FOP, FOM Programming Guide

Authors: Mandar Sawant <mandar_sawant@xyratex.com>,
Rajesh Bhalerao <rajesh_bhalerao@xyratex.com>
Version: Draft

Introduction

FOP stands for **F**ile **O**peration **P**acket. In a network or a distributed file-system, a file operation on a client may result in sending a message to a server to carry out that operation. FOP is a generic mechanism to define a application specific protocol within Colibri file-system. The application here means any sub-system (within the file-system) above networking layer. FOP is not necessarily restricted to the file operations - unlike the name suggests. Thus it provides more generic and flexible framework to develop a protocol. In this aspect, FOP is different from traditional network file-system protocols messages.

Getting Started

FDL (FOP declaration language)

A FOP structure can comprise of following user defined and native types:
User defined:-
record (a structure in c programming language)
union
sequence (an array in programing languages)

Native types:-
u32
u64
u8
void

A FOP can be declared as follows

- Consider following example

```
record {
        u64 f_seq;
        u64 f_oid
} m0_fop_file_fid;

sequence {
        u32 f_count;
        u8  f_buf
} m0_fop_io_buf;
```

FDL limitations

Following limitations need to be considered while declaring FOP:-

- Currently we cannot define FOPs dynamically, they need to be defined and built before they can be used.

Compiling FDL

A **.ff* file should be compiled using *ff2c* compiler.
After writing FOPs and creating .ff file for a particular module, we need to make an entry for the same in the module's Makefile.am file. This would automatically invoke ff2c on the .ff files and create corresponding "C" format structures.

- Consider following example

Simple FOP, containing native data types in file fom_io_xc.ff :

```
record {
        u64 f_seq;
        u64 f_oid
} reqh_ut_fom_fop_fid;
```

Makefile.am entry:

```
UT_SRCDIR = @SRCDIR@/reqh/ut

noinst_LTLIBRARIES              = libreqh-ut.la

INCLUDES                        = -I. -I$(top_srcdir)/include \
                                  -I$(top_srcdir)
FOM_FOPS                        = fom_io_xc.h fom_io_xc.c

$(FOM_FOPS): fom_io_xc.ff \
        $(top_builddir)/xcode/ff2c/ff2c
        $(top_builddir)/xcode/ff2c $<

libreqh_ut_la_SOURCES           = $(FOM_FOPS) \
                                  reqh_fom_ut.c fom_io_xc.ff

EXTRA_DIST                      = fom_io.ff

MOSTLYCLEANFILES                = $(FOM_FOPS)
```

On compiling fom_io_xc.ff file using ff2c compiler, it creates corresponding fom_io_xc.h and fom_io_xc.c.

Encoding-Decoding FOP

Rather than sending individual items between Colibri services as separate RPCs, as in traditional RPC mechanisms, multiple items are batched and sent as a single RPC. Batching allows larger messages to be sent, allowing the cost of message passing to be amortized among the multiple items. The upper layers of the RPC module, specifically, the Formation module, select which items are to be batched into a single RPC. Once items are selected, the RPC Formation module then creates that single in-core RPC object. This object is then encoded/serialized into an onwire rpc object and copied into a network buffer using the exported interfaces (m0_rpc_encode () and m0_rpc_decode()).
Each onwire rpc object includes a header with common information, followed by a sequence of items. This RPC is sent to the receiver stored and decoded into individual items. The items are queued on the appropriate queues for processing.

Sending a FOP

A fop can be sent as a request FOP or a reply FOP. A fop is sent across using the various rpc interfaces.
Every fop has an rpc item embedded into it.

```
struct m0_fop {
…
      /**
         RPC item for this FOP
       */
      struct m0_rpc_item       f_item;
…
```

Sending a fop involves initializing various fop and rpc item structures and then invoking the m0_rpc_post routines. The steps for the same are described below with few code examples.

(a) Declare a fop

Declare a fop in a .ff file as given below :

```
record {
        u64 rce_unused
} m0_rpc_fop_conn_establish;
```

(b) Define and initialize the fop_type ops

```
const struct m0_fop_type_ops m0_rpc_fop_conn_establish_ops = {
      .fto_fom_init = &m0_rpc_fop_conn_establish_fom_init,
};
```

(c) Define and initialize the rpc item_type ops

```
static struct m0_rpc_item_type_ops default_item_type_ops = {
      .rito_encode       = m0_rpc_fop_item_type_default_encode,
      .rito_decode       = m0_rpc_fop_item_type_default_decode,
      .rito_payload_size = m0_fop_item_type_default_onwire_size,
};
```

(d) Define and initialize the rpc item type

```
m0_RPC_ITEM_TYPE_DEF(m0_rpc_item_conn_establish,
                 m0_RPC_FOP_CONN_ESTABLISH_OPCODE,
                 m0_RPC_ITEM_TYPE_REQUEST | m0_RPC_ITEM_TYPE_MUTABO,
                 &default_item_type_ops);
```

(e) Define and initialize the fop type for the new fop and associate the corresponding item type

```
struct m0_fop_type m0_rpc_fop_conn_establish_fopt;

/* In module's init function */
foo_subsystem_init()
{
        m0_xc_foo_subsystem_xc_init() /* Provided by ff2c compiler */
        m0_FOP_TYPE_INIT(&m0_rpc_fop_conn_establish_fopt,
                  .name    = "rpc conn establish",
                  .opcode  = m0_RPC_FOP_CONN_ESTABLISH_REP_OPCODE,
                  /* Provided by ff2c */
                  .xt      = m0_rpc_fop_conn_establish_xt,
                  .fop_ops = m0_rpc_fop_conn_establish_ops);
}
```

A request FOP is sent by invoking a rpc routine **m0_rpc_post(),** and its corresponding reply can be sent by invoking **m0_rpc_reply_post()** (as per new rpc layer).

- **Client side**
  Every request fop should be submitted to request handler for processing (both at the client as well as at the server side) which is then forwarded by the request handler itself, although currently (**for "november" demo**) we do not have request handler at the client side. Thus sending a FOP from the client side just involves submitting it to rpc layer by invoking **m0_rpc_post()**.
  So, this may look something similar to this:

```
system_call()->m0t1fs_sys_call()

m0t2fs_sys_call() {
     /* create fop */
     m0_rpc_post();
}
```

- **Server side**
  At server side a fop should be submitted to request handler for processing, invoking **m0_reqh_fop_handle()** and the reply is then sent by one of the

standard/generic phases of the request handler.

Using remote fops (not present in same file) from one fop

The current format of fop operations need all fop formats referenced in the .ff file to be present in the same file. However with introduction of bulk IO client-server, there arises a need of referencing remote fops from one .ff file. Bulk IO transfer needs IO fop to contain a m0_net_buf_desc which is fop itself.
ff2c compiler has a construct called “require” for this purpose. "require" statement introduces a dependency on other source file. For each "require", an #include directive is produced, which includes corresponding header file, "lib/vec.h" in this case

```
require "lib/vec";
```

Example:

```
require "net/net_otw_types";
require "addb/addbff/addb";

sequence {
        u32              id_nr;
        m0_net_buf_desc id_descs
} m0_io_descs;

record {
        u64             if_st;
        m0_addb_record if_addb
} m0_test_io_addb;
```

FOM

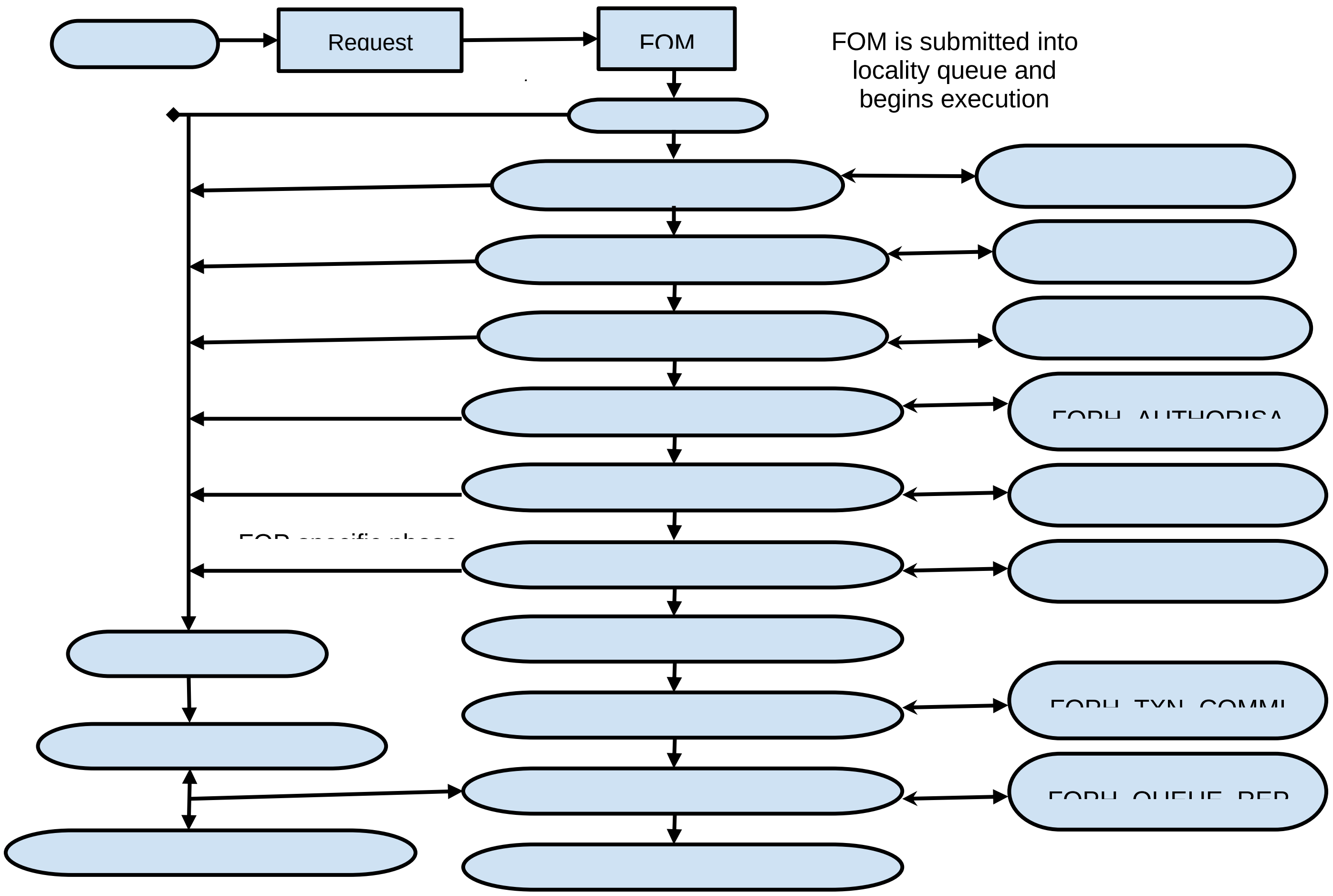
Introduction

Every file operation (FOP) is executed by its corresponding file operation machine (FOM). FOM for the corresponding FOP is instantiated by the request handler when it receives a FOP for execution. Every FOP should have corresponding FOM for its execution.

FOM-writing guidelines

The major purpose of having FOMs and request handler is to have a non-blocking execution of a file operation.

Below diagram shows FOM transition:



Following structures should be defined per FOM:

```
 struct m0_fop_type_ops {
    /** Create a fom that will carry out operation described by the fop. */

    int (*fto_fom_init)(struct m0_fop *fop, struct m0_fom **fom);

...
};

struct m0_fop_type {
         /** Unique operation code. */
         m0_fop_type_code_t ft_code;
...
         /** State machine for this fop type */
```

```
          struct m0_fom_type ft_fom_type;
...
};

/** Type of fom. m0_fom_type is part of m0_fop_type. */
struct m0_fom_type {
    const struct m0_fom_type_ops *ft_ops;

};

/** Fom type operation vector. */
struct m0_fom_type_ops {
    /** Create a new fom of this type. */
    int (*fto_create)(struct m0_fom_type *t, struct m0_fom **out);
};

struct m0_fom_ops {
    /** Finalise this fom. */
    void (*fo_fini)(struct m0_fom *fom)

   /**
        Execute the next phase transition. Returns value of enum
        m0_fom_state_outcome or error code.
    */
   int (*fo_tick)(struct m0_fom *fom);

   /**
      Finds home locality for this fom.
      Returns locality number used as subscript in fd_localities array,
      member of m0_fom_domain, based on fom parameters.
   */
   size_t (*fo_home_locality) (const struct m0_fom *fom);
};
```

FOM execution

A FOP is submitted to request handler through **m0_reqh_fop_handle()** interface
for processing.
Request handler then creates corresponding FOM by invoking

- **m0_fop_type::ft_fom_type::ft_ops::fto_create()**
- **m0_fop_type_ops::ft_fom_init()**

Once the FOM is created, a home locality is selected for the FOM by invoking

- **m0_fom_ops::fo_home_locality()**

After selecting home locality, FOM is then submitted into the locality's run queue for processing.
Every FOM submitted into locality run queue is picked up by the idle locality handler

thread for execution. Handler thread invokes **m0_fom_ops::fo_phase()** (core FOM execution routine also performs FOM phase transitions) method implemented by every FOM.
FOM initially executes its standard/generic phases and then transitions to FOP specific execution phases.
A FOM should check whether it needs to execute a generic phase or a FOP specific phase by checking the phase enumeration.
If the FOM phase enumeration is less than **FOPH_NR + 1,** then the FOM should invoke standard phase execution routine, **m0_fom_state_generic()**, else perform FOP specific operation.
Note:- All the standard phases have enumeration less than the FOP specific phases, thus a FOM writer should keep in mind that the fop specific phases should start from FOPH_NR + 1 (i.e enumeration greater than the standard FOM phase).

Writing a non-blocking FOM

All the operations done by the FOM should be non-blocking. Non-blocking behavior of the FOM in standard/generic phases is handled implicitly, the FOM needs to handle the same explicitly during FOP specific execution phases.
Every potentially blocking FOP specific operation should have a corresponding execution phase as well as waiting phase.
As mentioned previously, every FOM should implement corresponding **fo_state()** method that performs actual state transitions as well as FOP specific operations.

- **Calling Synchronous function from FOM**
    - For synchronous operations, FOM should invoke **m0_fom_block_enter(),** before the operation is started.
    - It creates and adds one or more locality worker threads so that there exist at least one thread to service incoming FOPs.
    - On completion of operation, FOM should call **m0_fom_block_leave(),** this is an undo routine corresponding to **m0_fom_block_enter(),** It terminates any extra idle locality worker threads.
- **Calling Asynchronous function from FOM**
    - For an asynchronous FOM operation, FOM should invoke **m0_fom_block_at(),** and **m0_fom_block_enter()** is not mandatory in this case**.**
    - Before executing a blocking operation, FOM should invoke **m0_fom_block_at()** and register the waiting channel, and transition FOM into its corresponding wait phase. **m0_fom_block_at()** puts the FOM onto locality wait list, so now the thread execution FOM, can pick up the next ready to be executed FOM from the locality run queue and begin its execution.
    - FOM waits until it receives a completion event on the registered channel.
    - On completion of blocking operation, the waiting channel is signaled. FOM is then removed from the locality wait list and put back on the locality runq for further execution.

Sending a reply FOP

- On successful execution, FOM creates the corresponding reply FOP and assigns it to **m0_fom::fo_rep_fop** (reply is sent by the request handler and not the FOM).
- Sending reply fop could be a blocking operation, So this is done by one of the generic or standard phases of the FOM.
- Once FOM execution is complete (that could mean success or failure) FOM sets appropriate reply FOP within the FOM object.
- Once the reply FOP is set, change the FOM phase to **FOPH_SUCCESS** or **FOPH_FAILURE** as per the result of operation and return from the **m0_fom::fo_state() method** (FOM execution routine).
- FOM is then transitioned back to its one of the standard phases (**FOPH_QUEUE_REPLY**) which sends the reply (as mentioned in the above diagram). Once reply is sent, FOM is transitioned back to one of the fop specific phases, in order to perform cleanup operations if any.

Example

Consider a simple write FOM example

- **Declaring FOP in reqh_ut_fom_xc.ff file**

```
record {
        u64 f_seq;
        u64 f_oid
} reqh_ut_fom_fop_fid;

record {
        reqh_ut_fom_fop_fid fiw_object;
        u8                  fiw_value
};
```

As above there are two types of structures defined,

1. reqh_ut_fom_fop_fid, is a structure with native data types, (i.e uint64_t). This is optional, although we would need to build these user defined types separately to use them in other structures.
2. reqh_ut_fom_io_write, is a structure containing an object of struct reqh_ut_fom_fop_fid and a native byte type member.

- **Defining and building a FOP**

```
To build a particular FOP we need to define its corresponding
      m0_fop_type_ops and m0_fop_type structures as follows:-

static struct m0_fop_type_ops reqh_ut_write_fop_ops = {
        .fto_fom_init = reqh_ut_io_fom_init,
};

struct m0_fop_type reqh_ut_fom_io_write_fopt;
```

After defining the above structure, we need to have two subroutines(something like below) which actually builds the FOPs, and adds them to the global FOPs list.

```
/** Function to clean reqh ut io fops */
void reqh_ut_fom_io_fop_fini(void)
{
        m0_fop_type_fini(&reqh_ut_fom_io_write_fopt);
        m0_xc_reqh_ut_fom_xc_fini();
}

/** Function to intialise reqh ut io fops. */
int reqh_ut_fom_io_fop_init(void)
{
        m0_xc_reqh_ut_fom_xc_init();
        return m0_FOP_TYPE_INIT(&reqh_ut_fom_io_write_fopt,
                                .name    = "write",
                                .opcode  = WRITE_REQ,
                                .fop_ops = reqh_ut_write_fop_ops,
                                /* See Below */
                                .fom_ops = reqh_ut_write_fom_ops);
}
```

After defining and building a FOP as above, we can now define its corresponding FOM.

- **Defining a FOM**

Following structures need to be defined per FOM,
FOM operations structure

```
static struct m0_fom_ops reqh_ut_write_fom_ops = {
      .fo_fini = reqh_ut_io_fom_fini,

      .fo_state = reqh_ut_write_fom_state, (implements actual fom operation)
      .fo_home_locality = reqh_ut_find_fom_home_locality,
};
```

FOM type operations structure

```
static const struct m0_fom_type_ops reqh_ut_write_fom_type_ops = {
      .fto_create = reqh_ut_write_fom_create,
};
```

FOM type structure, this is embedded inside struct m0_fop_type,

```
static struct m0_fom_type reqh_ut_write_fom_mopt = {
      .ft_ops = &reqh_ut_write_fom_type_ops,
};
```

A typical fom state function would look something similar to this:-

```
int reqh_ut_write_fom_state(struct m0_fom *fom)
{
        ...
        /*
           checks if FOM should transition into a generic/standard
           phase or FOP specific phase.
        */
        if (fom->fo_phase < FOPH_NR) {
                result = m0_fom_state_generic(fom);
        } else {
            ...
                 /* FOP specific phase */
                 if (fom->fo_phase == FOPH_WRITE_STOB_IO) {
                     ...
                      /* For synchronous FOM operation */
                        m0_fom_block_enter(fom);
                      /* For asynchronous FOM operation */
                        m0_fom_block_at(fom,
                                   &fom_obj->rh_ut_stio.si_wait);
                        result =
                           m0_stob_io_launch(&fom_obj->rh_ut_stio,
                                          fom_obj->rh_ut_stobj,
                                          &fom->fo_tx, NULL);
                     ...
                        if (result != 0) {
                            fom->fo_rc = result;
                            fom->fo_phase = FOPH_FAILURE;
                        } else {
                            fom->fo_phase =
                                          FOPH_WRITE_STOB_IO_WAIT;
                            result = FSO_WAIT;
                        }
                } else if (fom->fo_phase ==
                                        FOPH_WRITE_STOB_IO_WAIT) {
                       /*
                          Terminate extra idle threads created
                          by m0_fom_block_enter()
                        */
                       m0_fom_block_leave(fom);

                   ...
```

```
                        if (fom->fo_rc != 0)
                              fom->fo_phase = FOPH_FAILURE;
                       else {
                              …
                              fom->fo_phase = FOPH_SUCCESS;
                                                    }
                             }

      if (fom->fo_phase == FOPH_FAILURE || fom->fo_phase ==
                                       FOPH_SUCCESS) {
                        ...
                    result = FSO_AGAIN;
      }
...
```

Note:- For additional details on the implementation of above methods please refer to request handler UT code in **`colibri/core/reqh/ut/reqh_fom_ut.c`**