Mero Resource Manager (RM) Interface

Note: this document simply collects information from RM HLD and DLD and tries to organise from the perspective of a ‘resource type’ developers, helping them understand the basic concepts in RM and know how to develop a new ‘resource type’.

1. RM Notations

1.1 Resource, Resource Type and Resource Owner

A resource (`m0_rm_resource`) is associated with various file system entities:

- file metadata. Credits to use this resource can be thought of as locks on file attributes that allow them to be cached or modified locally.
- file data. Credits to use this resource are extents in the file plus access mode bits (read, write).
- free storage space on a server (a "grant" in Lustre terminology). Credit to use this resource is a reservation of a given number of bytes
- quota.

A resource owner (defined by `m0_rm_owner`) represents a collection of credits to use a particular resource. A resource owner uses the resource via a usage credit (also called resource credit or simply credit as context permits). E.g., a client might have a credit of a read-only or write-only or read-write access to a certain extent in a file. An owner is granted a credit to use a resource. A resource belongs to a specific resource type, which determines resource semantics.

1.2 Credit

A resource owner uses the resource via a `usage credit` (also called `resource credit` or simply `credit` as context permits). E.g., a client might have a credit of a read-only or write-only or read-write access to a certain extent in a file. An owner is granted a credit to use a resource.

Various terms are used to described credit flow of the resources in a cluster. Owners of credits for a particular resource are arranged in a cluster-wide hierarchy. This hierarchical arrangement depends on system structure (e.g., where devices are connected, how network topology looks like) and dynamic system behaviour (how accesses to a resource are distributed).

Originally, all credits on the resource belong to a single owner or a set of owners, residing on some well-known servers. Proxy servers request and cache credits from there. Lower level proxies and clients request credits in turn. According to the order in this hierarchy, one distinguishes "upward" and "downward" owners relative to a given one.

In a given ownership transfer operation, a downward owner is "debtor" and upward owner is "creditor". The credit being transferred is called a "loan" (note that this word is used only as a noun). When a credit is transferred from a creditor to a debtor, the latter "borrows" and the

former "sub-lets" the loan. When a credit is transferred in the other direction, the creditor "revokes" and debtor "returns" the loan.
A debtor can voluntary return a loan. This is called a "cancel" operation.

1.3 RM incoming request and RM remote request (outgoing)

To use a resource, a user of the resource manager creates an incoming resource request (`m0_rm_incoming`), that describes a wanted usage credit. An incoming request is created for

- local credit request, when some user wants to use the resource
- remote credit request from a "downward" owner which asks to sub-let some credits;
- remote credit request from an "upward" owner which wants to revoke some credits.

Sometimes the request can be fulfilled immediately, sometimes it requires changes in the credit ownership. In the latter case outgoing requests are directed to the remote resource owners (which typically means a network communication) to collect the wanted usage credit at the owner. When an outgoing request reaches its target remote domain, an incoming request is created and processed (which in turn might result in sending further outgoing requests). Eventually, a reply is received for the outgoing request. When incoming request processing is complete, it "pins" the wanted credit. This credit can be used until the incoming request structure is destroyed and the pin is released.

An outgoing request is created on behalf of some incoming request to track the state of credit transfer with some remote domain. An outgoing request is created to:

- borrow a new credit from some remote owner (an "upward" request) or
- revoke a credit sublet to some remote owner (a "downward" request) or
- cancel this owner's credit and return it to an upward owner.

1.4 RM Service

Resource manager service (RMS) provides service management API for resource manager. RMS is registered with Mero request handler and the service will provide interfaces to resource manager, like identification of owner, borrow or revoke requests. Note that there will be multiple resource manager services running in the system, the locations of other resource manager services can be obtained from `confd`.

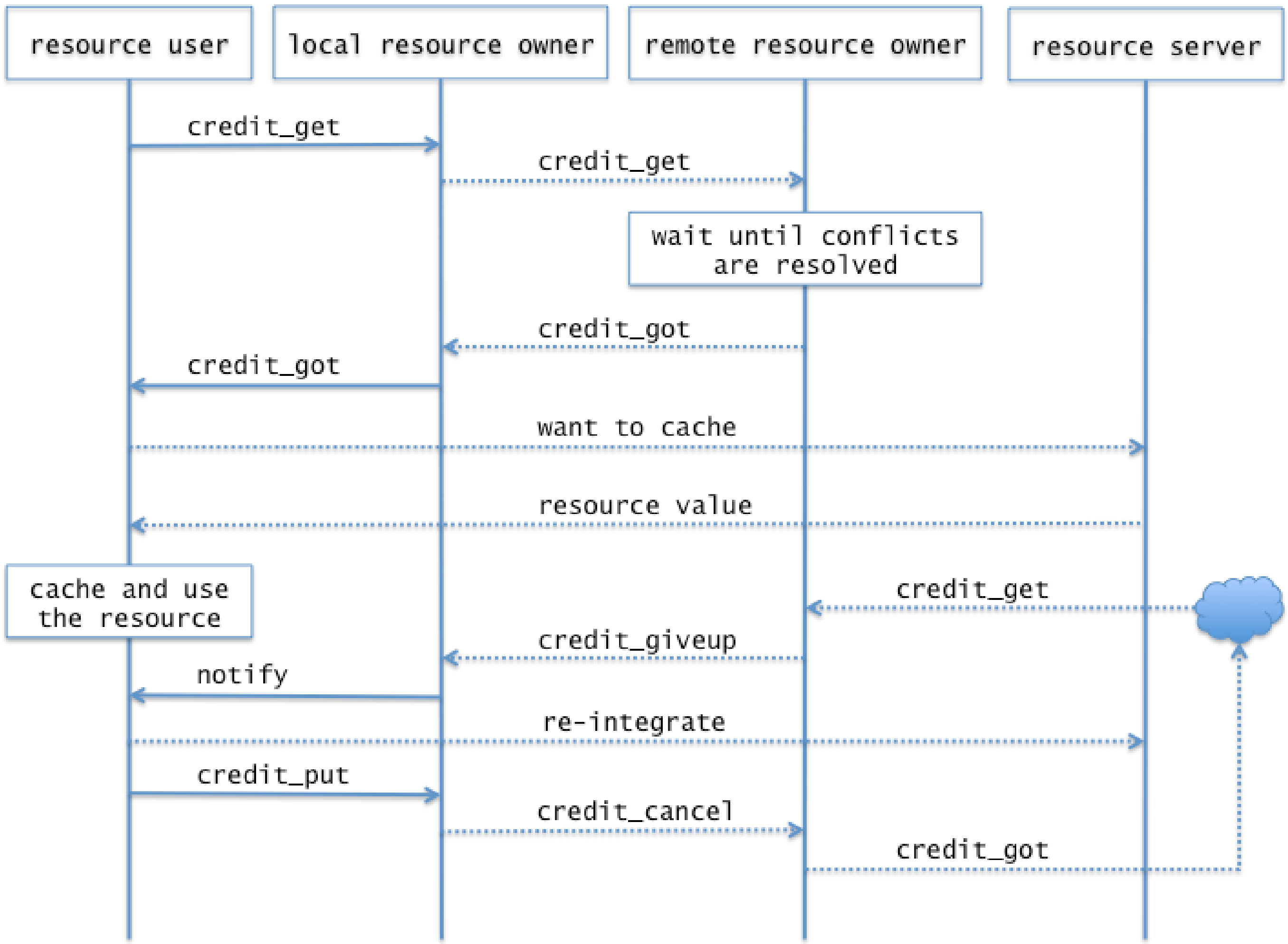

Figure 1. Interaction between resource users, resource owners and resource servers.

2. RM APIs

Resource management is split into two parts: (1) Generic functionality which provides the and is implemented by the code in rm/ directory; (2) resource type specific functionality. This part contains 3 operation vectors (m0_rm_resource_ops, m0_rm_resource_type_ops and m0_rm_credit_ops) provided by a resource type and called by the generic code. An RM application (such as the file lock implementation in file/file.c) has to provide the definition of resource type and these 3 operation vectors when it is registered.

2.1 Resource Type

```
struct m0_rm_resource_type {
      const struct m0_rm_resource_type_ops *rt_ops;
```

```
        const char                              *rt_name;
        uint64_t                                 rt_id;

        struct m0_mutex                          rt_lock;
        struct m0_tl                             rt_resources;

        /**
         * Active references to this resource type from resource instances
         * (m0_rm_owner::ro_resource). Protected by
         * m0_rm_resource_type::rt_lock.
         */
        uint32_t                                 rt_nr_resources;
        struct m0_sm_group                       rt_sm_grp;
        /**
         * Executes ASTs for this owner.
         */
        struct m0_thread                         rt_worker;
        /**
         * Flag for ro_worker thread to stop.
         */
        bool                                    rt_stop_worker;
        /**
         * Domain this resource type is registered with.
         */
        struct m0_rm_domain                    *rt_dom;
};
```

Description:
Resources are classified into disjoint types. Resource type determines how its instances interact with the resource management generic core and defines:

- how the resources of this type are named by the fields `rt_name` and `rt_id`. `rt_id` is the resource type identifier, globally unique within a cluster, used to identify resource types on wire and storage. This identifier is used as an index in m0_rm_domain::rd_types. where the resources of this type are located;
- where the resources of this type are located: a list of resources of this type is linked in `rt_resources`;
- A pointer to `m0_rm_resource_type_ops`.

2.2 m0_rm_resource_type_ops

```
struct m0_rm_resource_type_ops {
      bool (*rto_eq)(const struct m0_rm_resource *resource0,
                     const struct m0_rm_resource *resource1);

      bool (*rto_is)(const struct m0_rm_resource *resource,
                     uint64_t id);

      m0_bcount_t (*rto_len) (const struct m0_rm_resource *resource);

      int  (*rto_decode)(struct m0_bufvec_cursor *cur,
                         struct m0_rm_resource **resource);

      int  (*rto_encode)(struct m0_bufvec_cursor *cur,
                         const struct m0_rm_resource *resource);
};
```

Description:
m0_rm_resource_type_ops defines a set of operations for a resource type:

- `m0_rm_resource_type_ops::rto_eq` checks if the two resources are equal.
- `m0_rm_resource_type_ops:rto_is` checks if the resource has "id".
- `m0_rm_resource_type_ops:rto_len` return the size of the resource data.
- `m0_rm_resource_type_ops:rto_decode` deserialises the resource from a buffer.
- `m0_rm_resource_type_ops:rto_decode` serialise a resource into a buffer.

2.3 m0_rm_resource_ops

```
struct m0_rm_resource_ops {
      int (*rop_credit_decode)(struct m0_rm_resource *resource,
                               struct m0_rm_credit *credit,
                               struct m0_bufvec_cursor *cur);

      void (*rop_policy)(struct m0_rm_resource *resource,
                         struct m0_rm_incoming *in);

      void (*rop_credit_init)(struct m0_rm_resource *resource,
                         struct m0_rm_credit *credit);

      void (*rop_resource_free)(struct m0_rm_resource *resource);
};
```

Description:
m0_rm_resource_type_ops defines a set of resource specific operations

- m0_rm_resource_ops::rop_credit_decode is called when a new credit is allocated for the resource. The resource specific code should parse the credit description stored in the buffer and fill m0_rm_credit::cr_datum appropriately.
- m0_rm_resource_ops::rop_policy decides which credit should be granted, sublet, or revoked. "Policy" defines which credit to actually grant. For example, a client doing a write to the first 4KB page in a file asks for [0, 4KB) extent lock. If nobody else accesses the file, RM would grant [0, ~0ULL) lock instead to avoid repeated lock requests in case of sequential IO. If there are other conflicting locks, already granted on the file, the policy might expand requested credit to the largest credit that doesn't overlap with conflicting credits. And so on, there are multiple options. So this is literally a "credit policy" as used by banks. The name stems all the way back to VAX VMS lock manager. m0_rm_incoming_policy defines a list of a few predefined policies.
- m0_rm_resource_ops::rop_credit_init initialises a usage credit for this resource and sets up m0_rm_credit::cr_ops.

2.4 m0_rm_credit_ops

```
struct m0_rm_credit_ops {
      void (*cro_free)(struct m0_rm_credit *self);

      int  (*cro_encode)(struct m0_rm_credit *self,
                         struct m0_bufvec_cursor *cur);
      int  (*cro_decode)(struct m0_rm_credit *self,
                         struct m0_bufvec_cursor *cur);
      m0_bcount_t (*cro_len) (const struct m0_rm_credit *self);

      bool (*cro_intersects) (const struct m0_rm_credit *self,
                              const struct m0_rm_credit *c1);
      bool (*cro_is_subset) (const struct m0_rm_credit *self,
                             const struct m0_rm_credit *c1);

      int (*cro_join) (struct m0_rm_credit *self,
                       const struct m0_rm_credit *c1);
      int (*cro_disjoin) (struct m0_rm_credit *self,
                          const struct m0_rm_credit *c1,
                          struct m0_rm_credit *intersection);

      bool (*cro_conflicts) (const struct m0_rm_credit *self,
                             const struct m0_rm_credit *c1);
      int  (*cro_diff)(struct m0_rm_credit *self,
                       const struct m0_rm_credit *c1);

      int  (*cro_copy)(struct m0_rm_credit *dst,
                  const struct m0_rm_credit *self);
```

```
    void (*cro_initial_capital)(struct m0_rm_credit *self);
};
```

m0_rm_credit_ops defines a set of credit-related operations for a resource type.

- `m0_rm_credit_ops::cro_free` is called when the generic code is about to free a credit. Type specific code releases any resources associated with the credit.

- `m0_rm_credit_ops::cro_encode/decode` serialises/deserialises a credit of a resource to/from a buffer. `m0_rm_credit_ops::cro_len` returns the size of the credit's data.
- `m0_rm_credit_ops::cro_intersect` returns True iff 2 credits are intersected.Credits intersect when there is a overlapped usage authorised by 2 credits in question. For example, a credit to read an extent [0, 100] (denoted R:[0, 100]) intersects with a credit to read or write an extent [50, 150], (denoted RW:[50, 150]) because they can be both used to read bytes in the extent [50, 100]. "Intersects" is assumed to satisfy the following conditions:
    - intersects(A, B) iff intersects(B, A) (symmetrical)
    - (A != 0) iff intersects(A, A) (almost reflexive)
    - !intersects(A, 0)
- `m0_rm_credit_ops::cro_is_subset` returns True if 'self credit' is subset (or proper subset) of c1.
- `m0_rm_credit_ops::cro_join` adjoins 2 credits, `self` and `c1`, updating credit `self` to be the sum credit. `m0_rm_credit_ops::cro_join` splits self into two parts - `diff(self,c1)` and `intersection(self, c1)` destructively updates `'self'` credit with `diff(self, c1)` and updates intersection with `intersection of (self, c1)`.
- `m0_rm_credit_ops::cro_conflict` returns iff `'self'` credit conflicts with another credit `c1`.Credits conflict iff one of them authorises a usage incompatible with another.
- `m0_rm_credit_ops::cro_diff` returns the difference between credits.The difference is a part of self that doesn't intersect with c1. This function destructively updates "self" in place. For example, `diff(RW:[50, 150], R:[0, 100]) == RW:[101, 150]`. `X <= Y` means that `diff(X, Y) is 0`. `X >= Y` means `Y <= X`. Two credits are equal, `X == Y, when X <= Y and Y <= X`. "Difference" must satisfy the following conditions:
    - `diff(A, A) == 0`
    - `diff(A, 0) == A`
    - `diff(0, A) == 0`
    - `!intersects(diff(A, B), B)`
    - `diff(A, diff(A, B)) == diff(B, diff(B, A))`.

- `diff(A, diff(A, B))` is called a "meet" of A and B, it's an intersection of credits A and B. The condition above ensures that `meet(A, B) == meet(B, A)`
    - `diff(A, B) == diff(A, meet(A, B))`
    - `meet(A, meet(B, C)) == meet(meet(A, B), C)`
    - `meet(A, 0) == 0, meet(A, A) == A, &c.,`
    - `meet(A, B) <= A`
    - `(X <= A and X <= B) iff X <= meet(A, B),`
    - `intersects(A, B) iff meet(A, B) != 0.`
- `m0_rm_credit_ops::cro_copy` creates a copy of "src" in "dst".
- `m0_rm_credit_ops::cro_copy` setups initial capital for a credit.

2.5 Register and unregister a resource type

```
M0_INTERNAL int m0_rm_type_register(struct m0_rm_domain *dom,
                                    struct m0_rm_resource_type *rt);

M0_INTERNAL void m0_rm_type_deregister(struct m0_rm_resource_type
*rtype);
```

Description:
`m0_rm_type_register` registers a resource type with a domain.
`m0_rm_type_deregister` deregisters a resource type.

2.6 Other helper functions for an RM application

References:

[1] RM HLD.
https://docs.google.com/document/d/1WYw8MmItpp0KuBbYfuQQxJaw9UN8OuHKnlICszB8-Zs/edit#heading=h.xdogq7ycp49a
[2] RM DLD. rm/rm.h in Mero source tree.