# 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 6, 2019

## Contents

## 1 Introduction

The purpose of this project is to realize a kernel module to allow two user space application, a Server and a Client to exchange a payload representing a request and the corresponding response. The Server application can register a service and define a number of worker threads waiting to serve the incoming requests. it also define the *period* and *budget* used to set the RT 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 can send a request for a service specifying a relative deadline for the request and can choose between blocking and non blocking request and also between shared memory and kernel memory.

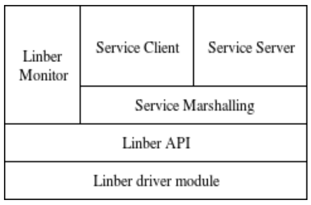
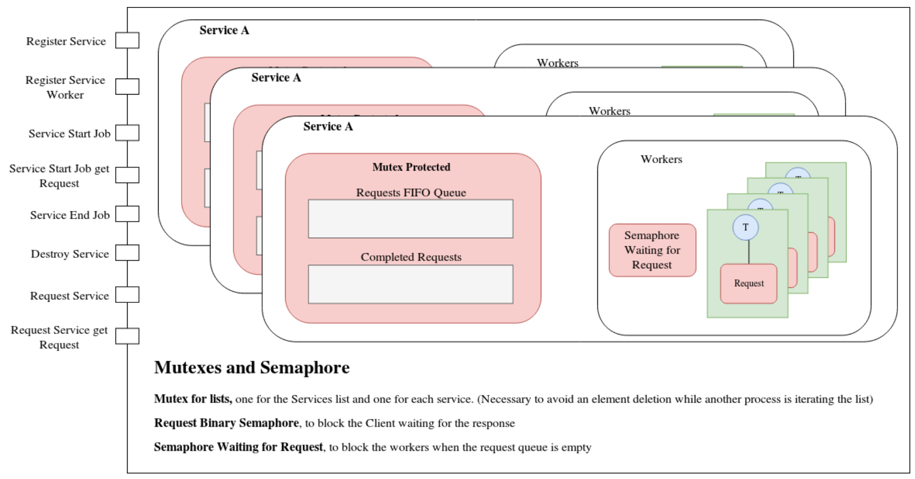


Figure 1: linber layers

# 2 Implementation

The module is realized as a driver module and uses IOCTL to comunicate with user space. It uses 0x20 ...

## 2.1 Module internals



**Mutexes and Semaphore**

**Mutex for lists,** one for the Services list and one for each service. (Necessary to avoid an element deletion while another process is iterating the list)

**Request Binary Semaphore**, to block the Client waiting for the response

**Semaphore Waiting for Request**, to block the workers when the request queue is empty

Figure 2: component view

Each service is mantained in a *RBtree* sorted using the service uri.

## 2.2 Request

A registered 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, this because we can have multiple incomng requests with the same absolute deadline. Once a request arrive the workers are notified of this event using a semaphore where the workers can block waiting to serve a request. 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 dedicated 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.

#### 2.2.1 Non Blocking request

In this 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 lie in all 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*

### 2.3 How the data are exchanged

The framework allow the client and the server to exchange the request and the response using both shared memory and kernel memory. There are two different kind of API function in ordere to use one or the other.

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 send again another ioctl call to get a copy of the response into the buffer.

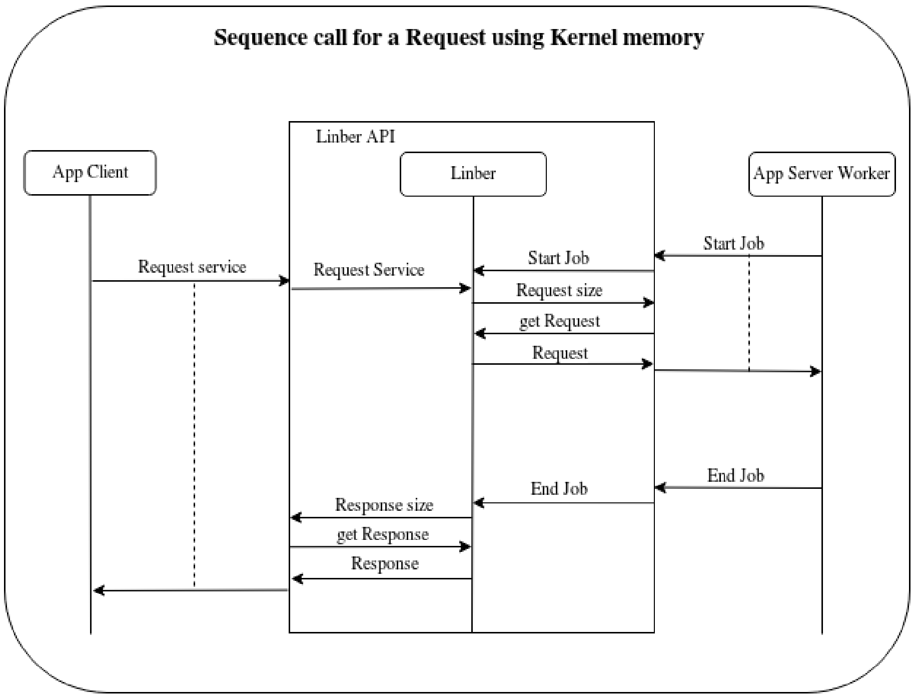


Figure 3: Sequence call of a blocking request

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.

## 2.4 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 incoming *RBtree* has no request with an asbsolute deadline less then the maximum one.

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

- Real time policy with guarantees
- Best effort, Real time policy without guarantees

### 2.4.1 Best effort

The best effort policy is obtained setting the worker's scheduler policy as SCHED_FIFO with a priority equal to 99, that is the maximum for SCHED_FIFO and SCHED_RR. The worker choose this policy if one of the following conditions are valid

- the absolute deadline is equal to the masimum value, $2^{64} - 1$
- the deadline is expired, (absolute deadline $>$ now)

### 2.4.2 RT policy

The best effort policy is obtained setting the worker's scheduler policy as SCHED_DEADLINE with period and budget defined during the serving registration. The deadline is comuted starting from the absolute deadline, if the relative deadline is greater then the period then is set to the service period.

## 3 Performance evaluation

The module have been tested measuring the elapsed time of a blocking request sent to a service worker, the same request have been sent 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 for the response.

### 3.1 Environment preconditions

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

- Set to maximum the budget for RT tasks
- 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
- 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
```

### 3.2 Results obtained using kernel memory to exchange data

The performance evaluation test have been done starting with a payload of 1 byte and multiplying its size until reaching 1 Mbyte.

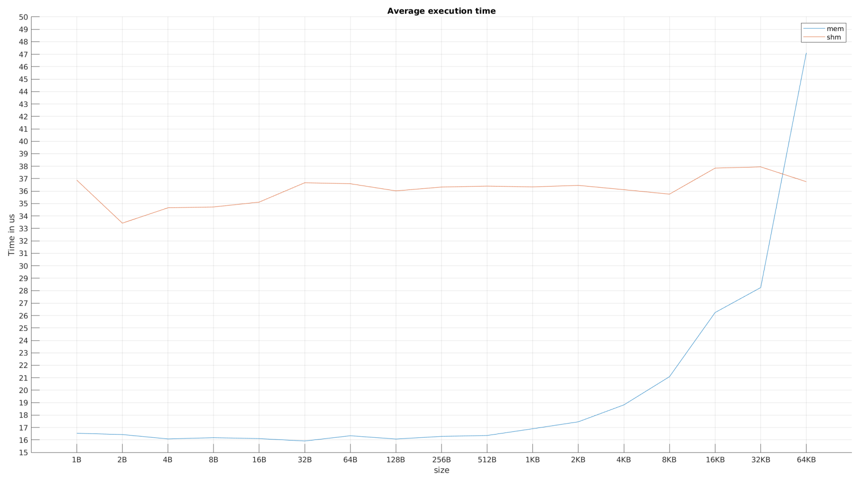


Figure 4: Execution time comparison between shared memory and kernel memory

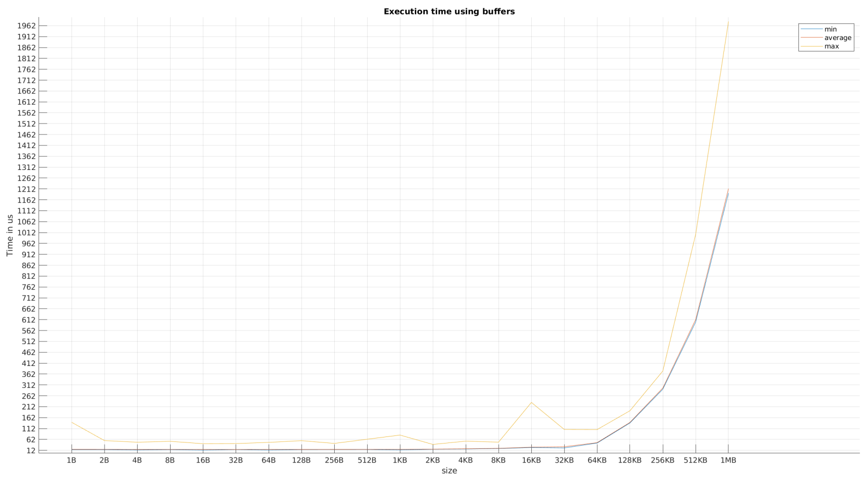


Figure 5: execution time results

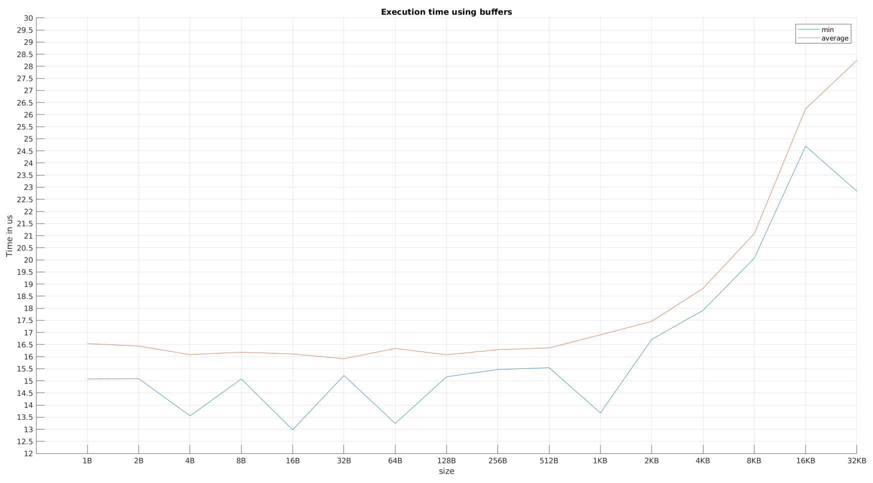


Figure 6: execution time results zoom

In figure 5 is reported for each size the maximum, mean and minimum execution time in microseconds obtained over the 1000 iterations; In figure 13 is reported a zoom of the same result.

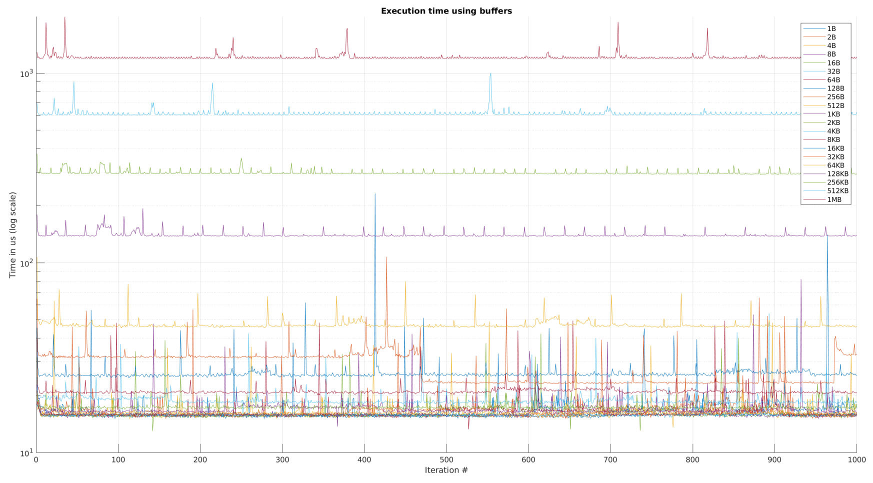


Figure 7: execution time

In figure 14 are reported all the cases in a logarithmic scale.

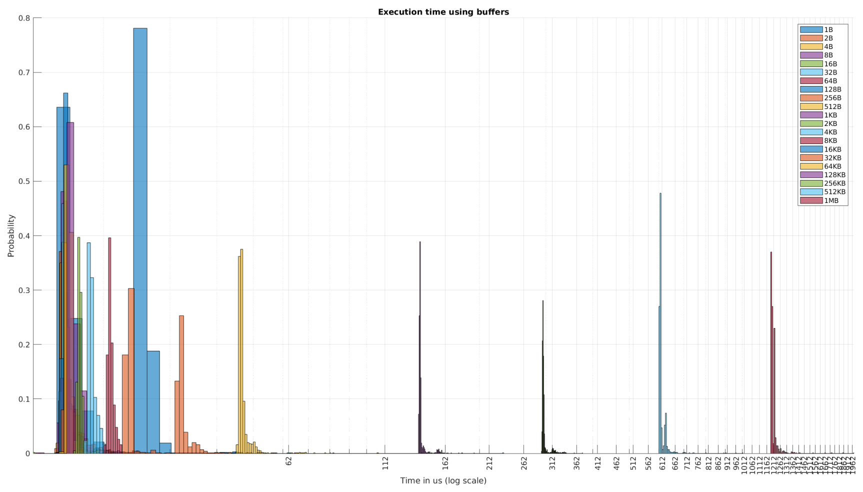


Figure 8: probablity distribution for every size

In figure 12 is reported the probability distribution for each size in a logarithmic scale, this picture allow to see how with a small payload under 4KB the probability is higher and the execution time is more stable.

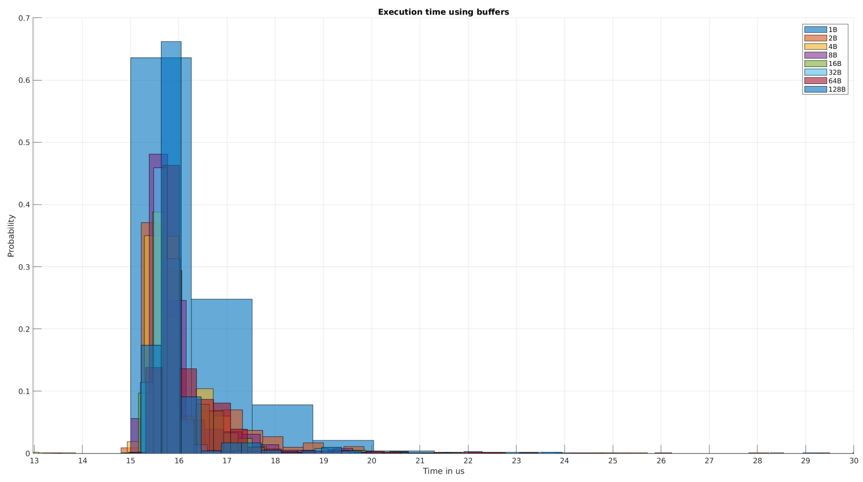


Figure 9: probablity distribution small payload

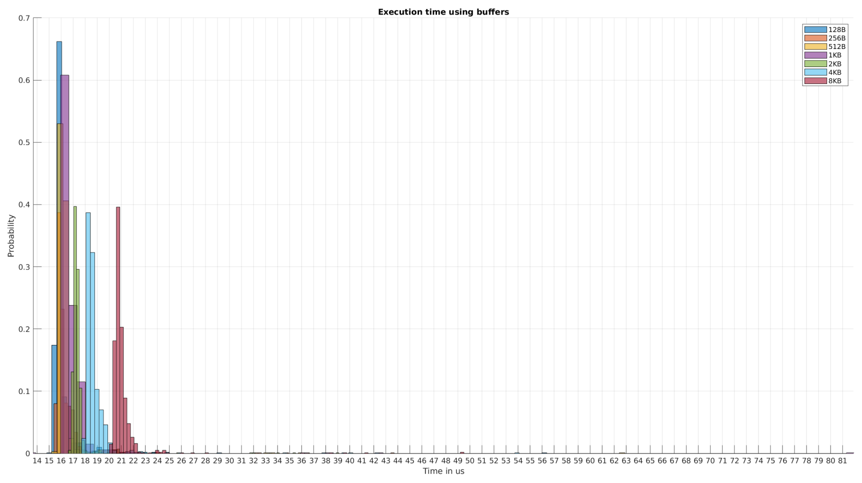


Figure 10: probablity distribution small payload

## 3.3 Results obtained using shared memory to exchange data

The performance evaluation test have been done starting with a payload of 1 byte and multiplying its size until reaching 1 Gbyte. The results are flatter respect other case because while using memory we have to copy from and into the kernel, here we share just the shared memory key. Notice that with a small payload we have more overhead with shared memory due to the syscall needed to manage it.

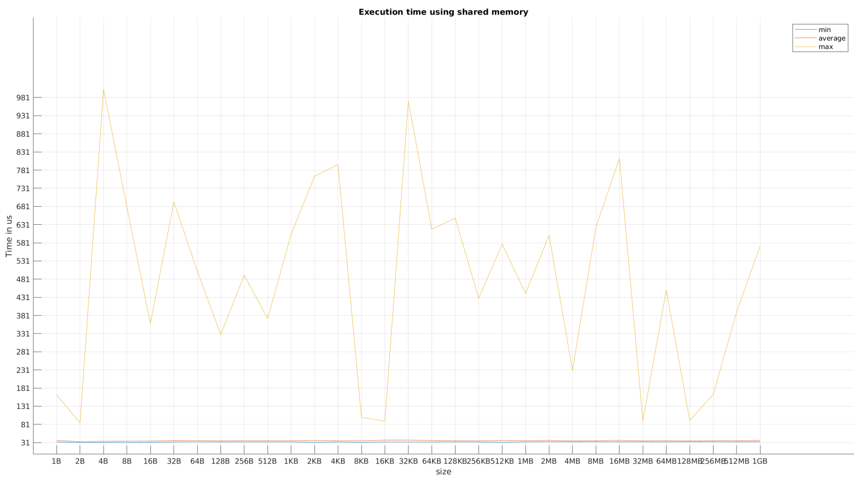


Figure 11: Execution time minimum average and maximum for all cases

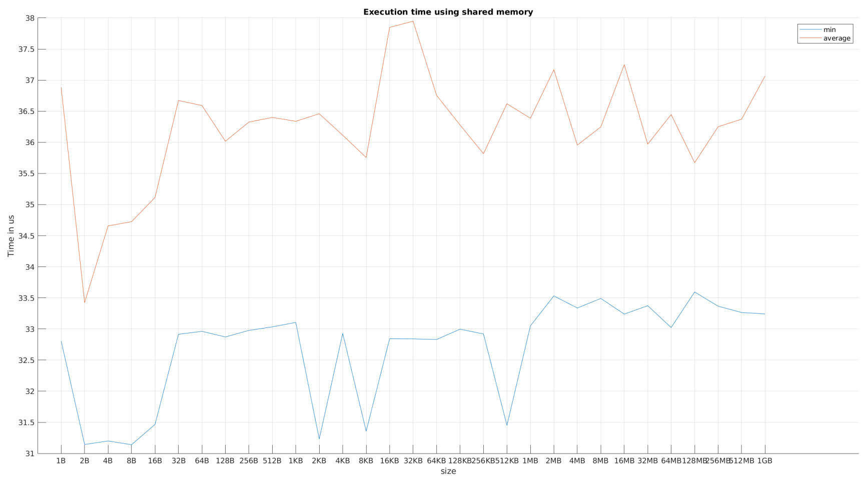


Figure 12: Execution time minimum and average for all cases

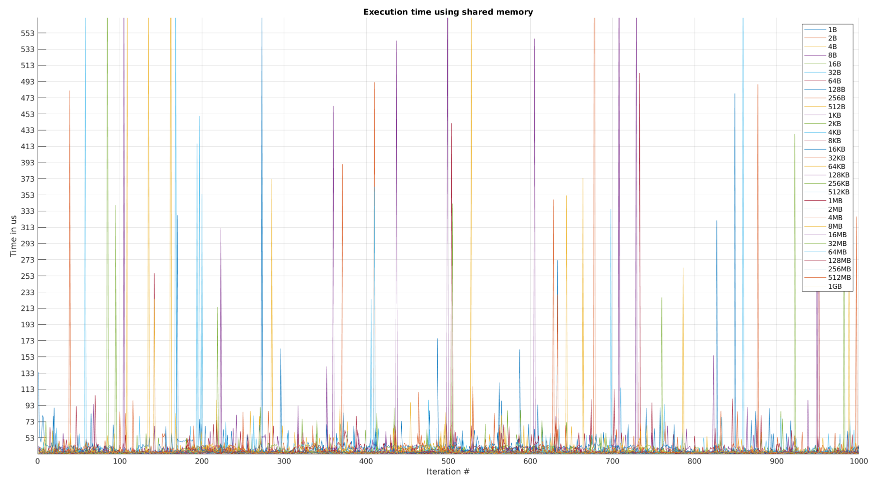


Figure 13: All test values

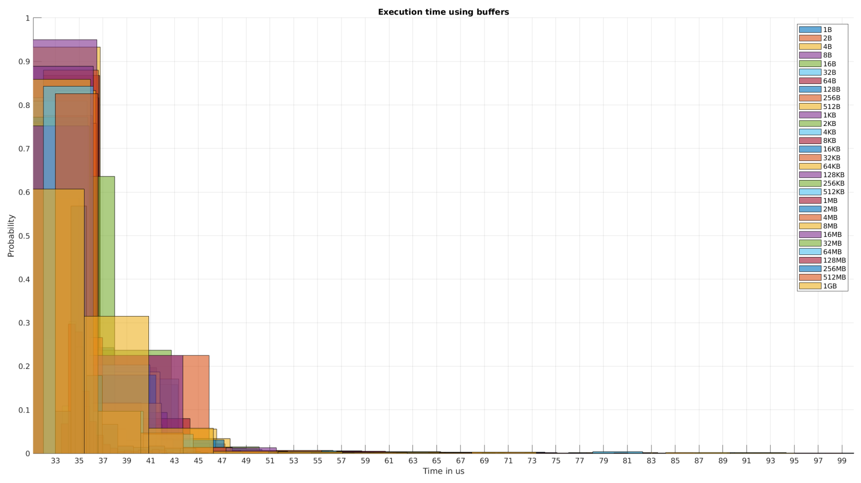


Figure 14: Probability distribution for all cases

## A Linber Worker UML

## B Calculator Worker example

```
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 defined
  Calculator::Calculator_response response_msg; // protobuf defined

  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;
};
```