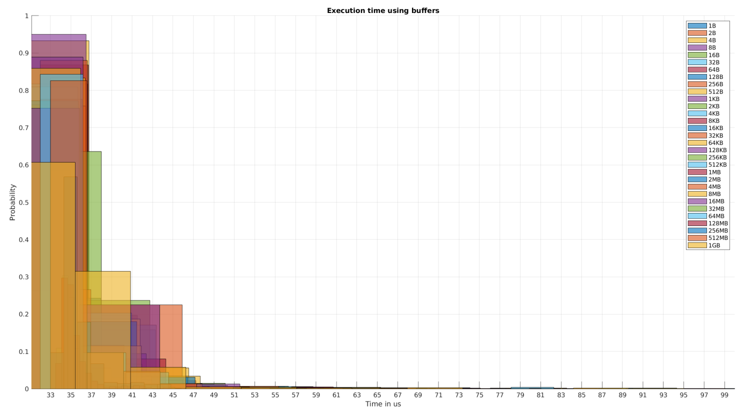

Figure 13: Probability distribution for all cases

A Calculator Worker example

Once registered the service using the *Linber API* the server needs to create the workers, it is possible to extend the *LinberServiceWorker* and implement the virtual funcion *execute_job* that will be called every time arrive a request.

.

```
class linberServiceWorker {
  char *service_uri;
  char *file_str;
  int uri_len;
  unsigned long service_token;
  unsigned int exec_time;
  unsigned int worker_id;
  unsigned int job_num;
  std::thread thread_worker;
  bool worker_alive;

  void worker_job();

protected:
    boolean request_shm_mode;
    char *request;
    int request_len;
    char *response;
    int response_len;

public:
  linberServiceWorker(char * service_uri, unsigned long service_token);
  virtual ~linberServiceWorker();
  void join_worker();
  void terminate_worker();
  virtual void execute_job();
};

class calculator_service : public linberServiceWorker{
  Calculator::Calculator_request request_msg;  // protobuf message
  Calculator::Calculator_response response_msg; // protobuf message

  public:
  calculator_service(char * service_uri, unsigned long service_token);
  float sum(float a, float b);
  float difference(float a, float b);
  float product(float a, float b);
  float division(float a, float b);
  float power(float a, float b);
  float square_root(float a);
  float compute(unsigned int operation, float a, float b);
  void execute_job()override;
};
```