In-Storage Compute (Function Shipping) Service User Guide

Preparing a library

APIs from external library cannot be linked directly with a mero instance. A library is supposed to have a function named `mero_lib_init()`. This function will then link the relevant APIs with Mero. Every function to be linked with mero shall confine to the following signature:

```
int comp(struct m0_buf *args, struct m0_buf *out,
         struct m0_isc_comp_private *comp_data, int *rc)
```

All relevant library APIs shall be prepared with a wrapper confining to this signature. Let `libarray` be the library we intend to link with Mero, with following APIs: `arr_max()`, `arr_min()`, `arr_histo()`.

Registering APIs

`mero_lib_init()` links all the APIs. Here is an example code (please see `iscservice/isc.h` for more details):

```
void mero_lib_init(void)
{
      rc = m0_isc_comp_register(arr_max, "max",
                                string_to_fid("arr_max"));
      if (rc != 0)
            error_handle(rc);
      rc = m0_isc_comp_register(arr_min, "min",
                                string_to_fid("arr_min"));
      if (rc != 0)
            error_handle(rc);
      rc = m0_isc_comp_register(arr_histo, "arr_histo",
                                string_to_fid("arr_histo"));
      if (rc != 0)
            error_handle(rc);
}
```

Registering the library

Let `libpath` be the path the library is located at. The program needs to load the same at each of the Mero node. This needs to be done using:

```
int m0_spiel_process_lib_load(struct m0_spiel *spiel,
                              struct m0_fid *proc_fid,
                              char *libpath)
```

This will ensure that `mero_lib_init()` is called to register the relevant APIs.

Invoking an API

Mero has its own RPC mechanism to invoke a remote operation. In order to conduct a computation on data stored with Mero it's necessary to share the computation's `fid` (a unique identifier associated with it during its registration) and relevant input arguments. Mero uses `fop/fom` framework to execute an RPC. A `fop` represents a request to invoke a remote operation and it shall be populated with relevant parameters by a client. A request is executed by a server using a `fom`. The fop for ISC service is self-explanatory. Examples in next subsection shall make it more clear.

```
/** A fop for the ISC service */
struct m0_fop_isc
{



        /** An identifier of the computation registered with the
            Service.
         */
        struct m0_fid        fi_comp_id;
        /**
         * An array holding the relevant arguments for the
         * computation.
         * This might involve gfid, cob fid, and few other
parameters
         * relevant to the required computation.
         */
        struct m0_rpc_at_buf fi_args;
        /**
         * An rpc AT buffer requesting the output of computation.
         */
        struct m0_rpc_at_buf fi_ret;
        /** A cookie for fast searching of a computation. */
        struct m0_cookie     fi_comp_cookie;
} M0_XCA_RECORD M0_XCA_DOMAIN(rpc);
```

For sharing arguments with a computation for receiving the result of a computation we use `RPC Adaptive Transmission Buffers (rpc/at.[ch]::m0_rpc_at)`. Examples in the next section shall make the use case clear. For more details please refer to the documentation in `rpc/at.[ch]`

Examples

Hello-World

Consider a simple API that on reception of string "`Hello`" responds with "`World`" along with return code `0`. For any other input it does not respond with any string, but returns an error code of `-EINVAL`. Client needs to send `m0_isc_fop` populated with "`Hello`". First we will see how client or caller needs to initialise certain structures and send them across.

Subsequently we will see what needs to be done at the server side. Following code snippet illustrates how we can initialize `m0_isc_fop`.

```
/**
 * prerequisite: in_string is null terminated.
 * isc_fop  : A fop to be populated.
 * in_args  : Input to be shared with ISC service.
 * in_string: Input string.
 * conn     : An rpc-connection to ISC service. Should be
established
 *            beforehand.
 */
int isc_fop_init(struct m0_fop_isc *isc_fop, struct m0_buf *in_args,
                 char *in_string, struct m0_rpc_conn *conn)
{
      int rc;

      /* A string is mapped to a mero buffer. */
      m0_buf_init(in_args, in_string, strlen(in_string));
      /* Initialise RPC adaptive transmission data structure. */
      m0_rpc_at_init(&isc_fop->fi_args);
      /* Add mero buffer to m0_rpc_at */
      rc = m0_rpc_at_add(&isc_fop->fi_args, in_args, conn);
      if (rc != 0)
            return rc;
      /* Initialise the return buffer. */
      m0_rpc_at_init(&isc_fop->fi_ret);
      rc = m0_rpc_at_recv(&isc_fop->fi_ret, conn, REPLY_SIZE,
false);
      if (rc != 0)
            return rc;
      return 0;
}
```

Let’s see how this fop is sent across to execute the required computation.

```
#include “iscservice/isc.h”
#include “fop/fop.h”
#include “rpc/rpclib.h”

int isc_fop_send_sync(struct m0_isc_fop *isc_fop,
                      struct m0_rpc_session *session)
{
      struct m0_fop         fop;
      struct m0_fop         reply_fop;
      /* Holds the reply from a computation. */
      struct m0_fop_isc_rep reply;
      struct m0_buf        *recv_buf;
      struct m0_buf        *send_buf;
      int                   rc;

      M0_SET0(&fop);

      m0_fop_init(&fop, &m0_fop_isc_fopt, isc_fop, m0_fop_release);
      /*
       * A blocking call that comes out only when reply or error in
```

```
         * sending is received.
        */
        rc = m0_rpc_post_sync(&fop, session, NULL,
M0_TIME_IMMEDIATELY);
        if (rc != 0)
                return error_handle();
        /* Capture the reply from computation. */
        reply_fop = m0_rpc_item_to_fop(fop.f_item.ti_reply);
        reply = *(struct m0_fop_isc_rep *)m0_fop_data(reply_fop);

        /* Handle an error received during run-time. */
        if (reply.fir_rc != 0)
                return error_handle();
        /* Obtain the result of computation. */
        rc = m0_rpc_at_rep_get(isc_fop->fi_ret, reply.fir_ret,
recv_buf);
        if (rc != 0) {
                comp_error_handle(rc, recv_buf);
        }
        if (!strcmp(fetch_reply(recv_buf), "World")) {
                comp_error_handle(rc, recv_buf);
        } else {
                /* Process the reply. */
                reply_handle(recv_buf);
                /* Finalize relevant structure. */
                m0_rpc_at_fini(&isc_fop->fi_args);
                m0_rpc_at_fini(&reply.fir_ret);
        }
        return 0
}
```

We now discuss the callee side code. Let's assume that the function is registered as `"greetings"` with the service.

```
void mero_lib_init(void)
{
        rc = m0_isc_comp_register(greetings, "hello-world",
                                  string_to_fid("greetings"));
        if (rc != 0)
                error_handle(rc);
}
int greetings(struct m0_buf *in, struct m0_buf
*out,


            struct m0_isc_comp_private *comp_data, int *rc)
{
        char *out_str;

        if (m0_buf_streq(in, "Hello")) {
                /*
                 * The string allocated here should not be freed by
                 * computation and Mero takes care of freeing it.
                 */
                out_str = m0_strdup("World");
```

```
            if (out_str != NULL) {
                  m0_buf_init(out, out_str, strlen(out_str));
                  rc = 0;
            } else
                  *rc = -ENOMEM;
      } else
            *rc = -EINVAL;
        /*
         * A computation returns two integers, one via returned
         * value of the computation and other by setting up an error
         * code in rc. The following table summarises use-cases:
         * return value | error code (rc)   | inference by Mero
         *              |      |
         * M0_FSO_AGAIN |  rc != -EAGAIN | computation is complete.
         *              |      |
         * M0_FSO_AGAIN |  rc == -EAGAIN | computation needs to be
         *              |      | re-triggered without any
         *              |      | wait.
         *              |      |
         * M0_FSO_WAIT  | rc == -EAGAIN   | computation be
         *              |      | re-triggered when the
         *              |      | caller fom stored in
         *              |      | comp_data is signalled.
         */
              .
      return M0_FSO_AGAIN;
}
```

Min/Max

Hello-World example sends across a string. In real applications the input can be a composition of multiple data types. It's necessary to serialise a composite data type into a buffer. Mero provides a mechanism to do so using `xcode/xcode.[ch]`. Any other serialization mechanism that's suitable and tested can also be used eg. [Google's Protocol buffers](). But we have not tested any such external library for serialization and hence in this document would use Mero's `xcode` APIs.

In this example we will see how to send a composite data type to a registered function. A declaration of an object that needs to be serialised shall be *tagged* with one of the types identified by `xcode`. Every member of this structure shall also be representable using `xcode` type. Please refer `xcode/ut/` for different examples.

Suppose we have a collection of arrays of integers, each stored as a Mero object. Our aim is to find out the min or max of the values stored across all arrays. The caller communicates the list of global fids(unique identification of stored object in Mero) with the registered computation for min/max. The computation then returns the min or max of locally (on relevant node) stored values. The caller then takes min or max of all the received values. The following structure can be used to communicate with registered computation.

```
/* Arguments for getting min/max. */
struct arr_fids {
      /* Number of arrays stored with Mero. */
      uint32_t       af_arr_nr;
```

```
    /* An array holding unique identifiers of arrays. */
    struct m0_fid *af_gfids
} M0_XCA_SEQUENCE;
```

Before sending the list of fids to identify the min/max it's necessary to serialise it into a buffer, because it's a requirement of ISC that all the computations take input in the form of a buffer. Following snippet illustrates the same.

```
int arr_to_buff (struct arr_fids *in_array, struct m0_buf *out_buf)
{
      int rc;

      rc = m0_xcode_obj_enc_to_buf(XCODE_OBJ(arr_fids),
                                   &out_buf->b_addr,
                                   &out_buf->b_nob);
      if (rc != 0)
            error_handle(rc);
      return rc;
}
```

The output buffer `out_buf` can now be used with RPC AT mechanism introduced in previous subsection. On the receiver side a computation can deserialize the buffer to convert into original structure. The following snippet demonstrates the same.

```
int buff_to_arr(struct m0_buf *in_buf, struct arr_fids *out_arr)
{
      int rc;

      rc = m0_xcode_obj_dec_from_buf(XCODE_OBJ(arr_fids),
                                     &in_buf->b_addr,
                                     in_buf->b_nob);
      if (rc != 0)
            error_handle(rc);
      return rc;
}
```

Preparation and handling of a `fop` is similar to that in `Hello-World` example. Once a computation is invoked, it will read each object's locally stored values, and find min/max of the same, eventually finding out min/max across all arrays stored locally. In the next example we shall see how a computation involving an IO can be designed.

Histogram

We now explore a complex example where a computation involves an IO, and hence needs to wait for the completion of IO. User stores an object with Mero. This object holds a sequence of values. The size of an object in terms of the number of values held is known. The aim is to generate a histogram of values stored. This is accomplished in two steps. In the first step user invokes a computation with remote Mero servers and each server generates a histogram of values stored with it. In the second step, these histograms are communicated with the user and it adds them cumulatively to generate the final histogram. Fig. 1.0 illustrates the overall flow of operations. The following structure describes a list of arguments that will be communicated by a caller with the ISC service for generating a histogram. ISC is associated only with the first part.

```
/* Input for histogram generation. */
struct histo_args {
      /** Number of bins for histogram. */
      uint32_t      ha_bins_nr;
      /** Maximum value. */
      uint64_t      ha_max_val;
      /** Minimum value. */
      uint64_t      ha_min_val;
      /** Global fid of object stored with Mero. */
      struct m0_fid ha_gob_fid;
} M0_XCA_RECORD;
```

The array of values stored with Mero will be identified using a global id represented here as `ha_gob_fid`. It has been assumed that maximum and minimum values over the array are known or are made available by previous calls to `arr_max()` and `arr_min()`.

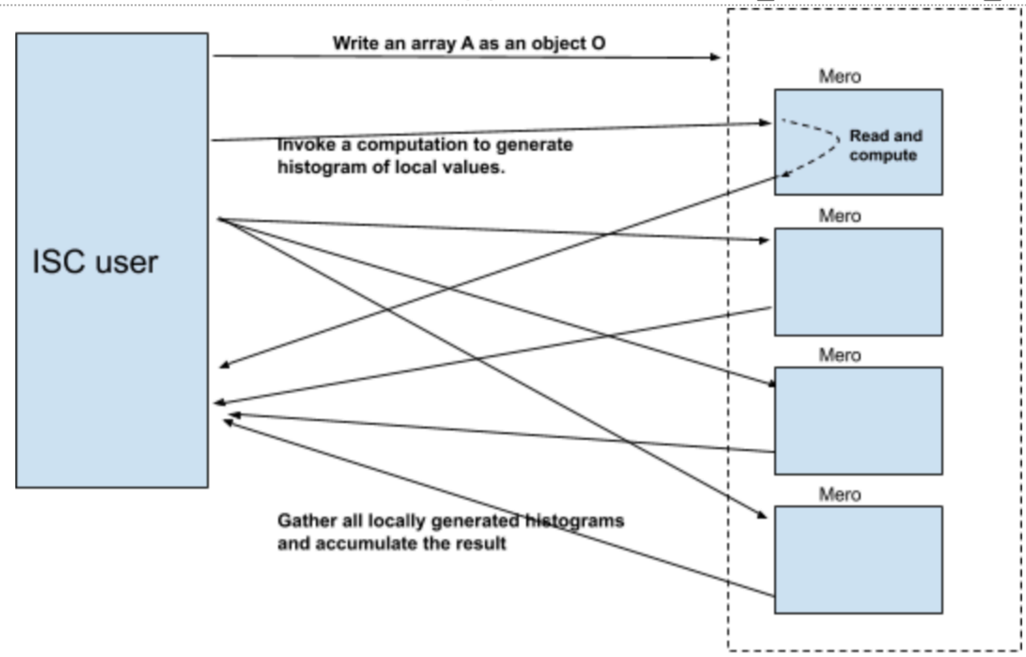

Fig. 1.0: ISC user first writes an array in the form of a Mero object. It is assumed that the relevant computation is already registered with each Mero instance. On invocation of the computation each instance of computation reads the local part of stored array, and generates a histogram of it. This histogram is communicated with ISC user which then accumulates the result.

Here we discuss the API for generating a histogram of values, local to a node. The caller side or client side shall be populating the `struct histo_args` and sending it across using `m0_isc_fop`.

```
/*
 * Structure of a computation is advisable to be similar to
 * Mero foms. It returns M0_FSO_WAIT when it has to wait for
 * an external event (n/w or disk I/O)else it returns
```

```
 * M0_FSO_AGAIN. These two symbols are defined in Mero.
 */
int histo_generate(struct m0_buf *in, struct m0_buf *out,
                   struct m0_isc_comp_private *comp_data,
                   int *ret)
{
      struct *histo_args;
      struct *histogram;
      Struct *hist_partial;
      uint32_t disk_id;
      uint32_t nxt_disk;
      int      rc;
      int      phase;

      phase = comp_phase_get(comp_data);
      switch(phase) {
      case COMP_INIT:
            /*
             * Deserializes input buffer into "struct histo_args"
             * and stores the same in comp_data.
             */
            histo_args_fetch(in, out, comp_data);
            rc = args_sanity_check(comp_data);
            if (rc != 0) {
                  private_data_cleanup(comp_data);
                  *ret = rc;
                  return M0_FSO_AGAIN;
            }
            comp_phase_set(comp_data, COMP_IO);
      case COMP_IO:
            disk  = disk_id_fetch(comp_data);
            /**
              This will make the fom (comp_data->icp_fom) wait
              on a Mero channel which will be signalled on
      completion
              of the IO event.
             */
            rc = m0_ios_read_launch(gfid, disk, buf, offset, len,
                                    comp_data->icp_fom);
            if (rc != 0) {
                  private_data_cleanup(comp_data);
                  /* Computation is complete, with an error. */
                  *ret = rc;
                  return M0_FSO_AGAIN;
            }

            comp_phase_set(comp_data, COMP_EXEC);
            /**
              This is necessary for Mero instance to decide whether
              to retry.
              **/
            *ret = -EAGAIN;
            return M0_FSO_WAIT;
      case COMP_EXEC:
            hist_args = hist_args_fetch(comp_data);
```

```
            M0_ALLOC_PTR(hist_partial)
            histo_generate(buf, histo_args->ha_max,
histo_args->ha_mix,
                            hist_args->ha_bins_nr, hist_partial);
            histogram = private_data_to_histo(comp_data);
            hist_accumulate(histogram, hist_partial);
            m0_free(hist_partial);
            /* Iterates over configuration to fetch next disk. */
            nxt_disk = next_disk_id_get(reqh, disk_id);
            if (nxt_disk == disk_id) // No new disk. {
                  histo_to_buffer(histogram, out);
                  private_data_cleanup(comp_data);
                  /* Computation is over, set error code. */
                  *ret = 0;
                  return M0_FSO_AGAIN;
            } else {
                  /**
                    * Proceed to do IO from the next disk.
                    */
                  next_disk_set(comp_data, nxt_disk);
                  comp_phase_set(comp_data, COMP_IO);
                  /* Re-trigger the computation immediately. */
                  *ret = -EAGAIN;
                  return M0_FSO_AGAIN;
           }
       }
}
```