# Extra clarification for IPC APIs

The client processes have two options how to call the remote service: synchronous or asynchronous.

### 1. Sync API

For instance, the client thread can call something like:

```
call_service(input_args_p, &result_p…)
```

At this point, the client thread is blocked, until the service returns. The result or status will be available at `result_p`.

In the meantime, what will happen in the `call_service` procedure:

- a request record (if appropriate) or at least the arguments will be placed in the shared memory channel, any sync mechanisms will be triggered (e.g., there may be shared condition variables or mutexes in the shared memory segment shared among this client thread and a thread in the server process, or other options are available, via shared memory message queues or semaphores)
- the library code will implement the waiting step for the results, and when the result is available, will make sure it's either stored in the `call_service` provided pointer, or the proper pointer is updated.

You have flexibility to define the semantics of the API, but make sure it is then used in a meaningful way. Particularly, think who will be responsible for making sure the client's result buffer persists, i.e., is not overwritten, by some future requests, how will you make sure you don't have any memory leaks and how you clean up or recycle buffers, etc.

### 2. Async API

There are two basic options for implementation:

#### ▸ Poll

The client thread calls something like:

```
request_handle = initiate_service(input_args_p, &result_p…)
```

The client side library code again places a request record/arguments in shared memory, does necessary signaling/condition variables etc., so the server can (eventually) see that there is a request pending, but the client thread doesn't wait for the results. It returns immediately afterwards (it would have to wait to make

sure there is some shared memory available to pass the arguments, but not to wait for the result). This operation should return some `request_handle` that can be later used to retrieve the result.

The client thread can now continue doing some other processing. At a later time, when the client thread has nothing else to do, it can call something like:

```
wait_for_results(request_handle)
```

This would be a blocking call. The client-side library code would at that point have to make sure that the server has already finished processing and made the result available (e.g., via the `result_p` buffer, which may have been specified when the service request was initialized in the first place).
When the client-side library code realizes that the result is available, the `wait_for_result` call returns, and the client thread is free to continue.

▸ **Push**

The client thread calls again something like `initiate_service()` but one of the call's arguments is a pointer for a `request_handler`, a function. Like above, the client side library code would need to make sure a request record (arguments, descriptor for the result memory, or anything else that might be necessary) is placed in the shared memory channel, and afterwards the client thread can proceed.

One thing that's different, is that at some point, the receipt of a response for this particular request should result in the execution of the handler code.
One possible way would be, by having a separate library thread keeps checking if any result returns, and gets which specific request is that result for, and then calls the `request_handler` function with the result as an argument.
Make sure that the handler executes some meaningful operation given the semantics of your implementation. For instance, if results are stored in temporary buffers that are dynamically allocated and could be reclaimed/reused, then the handler should copy the result from that library buffer to some application buffer.
As you can see, with this scenario, the calling thread in the client application doesn't have to explicitly poll to pick up the result. Instead, when the result is ready, the client side library pushes the result into the application (via the handler).

There are multiple ways to implement sync and async APIs. Please chose the idea that makes sense, and then provide the solution.