HLD of FDMI

This document presents a high level design (HLD) of Mero's FDMI interface.

CONTENTS

Introduction
- Document Purpose
- Intended Audience
- Definitions, Acronyms, and Abbreviations
- References

Overview
- Product Purpose
- Assumptions and Limitations

Architecture
- FDMI position in overall Mero Core design
- FDMI roles

FDMI Plug-in Dock
- Initialization
- Data Processing
- Filter "active" status
- De-initialization

FDMI Source Dock
- Initialization
- Input Data Filtering
- Deferred Input Data Release
- FDMI service found dead

Interfaces
- FDMI service
  - Structures
  - Functions
- FDMI source registration
- FDMI source implementation guideline
- FDMI record post
  - Enumerations
  - Structures
  - Functions
- FDMI source dock FOM

- Normal workflow
- Structures
- Filters set support
- Corner cases
- FilterD
- FilterC
- FDMI plugin registration
  - Enumerations
  - Structures
  - Functions
- FDMI plugin dock FOM
  - Structures
- FDMI plugin implementation guideline
  - Plug-in responsibilities
    - During standard initialization workflow plug-in:
    - During active subscription workflow looks like following:

Overall System Description

- Composition
  - Run-Time Composition

Revision History

DATE	REVISION TYPE	REVISION #	COMMENTS	INITIALS
2014-Aug-29	Major	1.0		IV
2014-Sep-16	Major	1.1	Modifications according to ND review notes	VS
2014-Oct-02	Minor	1.2	Point about mandatory walking through the whole filter set is taken. Accepted.	IV
2014-Oct-08	Minor	1.3	Changed testing filter against FDMI record logic	EN
2014-Oct-16	Major	1.4	Revision of Plug-in Dock filter concept	IV
2014-Oct-22	Minor	1.5	Design finalisation	IV

Introduction

The document is intended to specify the design for of Mero FDMI interface. FDMI is a part of Mero product. FDMI provides interface for Mero plugins and allows horizontally extending the features and capabilities of the system. The intended audience for this document are product architects, developers and QA engineers.

Definitions

FDMI -- File data manipulation interface
FDMI source
FDMI plugin
FDMI source dock
FDMI plugin dock
FDMI record
FDMI record type
FDMI filter

References

- Mero Object Store Architecture
- Mero technical (toi)
- Fdmi High Level Decomposition

OVERVIEW

overview

Mero is a storage core capable of deployment for a wide range of large scale storage regimes, from cloud and enterprise systems to exascale HPC installations. FDMI is a part of Mero core, providing interface for plugins implementation. FDMI is build around the core and allows for horizontally extending the features and capabilities of the system in a scalable and reliable manner.

product purpose

TBD

assumptions and limitations

TBD

architecture

In this section only architectural information like the following but not limited to:

1) *Common design strategy including*
    - *General approach to the decomposition*
    - *Chosen architecture style and template if any*
2) *Key aspects and consideration that affect on the other design*

fdmi position in overall mero core design

FDMI is an interface allowing Mero Core to scale horizontally. The scaling includes two aspects:

- Core expansion in aspect of adding core data processing abilities, including data volumes as well as transformation into alternative representation. The expansion is provided by introducing FDMI plug-ins.
- Initial design implies that FOL records are the only data plug-ins are able to process so far.
- Core expansion in aspect of adding new types of data the core is able to feed plug-ins. This sort of expansion is provided by introducing FDMI sources.
- Initial design implies that FOL record is the only source data type Mero Core provides so far.

  FDMI plug-in is an application linked with Mero Core to make use of corresponding FDMI interfaces and runs as a part of Mero instance/service. This Mero instance may provide various capabilities (data storing, etc.) or may not. The purpose of introducing plug-in is getting notifications from Mero Core on events of interest for the plugin and further post-processing of the received events for producing some additional classes of service, the Core currently is not able to provide.

  FDMI source is a part of Mero instance being linked with appropriate FDMI interfaces and allowing connection to additional data providers.

  Considering the amount of data Mero Core operates with, it is obvious that a plug-in typically requires a sufficiently reduced bulk of data to be routed to it for post-processing. The reduction is provided by introduction of mechanism of subscription to particular data types and conditions met at runtime. The subscription mechanism is based on set of filters the plug-in registers in Mero Filter Database during its initialization.

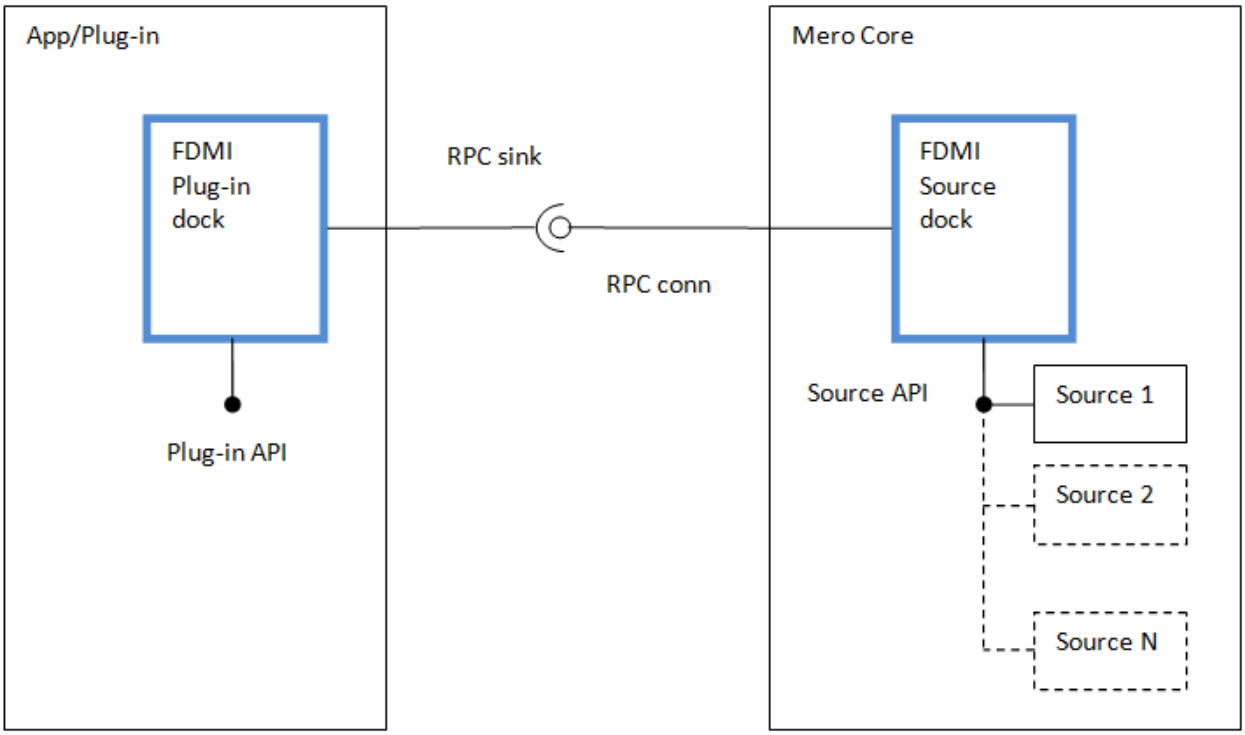
  Source in its turn refreshes its own subset of filters against the database. The subset is selected from overall filter set based on the knowledge about data types the source is able to feed FDMI with as well as operation with the data the source supports.

fdmi roles

FDMI consists of APIs implementing particular roles in accordance with FDMI use cases. The roles are:

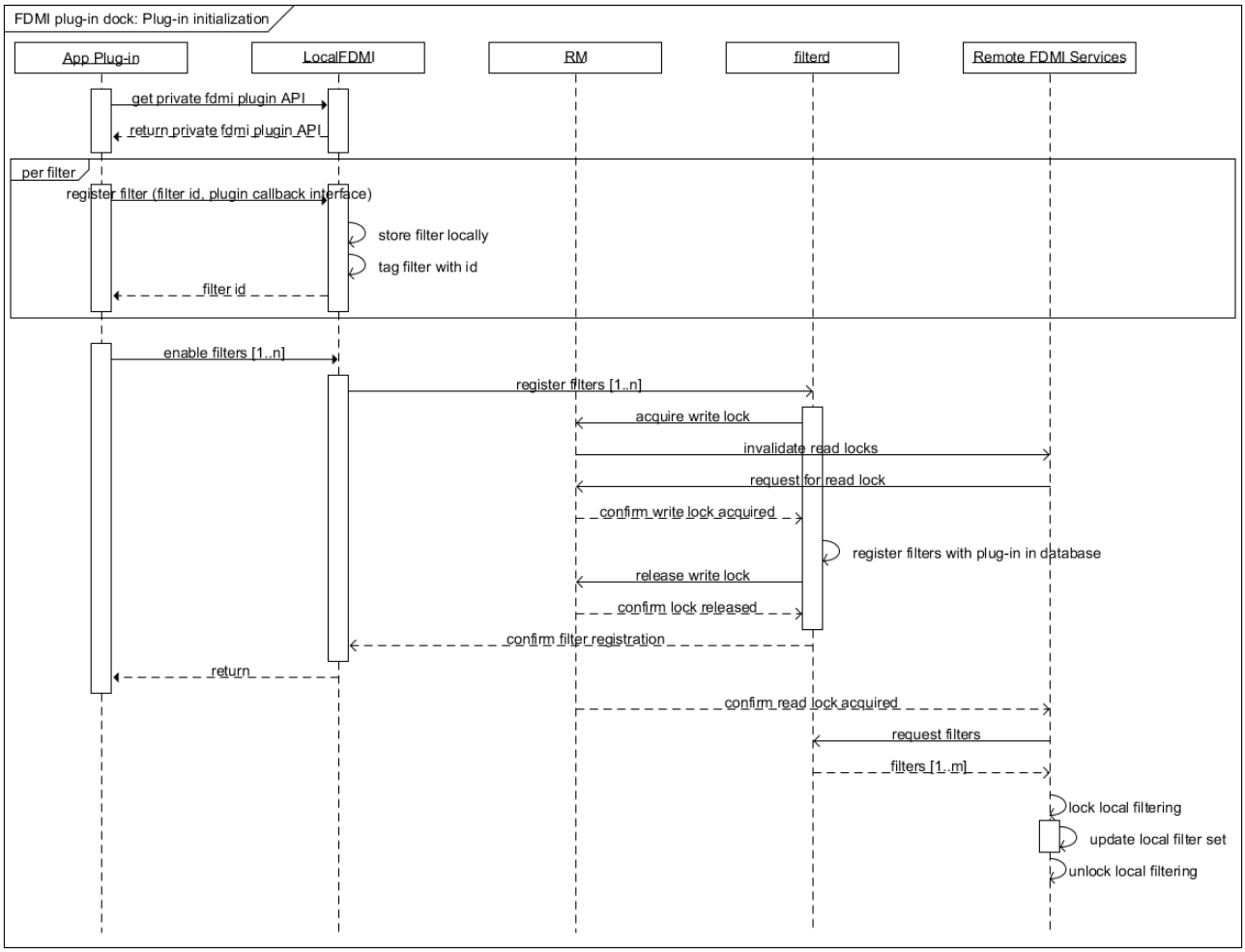
- Plug-in dock, responsible for:

    - Plug-in registration in FDMI instance
    - Filter registration in Mero Filter Database
    - Listening to notifications coming over RPC
    - Payload processing
    - Self-diagnostic (TBD)
- Source dock (FDMI service), responsible for:
    - Source registration
    - Retrieving/refreshing filter set for the source
    - Input data filtration
    - Deciding on and posting notifications to filter subscribers over Mero RPC
    - Deferred input data release
    - Self-diagnostic (TBD)



fdmi plug-in dock

1.5 Initialization



Application starts with getting private FDMI plugin dock API allowing it start communicating with the dock.

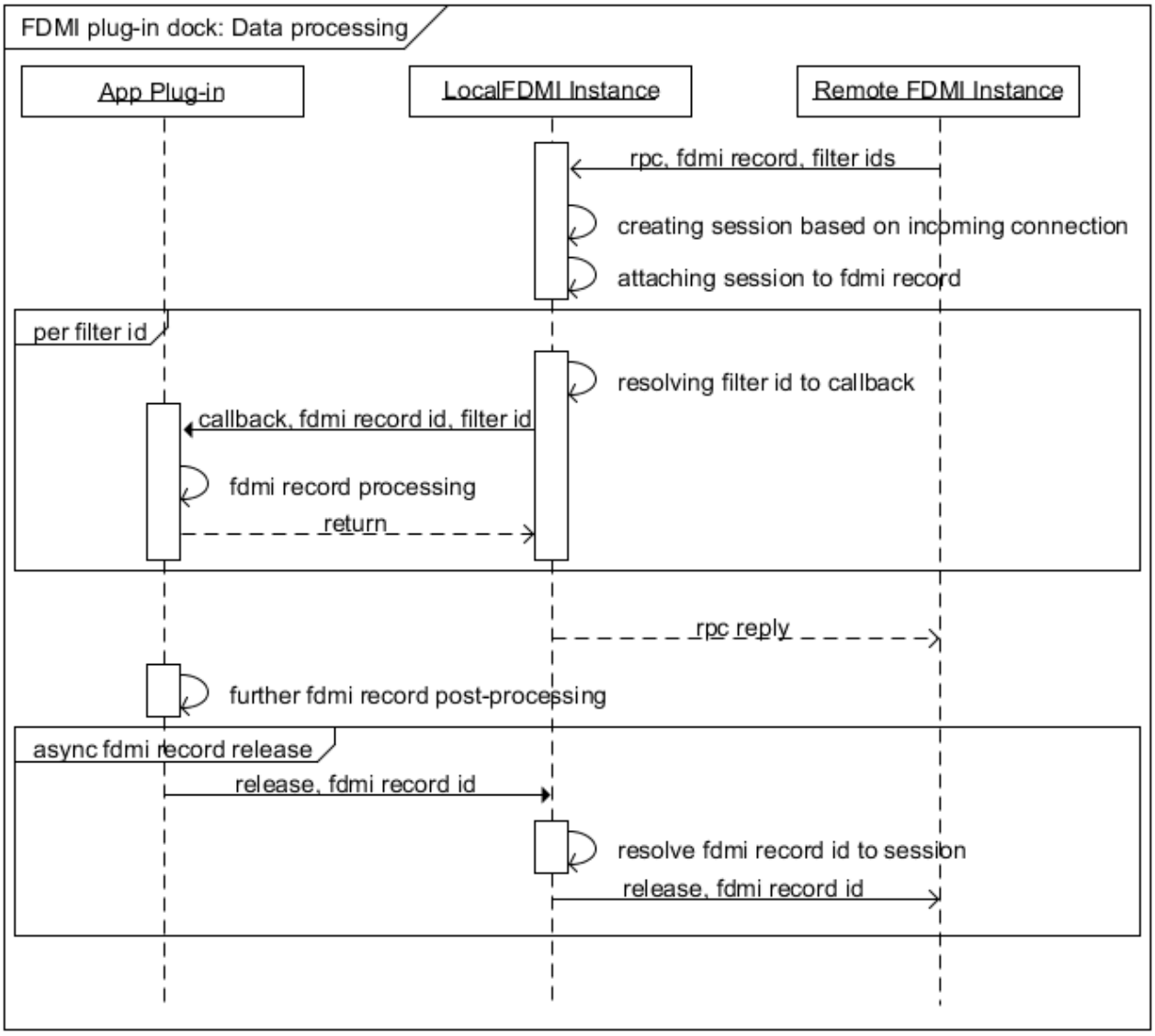
Further initialisation consists of registering a number of filters in **filterd** database. Every filter instance is given by plugin creator with a filter id unique across the whole system.
On filter registration plugin dock checks filter semantics. If filter appears to be invalid, registration process stops.

NB:
Although filter check is performed on registration it cannot be guaranteed that no error appears in run time during filter condition check. Criteria for filter

correctness will be defined later. If filter is treated as incorrect by FDMI source dock, corresponding ADDB record is posted and optionally HA is informed on it.

NB:
TBD if we really need to determine the moment when all sources appear to be running filter sets consistent across the whole system. Currently we need to consider if Plug-in should be notified about this point.

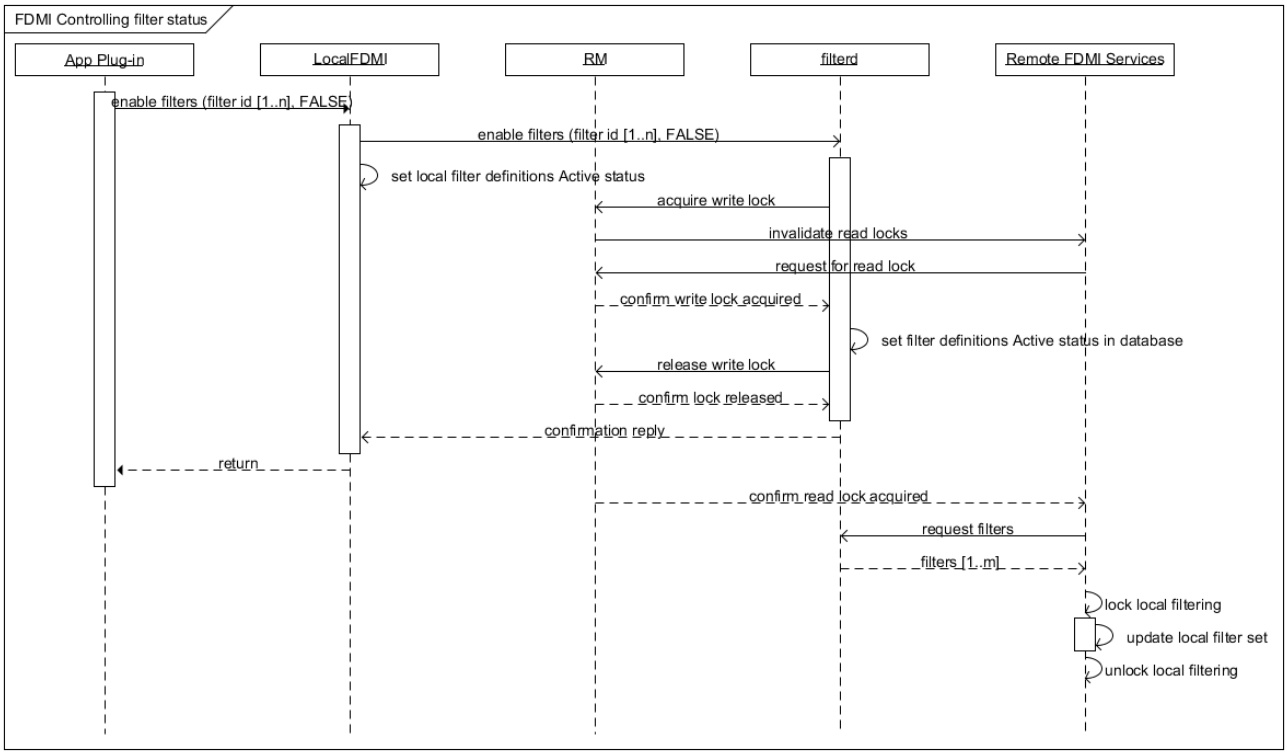
1.6 Data Processing



Remote FDMI instance running Source Dock role provides data payload via RPC channel. RPC sink calls back local FDMI instance running Plug-in Dock role. The latter resolves the filter id to plug-in callback, and calls the one passing the data to plug-in instance.

It may take some time for plug-in to do post-processing and decide if the fdmi record could be released. At the time plug-in instructs FDMI to notify corresponding source allowing particular fdmi record be released.

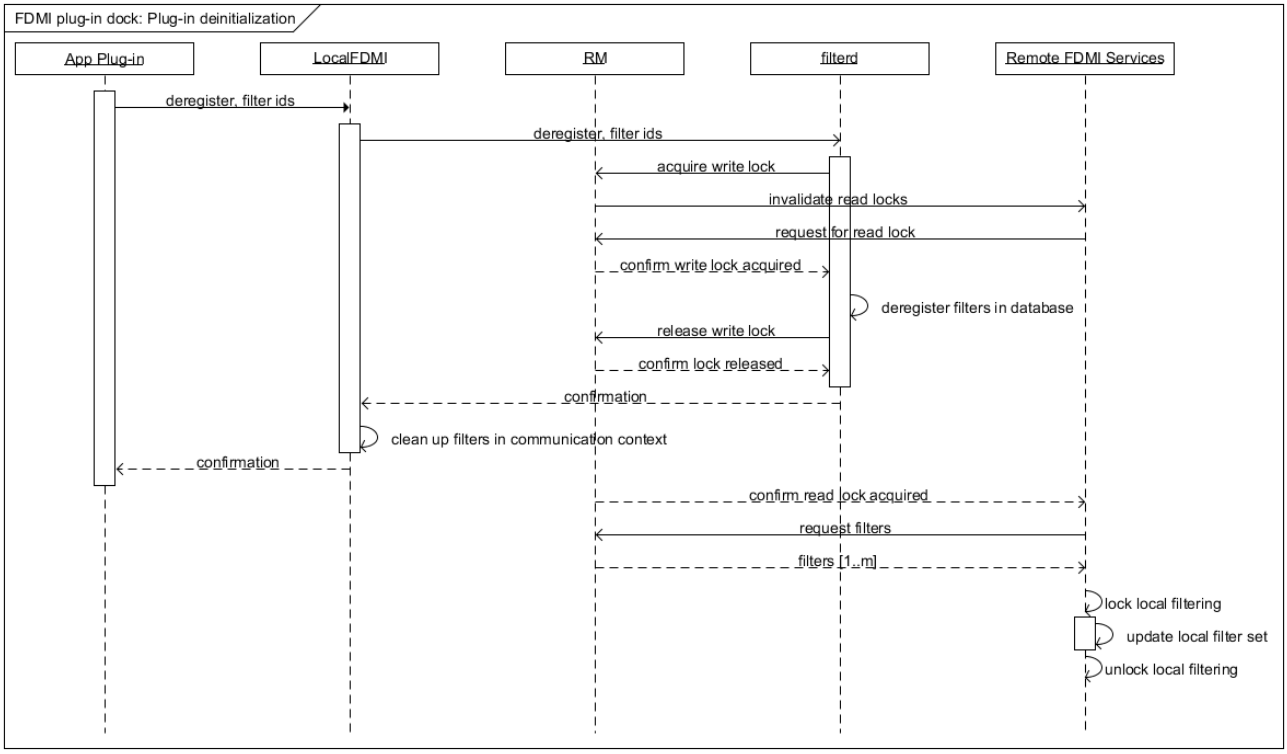
1.7 Filter “active” status



Filter “active” status is used for enabling/disabling this filter on the fly without removing it from **filterd** database. Upon filter active status change **filterd** notifies all the registered sources. If filter “active” status is set to false (filter is disabled), it is ignored by sources.
Application plugin may change filter “active” status by sending “enable filter” or “disable filter” command for the already registered filter:

- Filter “active“ status initial value is specified on filter registration
- To enable/disable a filter on the fly, application sends “enable filter” or “disable filter” request to **filterd** service. Filter ID is specified as a parameter.

1.8 De-initialization

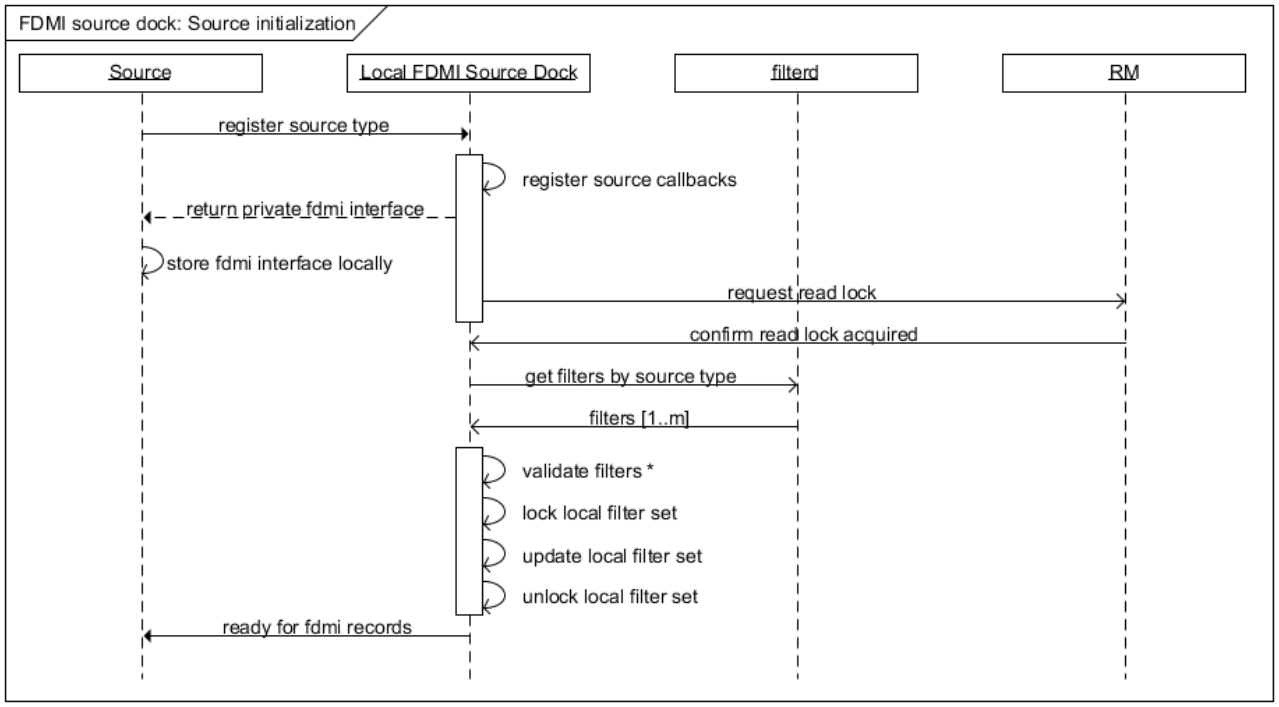


Plug-in initiates de-initialisation by calling local FDMI. The latter deregisters plug-in's filter set with **filterd** service. After confirmation it deregisters the associated plug-in's callback function.

All registered sources are notified about changes in filter set, if any occurred as the result of plug-in coming off.

fdmi source dock

1.9 Initialization

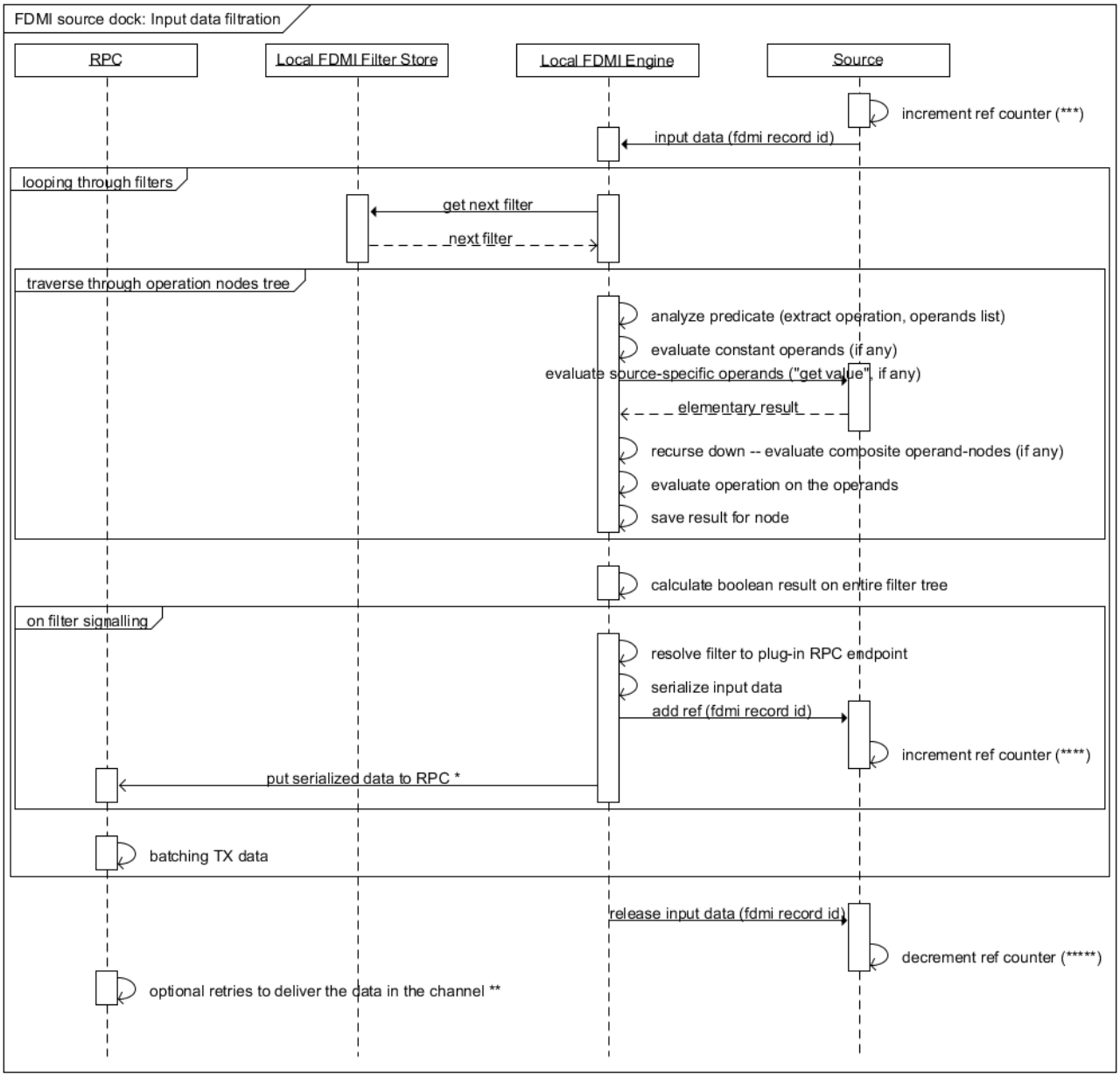


* TBD where to validate, on Source side or inside FDMI

FDMI Source dock does not need explicit registration in **filterd**. Each FDMI source dock on start requests filters list from **filterd** and stores it locally.
In order to notify FDMI source dock about ongoing changes in filter data set, Resource manager's locks mechanism is used. *Filters change notification: TBD. On read operation, FDMI source acquires Read lock for the **filterd** database. On filter metadata change, each instance holding read lock is being notified.*
On receiving filter metadata change notification, FDMI source dock re-requests filter data set.
On receiving each new filter, FDMI source dock parses it, checks for its consistency and stores its in-memory representation suitable for calculations.

As an optimization, execution plan could be built for every added filter to be kept along with the one. As an option, execution plan can be built on every data filtering action to trade off memory consumption for CPU ticks.

1.10 Input Data Filtering



* In case of RPC channel failure, input data reference counter has to be decremented. See Deferred Input Data Release.

** RPC mechanism is responsible for reliable data delivery, and is expected to do its best for re-sending data appeared to be stuck in the channel. The same way it is responsible for freeing RPC items once the connection found broken.

Steps (***) and (*****) are needed to lock data entry during internal fdmi record processing -- to make sure the source would not dispose it before FDMI engine evaluates all filters. Step (***), on the other

hand, increases the counter for each record FDMI sends out. Matching decrement operation is not displayed on this diagram, it's discussed later.

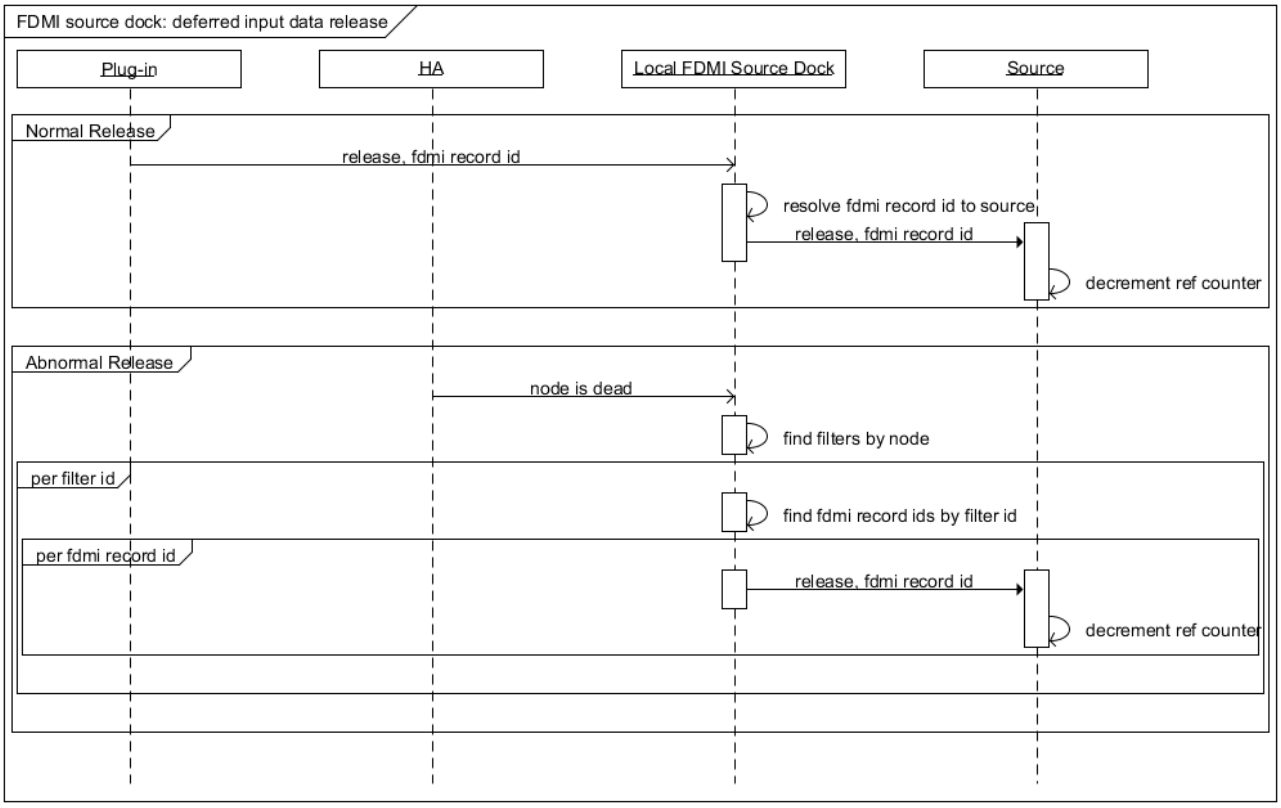
When input data identified by **fdmi record id** comes to Source, the latter calls local FDMI instance with the data. On data coming FDMI starts iterating through local filter set.
According to its in-memory representation (execution plan) each filter is traversed node by node, and for every node a predicate result is calculated by appropriate source callback.

NB:
It is expected that source will be provided with operand definitions only. Inside the callback the source is going to extract corresponding operand according to the description passed in. And predicate result is calculated based on the extracted and transformed data.

Note how FDMI handles tree: all operations are evaluated by FDMI engine, and only getting atomic values from the fdmi record payload are delegated to Source.

When done with traversing, FDMI engine calculates final Boolean result for the filter tree, and makes a decision whether to put serialized input data onto RPC for the plug-in associated with the filter.

1.11 Deferred Input Data Release

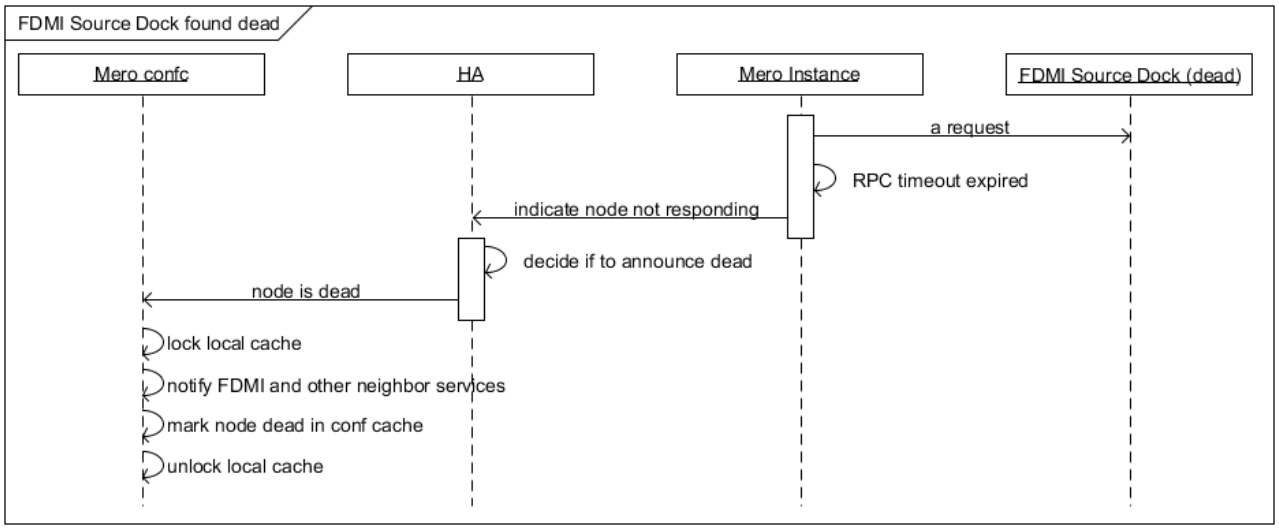


Input data may require to remain preserved in the Source until the moment when plug-in does not need it anymore. The preservation implies protection from being deleted/modified. The data processing inside plug-in is an asynchronous process in general, and plug-in is expected to notify corresponding source allowing it to release the data. The message comes from plug-in to FDMI instance hosting the corresponding source.

NB:

TBD: We need to establish a way to resolve fdmi record identifier to FDMI instance hosting particular source. Most probably the identifier itself may contain the information, easily deduced or calculated.

1.12 FDMI service found dead

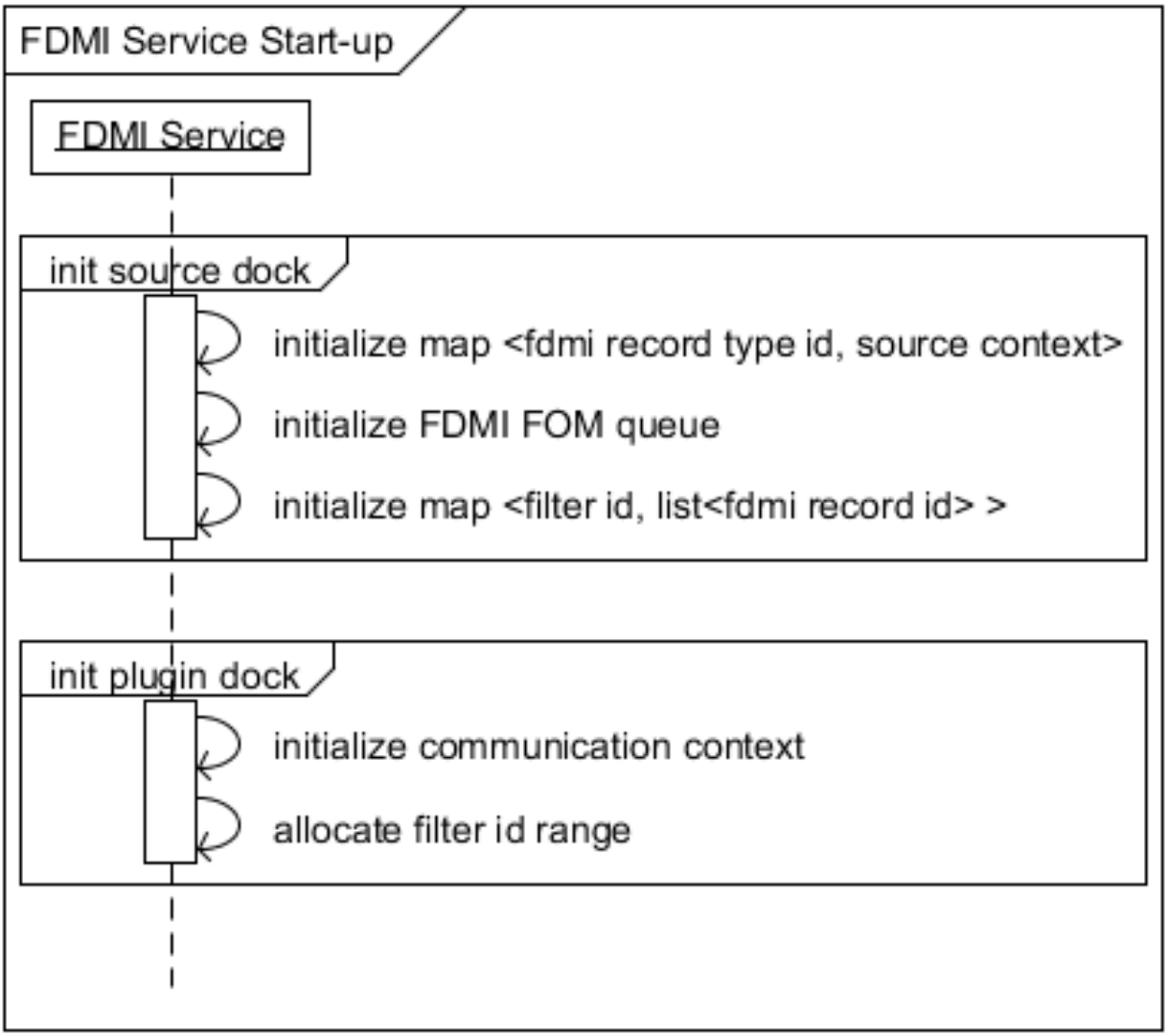


When interaction between Mero services results in a timeout exceeding pre-configured value, the not responding service needs to be announced dead across the whole system. First of all **confc** client is notified by HA about the service not responding and announced dead. After being marked dead in **confc** cache, the service has to be reported by HA to **filterd** as well.

interfaces

1) FDMI service
2) FDMI source registration
3) FDMI source implementation guideline
4) FDMI record
5) FDMI record post
6) FDMI source dock FOM
   a. Normal workflow
   b. FDMI source: filters set support
   c. Corner cases (plugin node dead)
7) FilterD
8) FilterC
9) FDMI plugin registration
10) FDMI plugin dock FOM
11) FDMI plugin implementation guideline

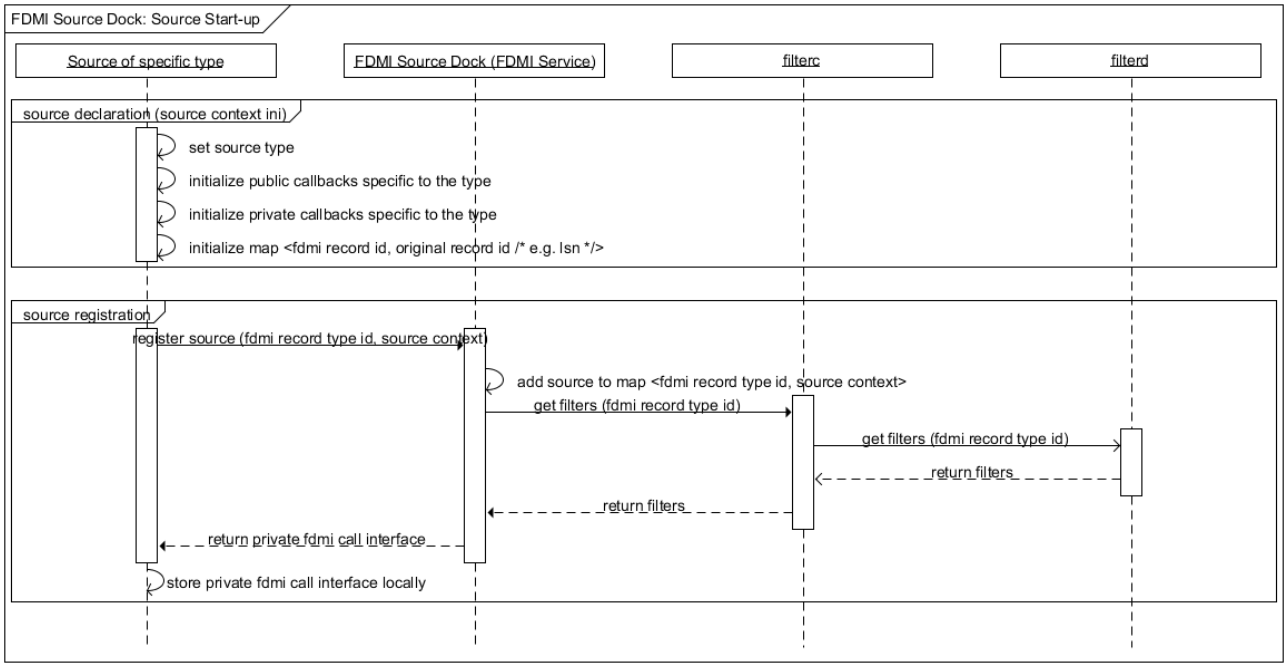
fdmi service



FDMI service runs as a part of Mero instance. FDMI service stores context data for both, FDMI source dock and FDMI plugin dock. FDMI service is initialized and started on Mero instance start up, FDMI Source dock and FDMI plugin dock are both initialised on the service start unconditionally.

TBD:
Later the docks can be managed separately, and specific API may be provided for this purposes.

fdmi source registration



FDMI source instance main task is to post FDMI records of a specific type to FDMI source dock for further analysis, Only 1 FDMI source instance with a specific type should be registered: FDMI record type uniquely identifies FDMI source instance. A list of FDMI record types:

- FOL record type
- ADDB record type
- TBD

FDMI source instance provides the following interface functions for FDMI source dock to handle FDMI records:

- Test filter condition
- Increase/decrease record reference counter
- Xcode functions

On FDMI source registration all its internals are initialized and saved as FDMI generic source context data. Pointer to FDMI source instance is passed to FDMI source dock and saved in a list. In its turn, FDMI source dock provides back to FDMI source instance an interface function to perform FDMI record posting. FDMI generic source context stores the following:

- FDMI record type
- FDMI generic source interface
- FDMI source dock interface

fdmi source implementation guideline

FDMI source implementation depends on data domain. Specific FDMI source type stores

- FDMI generic source interface
- FDMI specific source context data (source private data)

For the moment FDMI FOL source is implemented as the 1st (and currently the only) FDMI source. FDMI FOL source provides ability for detailed FOL data analysis. Based on generic FOL record knowledge, “test filter condition” function implemented by FOL source checks FOP data: FOL operation code and pointer to FOL record specific data.

For FOL record specific data handling FDMI FOL record type is declared and registered for each specific FOL record type (example: write operation FOL record, set attributes FOL record, etc.)

FDMI FOL record type context stores the following:

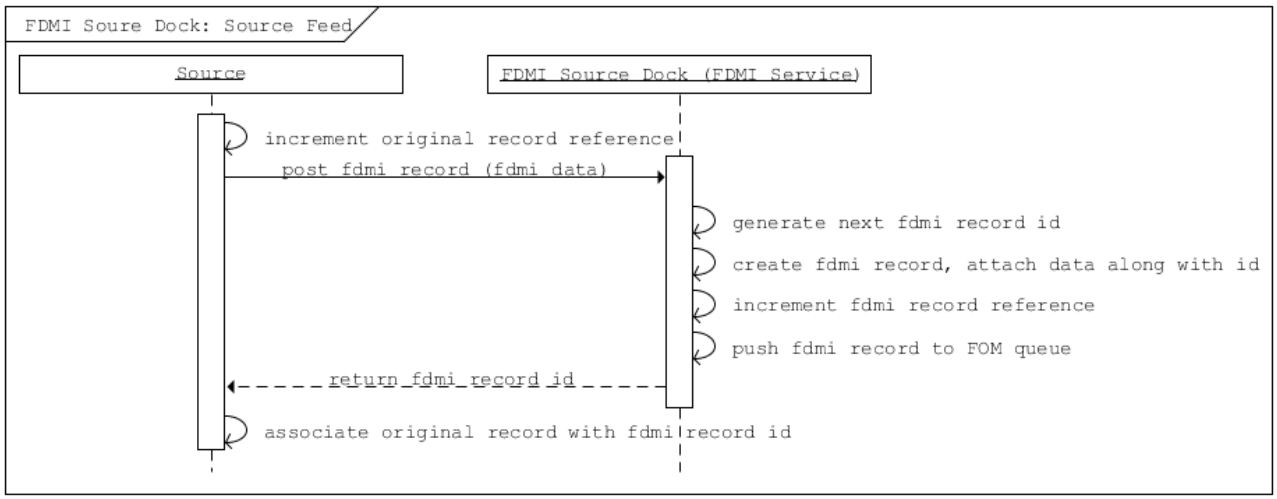
- FOL operation code
- FOL record interface

FOL record interface functions are aware of particular FOL record structure and provide basic primitives to access data:

- Check condition

On FDMI FOL record type FDMI record registration all its internals are initialized and saved as FDMI FOL record context data. Pointer to FDMI FOL record type is stored as a list in FDMI specific source context data.

fdmi record post



Source starts with local locking data to be fed to FDMI interface, then it calls posting FDMI API. On FDMI side a new FDMI record (data container) is created along with new record id, and posted data gets packed into the record. The record is queued for further processing to FDMI FOM queue, and FDMI record id is returned to Source.

To be able to process further calling back from FDMI regarding particular data, i.e. original record, Source is responsible for establishing unambiguous relation between returned FDMI record ID and original record identifier, whatever it look like.

NB:

Please note, the Source is responsible for initial record locking (incrementing ref counter), but FDMI is responsible for further record release.

fdmi source dock fom

FDMI source dock FOM implements the main control flow for FDMI source dock:

- Takes out posted FDMI records
- Examines filters
- Sends notifications to FDMI plugins
- Analyzes FDMI plugin responses

Normal workflow

FDMI source dock FOM remains in an idle state if no FDMI record is posted (FDMI record queue is empty). If any FDMI record is posted, the FOM switches to busy state, takes out FDMI record from a queue and starts analysis.

Before examining against all the filters, FOM requests filter list from filterc. On getting filter list, FOM iterates throw the filter list and examines filter one by one. If filters number is quite big, possible option is to limit a number of filters examined in one FOM invocation to avoid task blocking.

To examine a filter, FOM builds filter execution plan. Filter execution plan is a tree structure, with expressions specified in its nodes.

Each expression is described by elementary operation to execute and one or two operands. Operand may be a constant value, already calculated result of previous condition check or FDMI record specific field value.

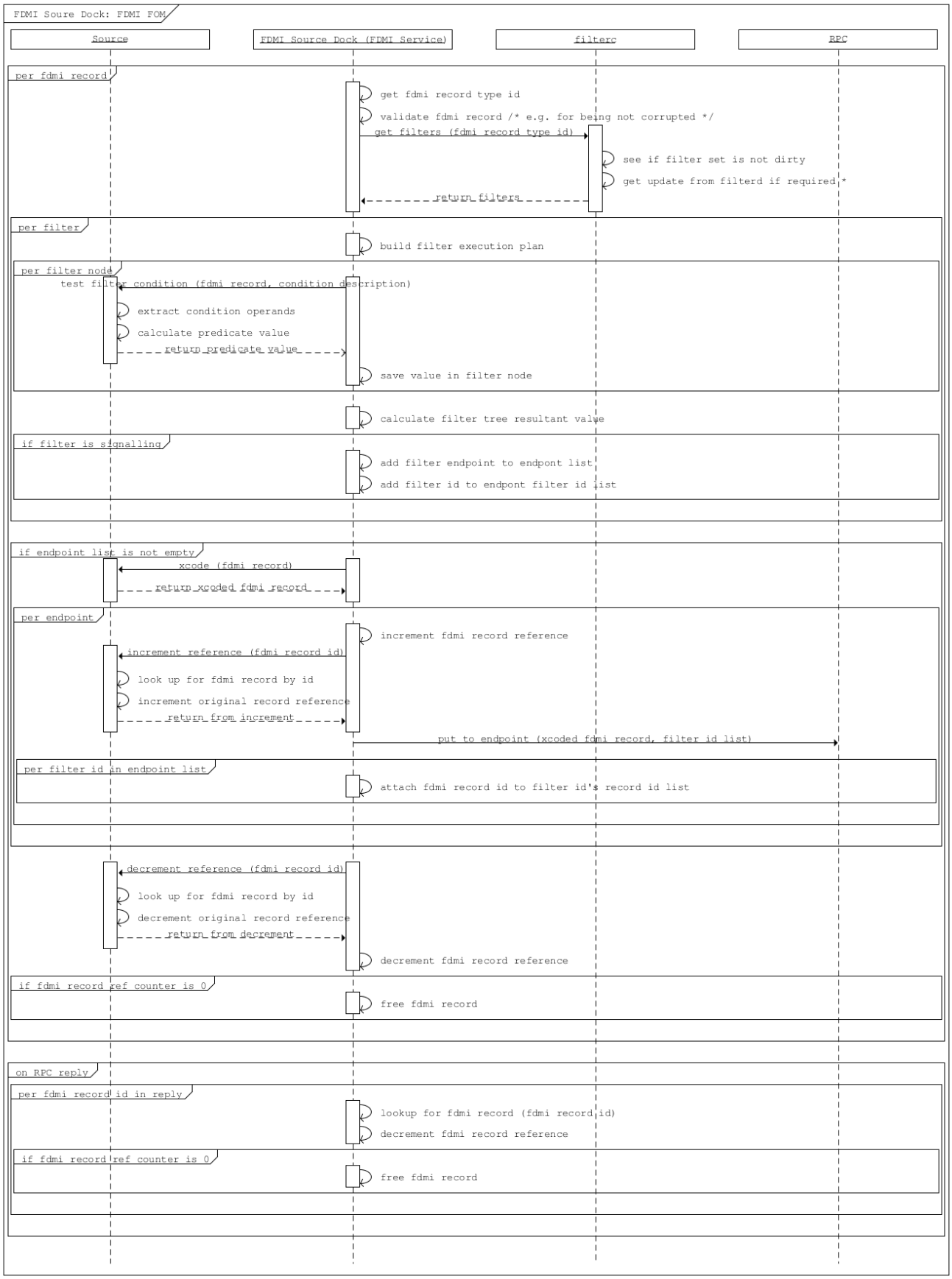
FDMI source dock calculates all expressions by itself. If some of the operands is FDMI record specific field value, then source dock executes callback provided by FDMI source to get operand value.

Also, operations supported during filter execution by FDMI source dock can be extended. So FDMI source can add new operation codes and corresponding handlers in order to support processing data types that are specific to FDMI source. Operation overloading is not supported, so if FDMI source want to define multiplication for some “non-standard” type, it should add new operation and handler for that operation.

If no filter shows a match for a FDMI record, the record is released. To inform FDMI source that this record is no more needed for FDMI system, FDMI generic source interface function “decrease record reference counter” is used.

If one or more filters match the FDMI record, the record is scheduled to be sent to a particular FDMI node(s). If several filters matched, the following operations are performed to optimize data flow:

- Send FDMI record only once to a particular FDMI node (filter provides RCP endpoint to communicate with)
- Specify a list of matched filters, include only filters that are related to the node
- On receipt, FDMI plugin dock is responsible for dispatching received FDMI records and pass it to plugins according to specified matched filters list


FDMI Soure Dock: FDMI FOM
Source
FDMI Source Dock (FDMI Service)
filterc
RPC
per fdmi record
get fdmi record type id
validate fdmi record /* e.g. for being not corrupted */
get filters (fdmi record type id)
see if filter set is not dirty
get update from filterd if required *
return filters
per filter
build filter execution plan
per filter node
test filter condition (fdmi record, condition description)
extract condition operands
calculate predicate value
return predicate value
save value in filter node
calculate filter tree resultant value
if filter is signalling
add filter endpoint to endpont list
add filter id to endpont filter id list
if endpoint list is not empty
xcode (fdmi record)
return xcoded fdmi record
per endpoint
increment fdmi record reference
increment reference (fdmi record id)
look up for fdmi record by id
increment original record reference
return from increment
put to endpoint (xcoded fdmi record, filter id list)
per filter id in endpoint list
attach fdmi record id to filter id's record id list
decrement reference (fdmi record id)
look up for fdmi record by id
decrement original record reference
return from decrement
decrement fdmi record reference
if fdmi record ref counter is 0
free fdmi record
on RPC reply
per fdmi record id in reply
lookup for fdmi record (fdmi record id)
decrement fdmi record reference
if fdmi record ref counter is 0
free fdmi record

In order to manage FDMI records i/o operations, the following information should be stored as FDMI source dock context information:

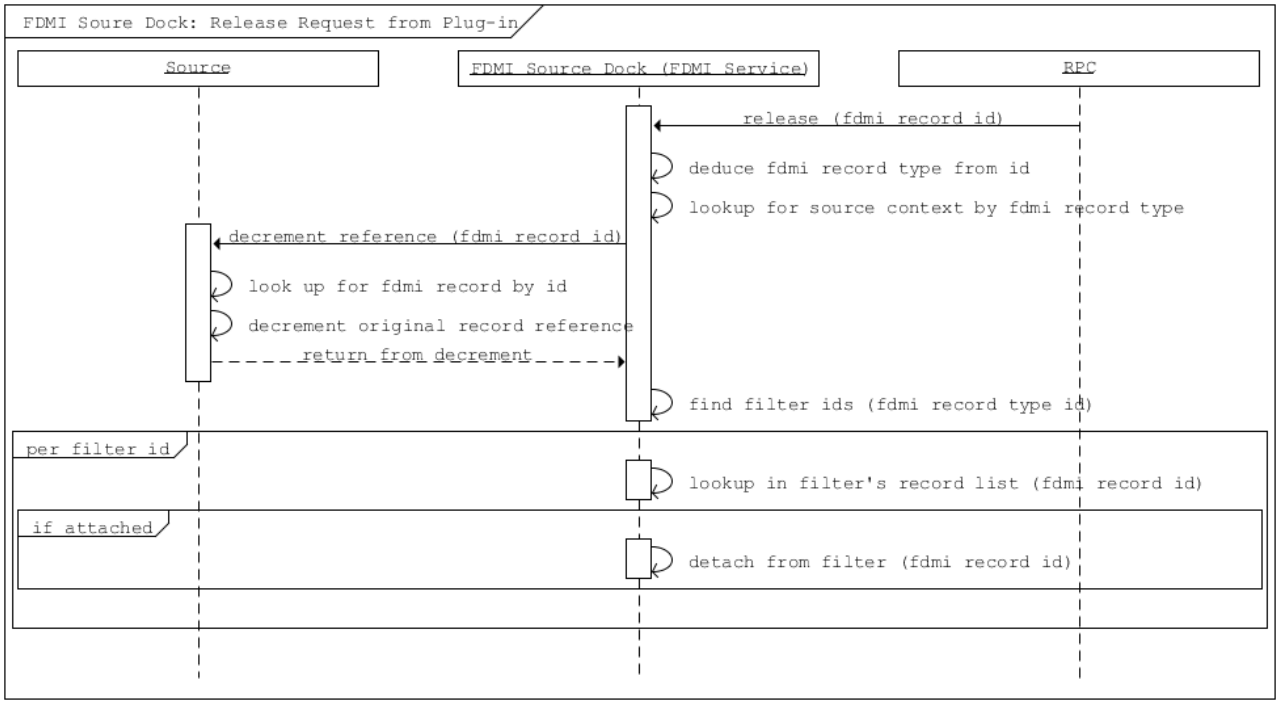
- Sent FDMI record is stored in a FDMI source dock communication context
- Relation between destination Filter Id and FDMI record id being sent to the specified Filter ID
    - Map <Filter Id, FDMI record id> may be used in this case
    - This information is needed to handle Corner case “Mero instance running “FDMI plugin dock” death” – see below.

FDMI record being sent is serialized using FDMI generic source interface function “Xcode functions”

On sending FDMI record its reference counter is increased: FDMI generic source interface function “increase record reference counter” is used.

FDMI source dock increments internal FDMI record reference counter for the FDMI record being sent for each send operation.

On FDMI record receipt, FDMI plugin dock should answer with a reply that is understood as a data delivery acknowledgement. The data acknowledgment should be sent as soon as possible – no blocking operations are allowed.



On receiving data acknowledgement internal FDMI record reference counter for the FDMI record is decremented. If internal reference counter becomes 0, FDMI record is removed from the FDMI source dock communication context.

After FDMI record is handled by all involved plugins, FDMI plugin dock should send FDMI record release request to the FDMI record originator (FDMI source dock). On receiving this request, FDMI source dock removes appropriate pair <Filter Id, FDMI record id> from its context and informs FDMI source that the record is released. FDMI generic source interface function “decrease record reference counter” is used for this purpose. If FDMI source reference counter for a particular FDMI record becomes 0, FDMI source may release this FDMI record.

NOTE: What value should be returned if “Test filter condition” cannot calculate particular filter? “record mismatch” (legal ret code) or “some error ret code”?

Filters set support

FilterC id responsible for storing local copy of filters database and supporting its consistency. By request FilterC returns a set of filters, related to specified FDMI record type. Filter set request/response operation is simple and easy to execute, because a pointer to a local storage is returned. It allows FDMI source dock to re-request filter set from FilterC every time it needs it without any resources over usage. No any additional actions should be done by FDMI source dock to maintain filter set consistency.

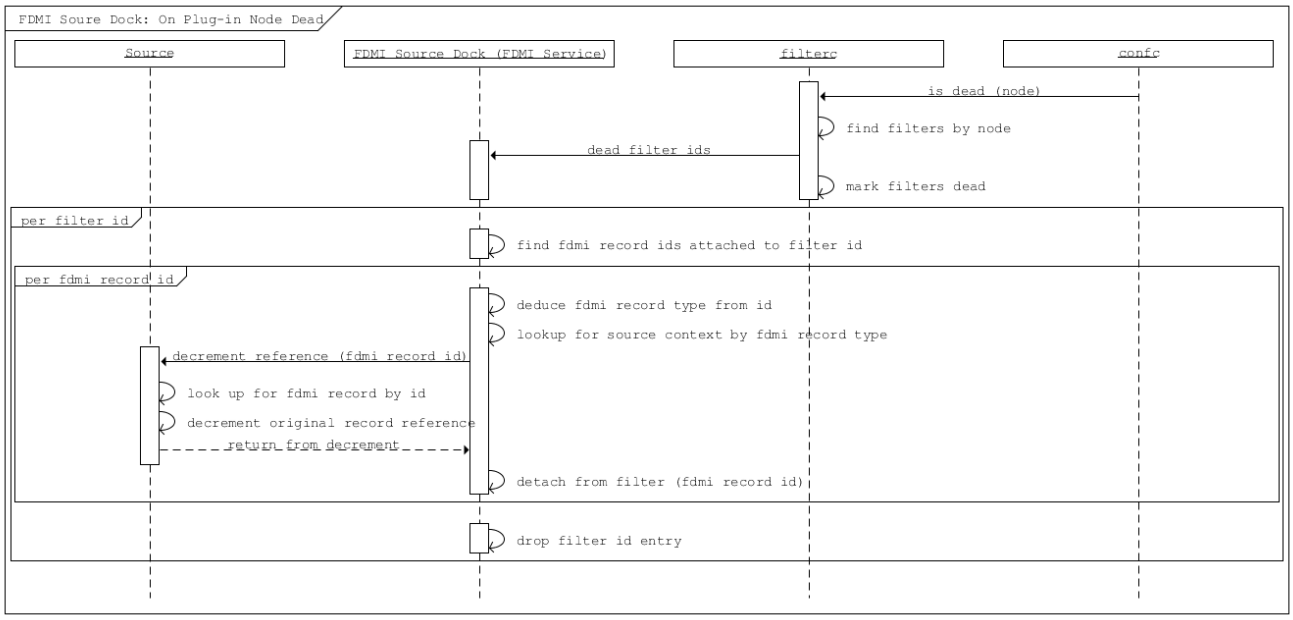
Corner cases

Special handling should be applied for the following corner cases:

- Mero instance running “FDMI plugin dock” death
- FDMI filter is disabled

Mero instance running “FDMI plugin dock” death may cause 2 cases:

- RPC error while sending FDMI record to a FDMI source dock. No data acknowledgement received.
- No “FDMI record release” request is received from FDMI plugin dock



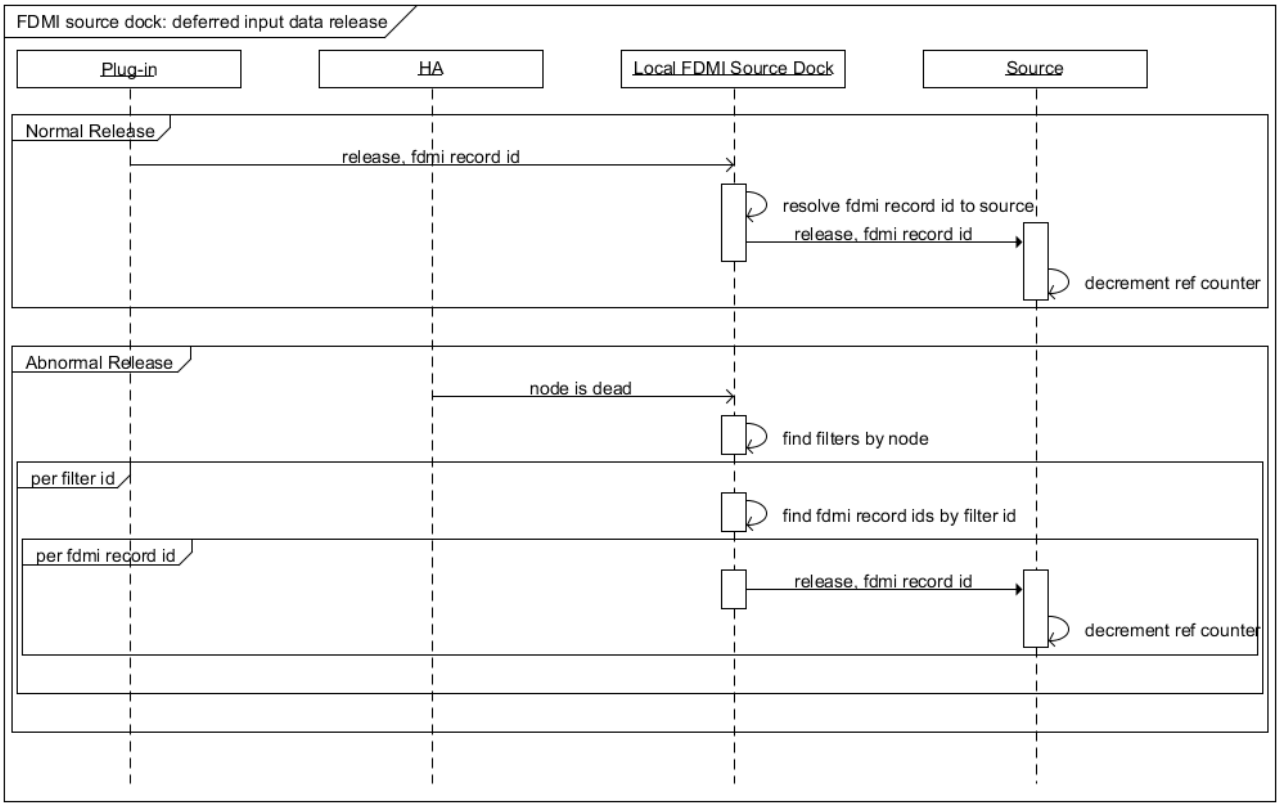
If RPC error while sending FDMI record to a FDMI source dock appears, FDMI source dock should decrement internal FDMI record reference counter and FDMI Source specific reference counter, following general logic described above. In this case all the FDMI record context information is stored in the communication context; it makes it obvious how to fill in parameters for interface functions calls.

“FDMI record release” request is not received from FDMI plugin dock case is not detected by FDMI source dock explicitly. This case may cause storing FDMI records on source during unpredictable time period (depends on FDMI source domain: it may store FDMI records permanently until receiving from Plugin confirmation on FDMI record handling). Possible ways to escape the described issue:

- Based on some internal criteria, FDMI source resets reference counter information and re-posts FDMI record
- FDMI source dock receives notification on a death of the node running FDMI plugin dock. Notification is sent by HA.

In the latter case FDMI source dock should inform FDMI source to release all the FDMI records that were sent to plugins that were hosted on the dead node. In order to do it, context information stored as relation between destination Filter Id and FDMI record id <Filter Id, FDMI record id > is used: all filters related to the dead node may be determined by EP address. The same handling that

is done for “FDMI record release request” should be done in this case for all the FDMI records, bound to the specified filters id.



FDMI filter may be disabled by plugin itself or by some 3rd parties (administrator, HA, etc.). On filter state change (disabling the filter) a signal is sent to FDMI source dock. Upon receiving this signal, FDMI source dock iterates through the stored map <Filter Id, FDMI record id> and check each filter status. If a filter status is found to be disabled, the same handling that is done for “FDMI record release request” should be done for all the FDMI records, bound to the specified filter id.

FilterD

FDMI plugin creates filter in order to specify criteria for FDMI records it is interested in. FDMI filter service (filterD) maintains central database of the FDMI filters existing in Mero cluster. There is only one (possibly duplicated) mero instance with filterD service in the whole Mero cluster. FilterD provides to users read/write access to its database via RPC requests.

FilterD service is started as a part of chosen for this purpose mero instance. Address of filterD service endpoint is stored in confd database. FilterD database is empty after startup.

FilterD database is protected by distributed read/write lock. When filterD database should be changed, filterD service acquires exclusive write lock from Resource Manager (RM), thus invalidating all read locks held by database readers. This mechanism is used to notify readers about filterD database changes, forcing them to re-read database content afterwards.
There are two types of filterD users:

- FDMI plugin dock
- FDMI filter client (filterC)

FDMI filter description stored in database contains following fields:

- Filter ID
- Filter conditions stored in serialized form
- RPC endpoint of the FDMI plugin dock that registered a filter
- Node on which FDMI plugin dock that registered a filter is running

FDMI plugin dock can issue following requests:

- Add filter with provided description
- Remove filter by filter ID
- Activate filter by filter ID
- Deactivate filter by filter ID
- Remove all filters by FDMI plugin dock RPC endpoint
  Also there are other events that cause some filters deactivation in database:
- HA notification about node death
  Filters stored in database are grouped by FDMI record type ID they are intended for.
  FilterD clients can issue following queries to filterD:
- Get all FDMI record type ID's known to filterD
- Get all FDMI filters registered for specific FDMI record type ID

NB:
Initial implementation of filterD will be based on confd. Actually, two types of conf objects will be added to confd database: directory of FDMI record types IDs and directory of filter descriptions for specific FDMI record type ID.

This implementation makes handling of HA notifications on filterD impossible, because confd doesn't track HA statuses for conf objects.

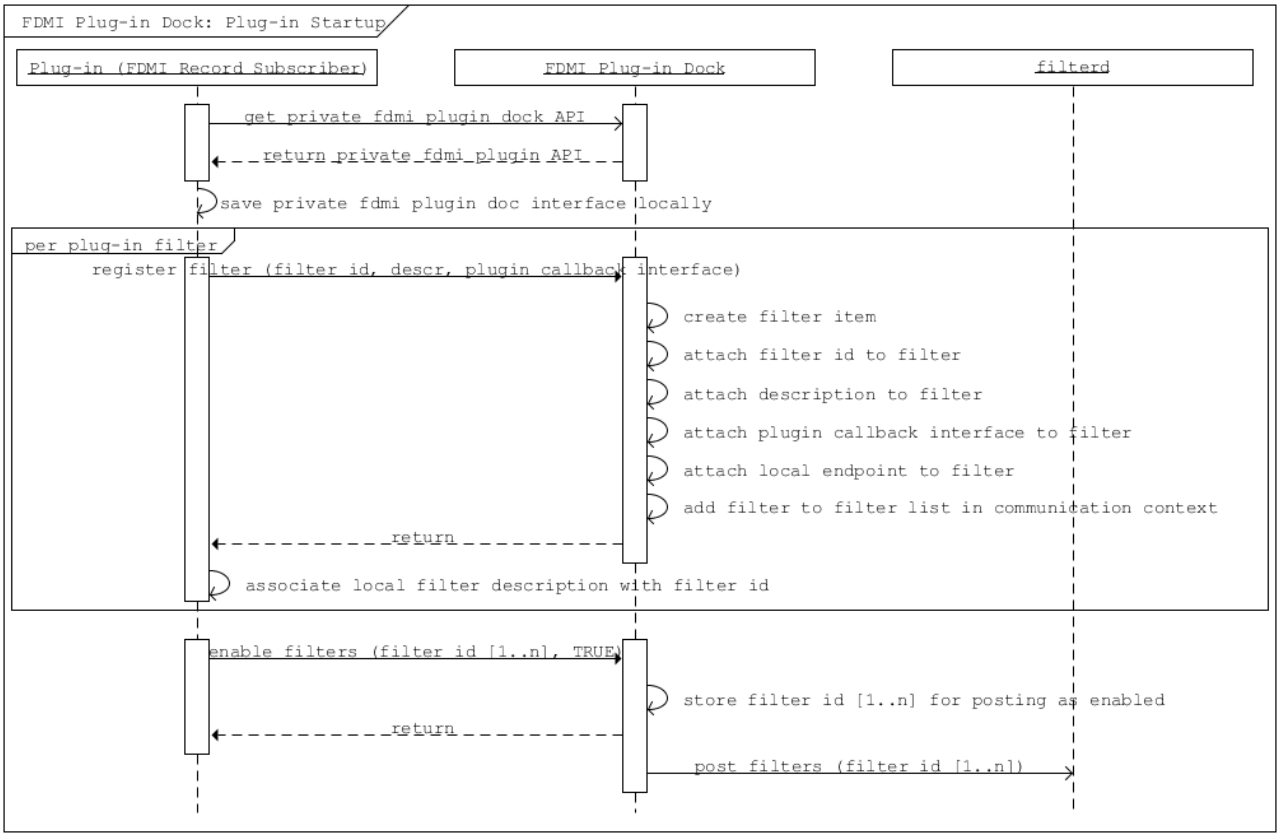
FilterC

FilterC is a part of Mero instance that caches locally filters obtained from filterD. FilterC is initialized by FDMI source dock service at its startup.
Also, filterC have a channel in its context which is signaled when some filter state is changed from enabled to disabled.

FilterC achieves local cache consistency with filterD database content by using distributed read/write lock mechanism. FilterD database change is the only reason for filterC local cache update. HA notifications about filter or node death are ignored by filterC.

NB:
Initial implementation of filterC will be based on confc. So confc will cache filter descriptions locally. In that case implementation of filterC channel for signaling disabled filters is quite problematic.

fdmi plugin registration



- Filter id:
  - Expected to be 128 bits long
  - Filter id is provided by plug-in creator
  - Providing filter id uniqueness is a responsibility of plug-in creator
  - Filter id may reuse m0_fid structure declaration

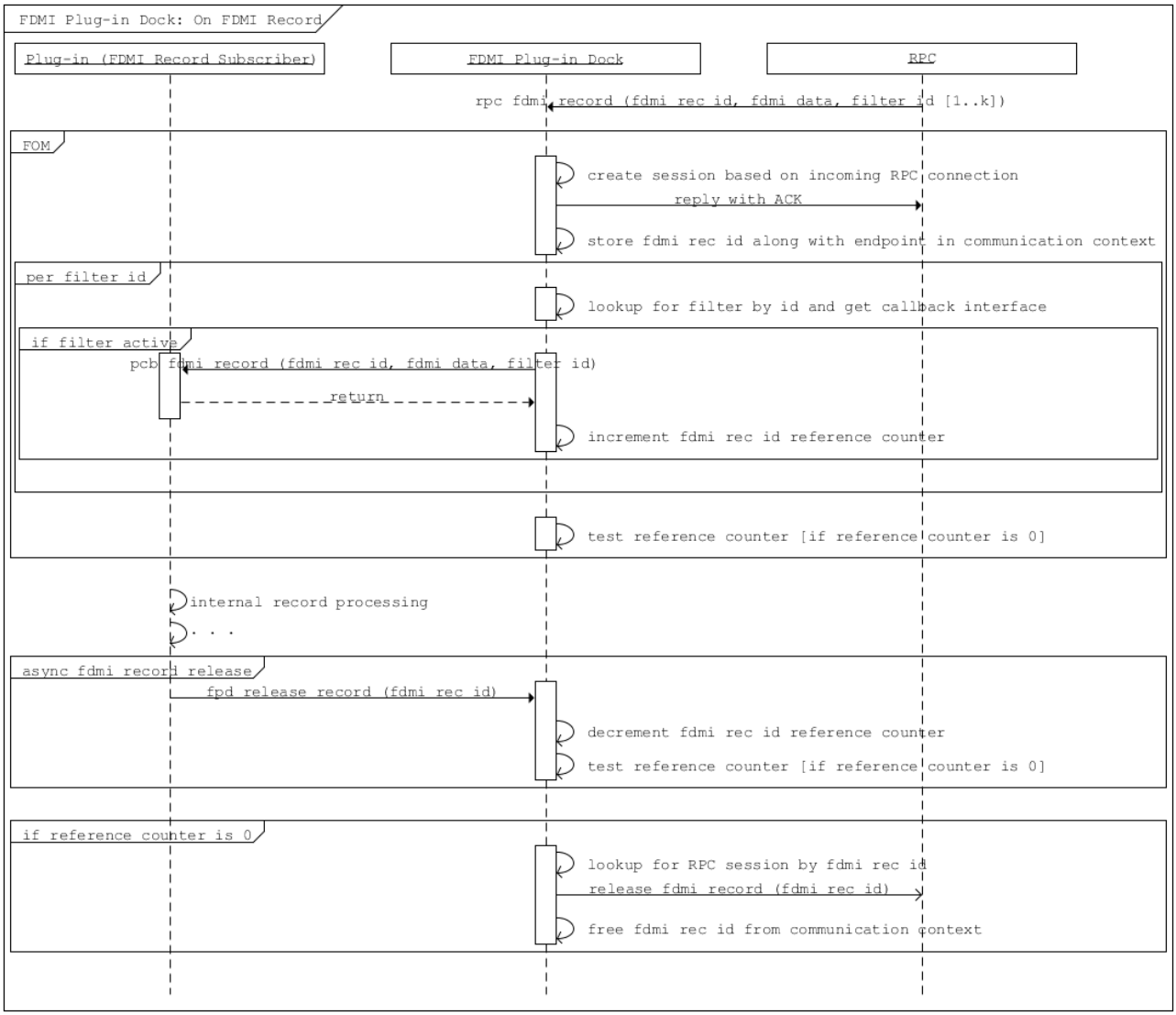
TBD:

A possible situation is pluging being notified with filter id that is already announced inactive, which change did not reach the source to the moment of emitting notification. Should the id be passed to plugin by FDMI?

Another thing to consider on: what should be done by FDMI in case filter id arrived in notification is unknown to the node, i.e. no match to any locally registered filter rule encountered?

A complimentary case occurs when plugin was just fed with FDMI record and did not instructed FDMI to release the one yet. Instead, it declares the corresponding filter instance to be de-activated. Current approach implies that plug-in is responsible for proper issuing release commands once it was fed with FDMI record, disregarding filter activation aspect.

fdmi plugin dock fom



Received FDMI record goes directly to Plugin Dock's FOM. At this time a new session re-using incoming RPC connection needs to be created and stored in communication context being associated with **fdmi record id**. Immediately at this step RPC reply is sent confirming fdmi record delivery.

Per filter id, corresponding plug-in is called feeding it with fdmi data, fdmi record id and filter id specific to the plug-in. Every successful plug-in feed results in incrementing **fdmi record id** reference counter. When done with the ids, FOM needs to test if at least a single feed succeeded. In case it was no success, i.e. there was not a single active filter encountered, or plug-ins never confirmed fdmi record acceptance, the fdmi record has to be released immediately.

Plug-in decides on its own when to report fdmi original record to be released by Source. It calls Plug-in Dock about releasing particular record identified by fdmi record id. In context of the call fdmi record reference counter is decremented locally, and in case the reference counter gets to 0, the corresponding Source is called via RPC to release the record (see Normal workflow, FDMI Source Dock: Release Request from Plug-in).

fdmi plugin implementation guideline

The main logic behind making use of a FDMI plug-in is a subscription to some events in sources that comply with conditions described in filters that plug-in registers at its start. In case some source record matches with at least one filter, the source-originated record is routed to corresponding plug-in.

Plug-in responsibilities

During standard initialization workflow plug-in:

- Obtains private Plug-in Dock callback interface
- Registers set of filters, where filter definition:
    - Identifies FDMI record type to be watched
    - Provides plug-in callback interface
    - Provides description of condition(s) the source record to meet to invoke notification

NB:
Condition description syntax must follow source functionality completely and unambiguously. Source of the type specified by filter description must understand every elementary construct of condition definition. This way the evolution of filter definition syntax is going to be driven by evolution of source functionality.

NB:
Source is responsible for validation of filter definition. This may result in deactivating filters that violate syntax rules the particular source supports. The facts of syntax violation ideally must become known some way to Mero cloud admin staff.

- Starts subscription by activating registered filters. Opaquely for the plug-in the filter set is propagated among Mero nodes running FDMI Source Dock role which enables source record filtering and notifications.

During active subscription workflow looks like following:

- Plug-in is called back with:
  - FDMI record id
  - FDMI record data
  - Filter id indicating the filter that signaled during the original source record processing
- Plug-in must keep trace of FDMI record (identified by FDMI record id globally unique across the Mero cloud) during its internal processing.
- Plug-in must return from the callback as quick as possible to not block other callback interfaces from being called. Plug-in writers must take into account the fact that several plug-ins may be registered simultaneously, and therefore, must do their best to provide smooth cooperation among those.
- However plug-in is allowed to take as much time as required for FDMI record processing. During the entire processing the FDMI record remains locked in its source.
- When done with the record, plug-in is responsible for the record release.
- Plug-in is allowed to activate/deactivate any subset of its registered filters. The decision making is entirely on plug-in's side.
- The same way plug-in is allowed to de-register and quit any time it wants. The decision making is again entirely on plug-in's side. After de-registering itself the plug-in is not allowed to call private FDMI Plug-in Dock in part of filter activation/deactivation as well as fdmi record releasing. The said actions become available only after registering filter set another time.

Implementation Plan

Phase 1

1) Implement FDMI service
2) FDMI source dock
    a) FDMI source dock API
    b) Generic FDMI records handling (check against filters, send matched records (only one recipient is supported))
    c) Handle FDMI records deferred release
3) FOL as FDMI source support
    a) Generic FDMI source support
    b) Limited FOL data analysis (operation code only)
4) Filters
    a) Simplified filters storing and propagation (use confd, confc)
    b) Static filter configuration
    c) Limited filtering operation set
    d) Generic filters execution
5) FDMI plugin dock
    a) FDMI plugin dock API
    b) Generic FDMI records handling (receive records, pass records to target filter)
6) Sample echo plugin

Backlog

1) Filters
    a) FilterD, FilterC
    b) Full filtering operation set
    c) Register/deregister filter command
    d) Enable/disable filter command
    e) Filter sanity check
    f) Query language to describe filters
2) FDMI Source dock
    a) Multiple localities support
    b) Filters validation
    c) FDMI kernel mode support
    d) Support several concurrent RPC connections to clients (FDMI plugin docks)
3) FDMI Plugin dock
    a) Filter management (register/enable/disable)

4) HA support (node/filter is dead) in
    a) FDMI source dock
    b) FDMI plugin dock
    c) Filters subsystem
5) FOL as FDMI source support
    a) FOL data full analysis support
    b) Transactions support (rollback/roll-forward)
6) ADDB diagnostic support in both FDMI source dock and plugin dock
7) ADDB as FDMI source support