

HLD of distributed transaction manager

by Nikita Danilov <nikita_danilov@xyratex.com>

Date: 2012/12/17

Revision: 1.0

a monstrous extension of Angus MacDiarmid's „incoherent transactions“

—Ch. Kinbote

This document presents a high level design (HLD) of the distributed transaction manager (DTM), which is a core component of Mero.

The main purposes of this document are: (i) to be inspected by Mero architects and peer designers to ascertain that high level design is aligned with Mero architecture and other designs, and contains no defects, (ii) to be a source of material for Active Reviews of Intermediate Design (ARID) and detailed level design (DLD) of the same component, (iii) to serve as a design reference document.

The intended audience of this document consists of Mero customers, architects, designers and developers.

Introduction

Distributed transactions are groups of storage operations that are atomic in the face of certain failures. Distributed transaction manager (DTM) implements interfaces to create and control distributed transactions.

This document describes a very general DTM able to efficiently handle a variety of system configurations and work-loads. A [separate document](#) describes a staged implementation plan.

[The following color marking is used in this document: [incomplete or todo item](#), [possible design extension or future directions](#).]

Definitions

[Definitions of terms and concepts used by the design go here. The definitions must be as precise as possible. References to the [Mero Glossary](#) are permitted and

encouraged. Agreed upon terminology should be incorporated in the glossary.]

DTM: distributed transaction manager, which provides interfaces to manipulate distributed transactions. DTM consists of *DTM instances* running as part of every Mero instance. A DTM instance maintains information about distributed transactions, receives calls from DTM users, adds DTM information to user messages sent over network, uses FOL to store information about executed operation and changes in distributed transaction status. In addition to placing DTM information to user messages, DTM instances exchange DTM-private messages over network.

Action: a request to change or inspect Mero data or meta-data. Actions originate from applications using external Mero interfaces, Mero extension plugins or Mero itself. Examples of actions are:

- scatter-gather write of data into a Mero object (file);
- scatter-gather read of data from a Mero object;
- creation of a new sub-directory;
- renaming a file;
- write of a data unit reconstructed from parity blocks in a spare unit.

An action is an instance of an *action type*. The action is fully identified by its type and a set of parameters. The set of parameters depends on the action's type. For example, the action to create a new sub-directory is fully identified by

- a unique identifier of the parent directory;
- a name of the new sub-directory;
- attributes of the new sub-directory.

An action is executed (see operation and query below). The replies from action's execution constitute *action result*.

Persistent store: is a store that is guaranteed to survive any allowed failure (see [below](#)). Persistent store is attached to some, but not necessarily all Mero instances.

Persistency: an entity (operation, update, object, object version, distributed transaction, &c., see below) is *persistent* on a certain Mero instance, when a record, describing this entity, is written to the persistent store, attached to this Mero instance. The format of the record and method of its writing depends on the entity

in question. For example, for operations and updates, FOL records are used; for objects, records in meta-data tables are used.

Stability: an entity is stable when DTM guarantees that the entity will remain accessible after any allowed failure (see [below](#)).

Operation: an action that changes system state. As far as DTM is concerned, the life-cycle of an operation is as following:

0. an operation originates in a Mero instance;
1. the operation is executed locally in the Mero instance;
2. changes to the local state are cached for some time;
3. the changes are re-integrated (see [below](#)) to a number of remote Mero instances;
4. remote instance executes the corresponding parts of the operation (called updates, see [below](#));
5. remote instance replies back to the originator;
6. eventually, the updates (see [below](#)) of the operation become stable, DTM optionally notifies the originator about this;
7. eventually, the operation becomes stable, DTM optionally notifies the originator about this.

Step (1) is not present for so-called intent operations, step (2) is not present for non-cached operations, steps (3) through (5) are not present when the operation is local.

Re-integration: a process of requesting remote Mero instances to execute an operation. A single operation can require re-integration to multiple remote Mero instances. Part of an operation that a remote Mero instance must execute is called an *update*. For example, a single write to a parity de-clustered file requires multiple updates to data and parity units hosted of different ioservices.

Query: an action that doesn't change system state. Query's life-cycle is the following:

0. a query originates in a Mero instance;
1. if the query can be satisfied locally, it is;
2. otherwise query is sent to a number of remote Mero instances;
3. remote instance execute the corresponding parts of the query;
4. remote instance replies back to the originator.

Allowed failure: is a failure that doesn't affect DTM transactional guarantees. The following failures are allowed:

- transient network failures (message loss, message corruption, message duplication, message re-ordering);
- node crash and restart, when Mero instance re-starts with the contents of volatile store lost and the contents of persistent store, if any, intact.

DTM doesn't deal with other failures, such as permanent loss of persistent store or permanent loss of connectivity. It is possible that a stable entity is lost due to such ("unallowed") failure. Other mechanisms provide recovery for these types of failures.

Distributed transaction: a group of operations that must survive allowed failures together. A DTM user creates a distributed transaction and populates it with operations (somewhere between the steps 0 and 3 in the operation life-cycle). The user then explicitly closes the distributed transaction, indicating that no further operations will be added to it. DTM will notify the user when the transaction becomes stable.

Local transaction: a group of local updates that must become persistent together. DTM uses local transactions to implement distributed transactions. A DTM instance uses local transactions to modify persistent store attached to a Mero instance.

Consistency: an entity (for example, an object, an [application domain](#), the whole cluster) is consistent if result of any action on the entity corresponds to a sequence of complete distributed transactions executed against the entity. In other words, an entity is inconsistent when its state is a result of partial execution of a distributed transaction.

Recovery: a process of restoring system consistency after an allowed failure. Recovery is carried out by cooperating DTM instances.

2. Requirements

[This section enumerates requirements collected and reviewed at the Requirements Analysis (RA) and Requirements Inspection (RI) phases of

development. References to the appropriate RA and RI documents should go here. In addition this section lists architecture level requirements for the component from the Summary requirements table and appropriate architecture documentation.]

Requirements from the [Mero Summary Requirements Table](#), see [Mero Detailed QAS](#) for more information:

- **[R.M0.DTX]** distributed transactions (dtx) are supported.
- **[R.M0.DTX.APPS]** storage application (FDML extension plugin) activity is executed as a part of the same dtx as the original file system operation. This guarantees that if file system operation is later rolled back due to a failure, storage application state is also rolled back, keeping the latter consistent with the core.

An FDML plugin is an extension to the core Mero functionality that asynchronously consumes FOL records produced by Mero instances. As an example, consider a replication plugin that consumes FOL records related to operations on a given set of files and re-executes these operations on a different file system. Replication plugin would typically run as a service on a separate node with network access to both source and target file systems. FOL records are delivered to a plugin asynchronously, that is, after the operation reply is sent to the user. **R.M0.DTX.APPS** means that plugin activity, executed asynchronously is part of original distributed transaction in context of which FOL records were produced.

A plugin receives FOL records to which the plugin subscribed through an *update stream* (explained elsewhere). Each FOL record carries with it the original transaction context. The plugin executes its operations in this context. As a result, if the original distributed transaction is undone for whatever reason (e.g., in case of the replication plugin because of a failure in the source file system), the operations that the plugin executed when processing the FOL record from that transaction are undone too.

- **[R.M0.DTX.ATOM]** Dtx can be used to implement ATtributes On Meta-data node (ATOM).
- **[R.M0.DTX.AVAILABILITY]** Non-blocking availability layer can be based on

dtx.

- **[R.M0.DTX.BACK-ENDS]** a distributed transaction can contain updates for state on multiple back-ends both local and remote.
- **[R.M0.DTX.COUPLING]** a degree of coupling can be specified for a distributed transaction.

Coupling between distributed transactions specifies which transactions must be undone if a particular transaction is undone. For example, with client-time coupling, if a transaction is undone, all transactions issued later by the same client must be undone. This level of coupling guarantees that after a recovery the system corresponds to a certain point in time, as measured by the client clock. Similarly, with thread-time coupling, if a transaction is undone, all transactions issued later by the same thread must be undone.

- **[R.M0.DTX.FSYNC]** A dtx stabilization can be forced.
- **[R.M0.DTX.GROUP]** a collection of file system updates can be grouped into a distributed transaction.
- **[R.M0.DTX.ISOLATION]** a degree of isolation can be specified for a distributed transaction.

Dual to coupling, the degree of isolation specifies which transaction must *not* be undone if a particular transaction is undone. For example, commit-on-client-share isolation level prescribes that before an object can be involved in a distributed transaction, all updates executed against this object by earlier transactions issued by different clients must become persistent. This guarantees that undo of a transaction won't affect transactions made by different clients. commit-on-thread-share does similar thing on thread level.

- **[R.M0.DTX.MULTIPLE]** a distributed transaction can contain updates for multiple objects both data and meta-data.
- **[R.M0.DTX.SERIALIZABLE]** Dtxs are "weakly serializable": all conflicting updates from a pair of dtxs are applied in the same order.

Suppose transactions T and S update a pair of shared objects A and B (each

transaction can update other objects). Corresponding updates from T are TA and TB, and from S: SA and SB. Weak serializability requirement means that either

- TA is executed before SA and TB is executed before SB, or
- SA is executed before TA and SB is executed before TB.

In the first case "T is before S" and in the latter case "S is before T". This requirement makes it possible to define a [partial order between transaction](#). Compare with the notion of [serializability](#) for data-base transactions.

- **[R.M0.DTX.VARIABILITY]** DTM transparently takes advantage of new facilities (e.g., of a new fast logging device) or adapts when facilities are removed.

Additional requirements:

- **[r.m0.dtx.clovis]** [clovis](#) interface can be implemented on top of DTM.
- **[r.m0.dtx.large]**: a transaction can include "large" operations in the sense that it is *not* permissible for DTM to duplicate operation descriptor (containing enough information to undo or redo the operation) and to send every descriptor in the transaction to every DTM instance participating in the transaction.

An example of large operation is file write, where multiple component objects stored on different services have to be updated. In this case each service receives updates only for its part of data and not the entire data written by the operation.

Standard scenarios

This sub-section discusses usage patterns that DTM must support. For simplicity, the descriptions below identify a DTM instance with the Mero instance the DTM instance runs in. That is, one talks about a DTM instance with persistent store or a DTM instance hosting an object.

scenario name	[r.m0.dtx.local-operation]
---------------	-----------------------------------

context	A DTM instance with persistent store. A user of the same Mero instance.
user action	a sequence of one of more operations on persistent Mero objects
requirements	the sequence must be atomic w.r.t. DTM instance failures
interaction	<ul style="list-style-type: none"> the user explicitly adds the operation descriptors to a transaction; operation descriptors provide redo and undo operations; DTM invokes user provided call-back when the transaction becomes stable

scenario name	[r.m0.dtx.local-distributed]
context	A DTM instance S with persistent store. A DTM instance C. A user of C.
user action	a sequence of one of more operations on persistent objects managed by S
requirements	the sequence must be atomic w.r.t. S failures, C failures and network failures
interaction	<ul style="list-style-type: none"> the user calls C to open a distributed transaction; the user adds operation descriptors to the transaction; the user re-integrated the operations; the user closes the transaction; S notifies the user (<i>via</i> C) when the transaction becomes stable

In local-distributed scenario a client makes multiple updates to objects on a single remote service. This scenario is used in Mero-2013 version for meta-data.

scenario name	[r.m0.dtx.application-domain] , see application domain
context	A user of DTM instance C. A set of DTM instances S_i with persistent store. An application domain owned by the user and hosted on S_i .

user action	a sequence of one or more updates to persistent Mero objects in the domain
requirements	the sequence must be atomic w.r.t. S_i failures, C failures and network failures
interaction	<ul style="list-style-type: none"> • the user calls C to open a distributed transaction; • the user adds operation descriptors to the transaction; • the user re-integrated the operations to S_i; • the user closes the transaction; • C accumulates S_i's notifications about the transaction status and notifies the user when the transaction becomes stable

scenario name	[r.m0.dtx.general]
context	A user of DTM instance C. A set of DTM instances S_i with persistent store.
user action	a sequence of one or more updates to persistent Mero objects managed by S_i , where the objects can be concurrently queried and updated by other users (subject to RM constraints)
requirements	the sequence must be atomic w.r.t. S_i failures, C failures, failures of other users modifying the objects and network failures
interaction	<ul style="list-style-type: none"> • the user calls C to open a distributed transaction; • the user adds operation descriptors to the transaction; • the user re-integrated the operations to S_i; • the user closes the transaction; • C accumulates S_i's notifications about the transaction status and notifies the user when the transaction becomes stable

scenario name	[r.m0.dtx.pre-established]
context	Multiple users of DTM instances C_j . A set of DTM instances S_i with persistent store.
user action	the users re-integrate sequences of updates to persistent

	Mero objects managed by S_i . The updates are part of the same distributed transaction.
requirements	the sequences must be atomic w.r.t. S_i failures, C_j failures, failures of other users modifying the objects and network failures. The users do not have to communicate with each other in the course of transaction, except as required by the RM.
interaction	<ul style="list-style-type: none"> the user calls C_j to open a distributed transaction, supplying identities of all C_j together with a unique transaction identifier shared by all members of C_j (such identifier should be computable, because members do not communicate); the user adds operation descriptors to the transaction; the user re-integrated the operations to S_i; the user closes the transaction; C_j accumulates S_i's notifications about the transaction status and notifies the user when the transaction becomes stable

The motivating example for the **[r.m0.dtx.pre-established]** requirement is SNS repair, where multiple units in a parity group are re-constructed transactionally.

3. Design highlights

[This section briefly summarises key design decisions that are important for understanding of the functional and logical specifications and enumerates topics that a reader is advised to pay special attention to.]

DTM must support a great variety of operational environments, work-loads and system configurations. During normal operation, DTM records transaction boundaries in the FOLs of Mero instances. After a failure, the recovery process examines FOLs to determine consistent system state that includes only complete transactions. A few observations are critical to this:

- a FOL record must be in the same local transaction as the execution of the update this record describes [**u.M0.FOL ST**];
- sometimes consistency can be restored by undoing updates, sometimes it can be restored by redoing updates [**u.M0.FOL.REDO ST**], [**u.M0.FOL.UNDO ST**];

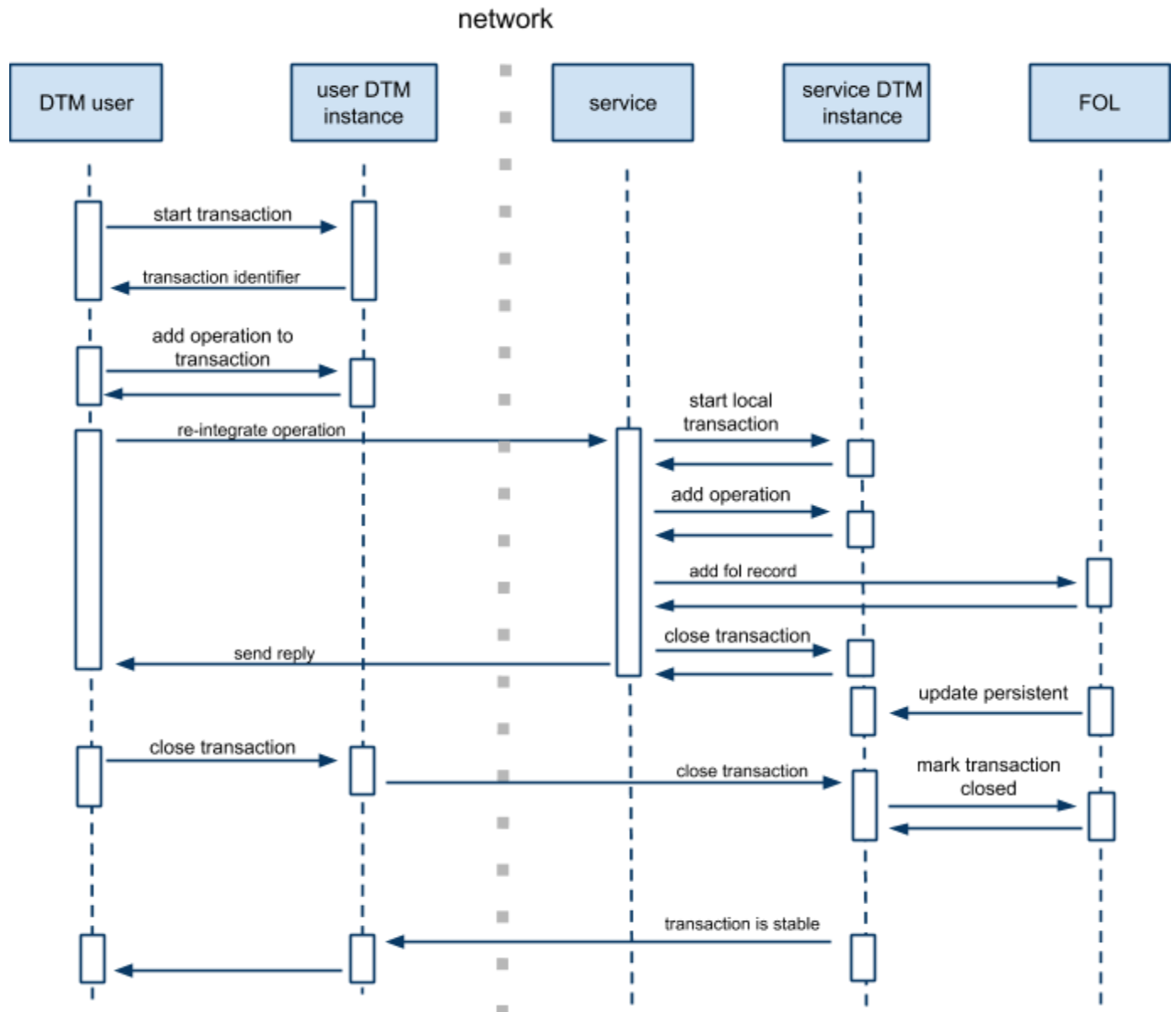
- sometimes an overhead of DTM can be greatly reduced by grouping distributed transaction in large groups and restoring to the boundary of such a group [**u.M0.FOL.EPOCHS** ST];
- a part of the system that is not accessed by applications is consistent (see the definitions of consistency and stability);
- and the last, but not the least (the foremost, in fact): the asynchronous nature of the system and the arbitrary timings of events and failures make rigorous definition of DTM invariants and algorithms the goal of a paramount importance.

4. Functional specification

[This section defines a [functional structure](#) of the designed component: the decomposition showing *what* the component does to address the requirements.]

DTM interacts with

- DTM users (hereinafter simply users), which start transactions and issue operations in the context of transactions. DTM and users interact through the DTM user interface;
- services, which execute updates. One can think about services as Mero request handler services (`m0_reqh_service`), but specifics are not important for the DTM design. DTM and services interact through the DTM service interface;
- FOL. DTM stores additional information in the FOL records generated by the services. On a node failure and restart, DTM scans the FOL of the restarted node to determine the consistent state to recover to.



Sequence diagram of interaction between a DTM user, DTM, DTM service and FOL.

The diagram above shows one possible scenario of interaction between a user starting a distributed transaction containing a single operation and DTM. In the scenario depicted, it is the service DTM instance that determines that the distributed transaction is stable. Depending on the configuration, stability can be determined by various mechanisms, including service DTM instance, client DTM instance or a special distributed stability detection algorithm, see the [Stability detector](#) sub-section for details.

Multiple operations can be added to a transaction and re-integrated concurrently. It

is not, in general, necessary to add all the operations to a transaction before starting re-integration. There are, however, some restrictions:

- when [epoch-based](#) stability detection is used, all operations must be added in the same epoch;
- if an object, updated by a transaction is protected by the resource manager, resource manager lock (usage credit) is released only when all updates for the object are re-integrated.

When a user calls DTM to add an operation to a transaction, DTM records some internal DTM information (transactional context) in the operation data-structure. When the operation is re-integrated, its transactional context is transmitted to the service, transparently to the user. When the operation is executed by the service, operation description and its transactional context are added to the FOL record, that is written to the FOL at the end of operation execution. Operation execution is done as a local transaction. The FOL record is written as part of the same local transaction. Local transaction manager invokes a special call-back when local transaction becomes persistent. The service DTM instance uses this call-back to mark parts of distributed transaction persistent.

Note that DTM neither guarantees nor checks semantical consistency of operations executed by its users. Users should use resource manager (RM) to prevent inconsistent use of objects involved in transactions.

Similarly, DTM doesn't guarantee that services manipulate the objects consistently. The services should use RM or other means of concurrency control for this.

At last, a DTM instance doesn't dictate when re-integration of operations must happen. Updates of the same operation can be re-integrated at different time as determined by a caching policy.

5. Logical specification

[This section defines a logical structure of the designed component: the decomposition showing *how* the functional specification is met. Subcomponents and diagrams of their interrelations should go in this section.]

This long section starts by introducing a lot of new abstract notions and describing

their relationships with minimal examples. The use cases are described later, in the [Use cases](#) section. The material is arranged in this order to describe use cases in terms of the DTM abstractions. If the reader is stuck with the Logical specification, turning to the use cases is advised.

DTM contains sub-modules:

- nucleus: maintains histories of updates and operations. Nucleus deals with changes in persistency;
- stability detector: a mechanism that determines when an update is stable;
- undo closure: a mechanism that determines which updates should be undone during recovery.

The rest of DTM organises the interface between sub-modules and external users.

Nucleus

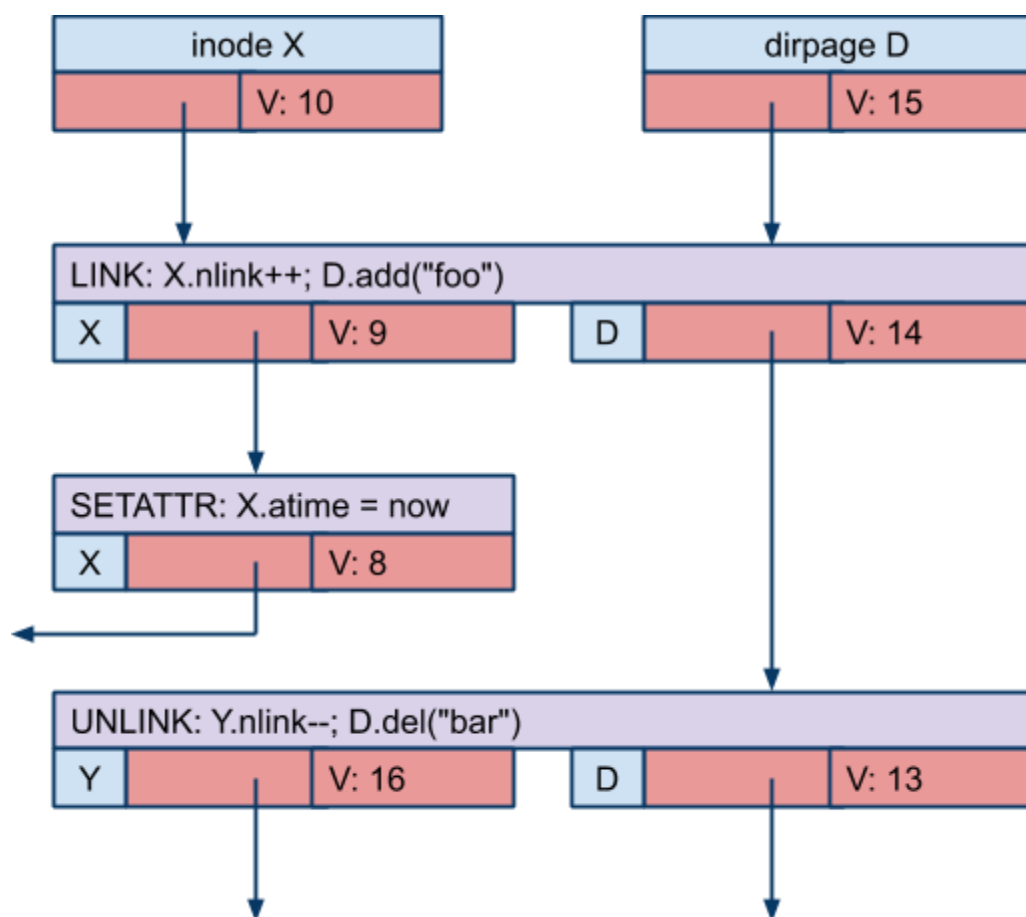
Toward the goal of providing a rigorous definition of DTM algorithms and invariants, a central part of DTM, called nucleus, is defined. The nucleus is fully abstract in that it doesn't deal with the details of storage, network or service operations. The nucleus provides the following abstractions:

- *a history* is a sequence of *updates* to some object (e.g., a directory entry, a file, 3a data page in a file). A history corresponds to a unit in the [HLD of version numbers](#);
- *an operation* is a collection of updates, together with an optional *result*. Each update in an operation belong to some history. An operation is said to belong to the histories its updates are elements of;
- *a version* in a history is a point in history, corresponding to all updates in the history up to a certain update. Each version in a given history is identified by a 64-bit integer unique within the history. This integer is called *a version number*.

Nucleus fully abstracts itself from the details of object and operation identification. Other layers inform the nucleus when a new history should be created and when a new operation should be added.

Here is an example of a fragment of 2 histories "inode X" and "dirpage D", with operation results omitted for simplicity. Semantically, "inode X" corresponds to some file and "dirpage D" corresponds to part of directory, containing directory entries.

The "inode X" history fragment consists of two updates: `X.atime = now` and `X.nlink++` (in the chronological order). The "dirpage D" history fragment consists of 2 updates: `D.del("bar")` and `D.add("foo")`.



Fragments of DTM nucleus histories

Updates `X.nlink++` and `D.add("foo")` are part of a LINK operation. Update `X.atime = now` is part of a SETATTR operation. Update `D.del("bar")` is part of an UNLINK operation, which also contains `Y.nlink--` update, which is part of some other, not shown, history.

As shown on the picture, an operation "joins" multiple histories to which it belongs.

With each update a version number is associated. Version numbers increase in the chronological order (from bottom to top in the picture above).

Nucleus encapsulates handling of basic concepts related to distributed transaction management:

- sequential versioned histories of objects;
- hierarchical grouping of updates into operations, transactions, [epochs](#), &c.;
- transition between volatile and persistent state.

Nucleus delegates the rest of transaction management to the outside modules, driven by call-backs from nucleus. Specifically, other modules deal with:

- representation of histories in persistent store (FOL);
- transmission of operations and notifications through network (FOP);
- detection of stability, based on persistency ([stability detector](#), see below);
- construction of [undo closure](#).

An update in a history can be in one of the following states:

- FUTURE: the update is not yet reflected in the state of the object represented by the history;
- INPROGRESS: the update is being executed;
- VOLATILE: the update is reflected in the state of the object, but the record of the operation this update is part of exists only in volatile store;
- PERSISTENT: the update is reflected in the state of the object and the record of the operation is persistent;
- STABLE: the update is reflected in the state of the object and corresponding records are stable.

An operation is said to be in state X if X is the minimum of the states of the operation's updates, according to the ordering $\text{FUTURE} < \text{INPROGRESS} < \dots < \text{STABLE}$.

Nucleus assumes that objects for which histories are recorded are deterministic

state machines: if the same sequence of updates is applied to the same initial object state, the final observable state is the same, that is, an action applied in the final state has the same result.

This means, among other things, that update execution result cannot depend on timings or races.

Two updates executed one after another in a history are said to commute, when the final observable state after their execution, including the results of the updates, is the same no matter in what order these updates are executed. A pair of histories where only the order of commutative updates differ are called *equivalent*.

An example of commutative pair of updates is a pair of updates incrementing a counter.

An operation and its updates are said to *arrive* to the nucleus (see [op_add\(\)](#) below), when the updates are added to their respective histories in FUTURE state. Then, the updates state changes, according to the algorithms described in the rest of the document, until the update reaches STABLE state. Stable update is eventually garbage-collected.

When an operation moves from INPROGRESS to VOLATILE state, the result of execution is associated with the operation.

For each history, nucleus records a version number of the latest persistent update, called history persistent version number, and a version number of the latest stable update, called history stable version number. A current version number for a history is the version number of the latest at least volatile update in the history, that is, the latest version number reflected in the state of the object.

The nucleus derive much of its simplicity from the flexibility of the notion of history.

Examples of histories are:

- a file (inode). File's history records all updates to file attributes;
- a page of file data. Page's history records all changes to this page;
- a directory page. This history records addition and deletion of directory

entries on the page;

- a distributed transaction. This history records updates executed by a DTM instance as part of the distributed transaction;
- an RPC slot. Slot's history records all rpc items sent through the slot in order;
- an [application domain](#). Domain's history records all operations executed in the domain;
- a local transaction. This history records all updates executed as part of the local transactions;
- an [epoch](#). This history records all operations executed in the epoch;
- a FOL. This history records all operations executed by a particular request handler;
- a clock. This history records all operations from some set of histories (e.g., a per-request-handler clock, per-application-domain clock) and orders them by according to a monotone physical clock of some precision. Clock history is similar to FOL, except that the clock ordering is not necessarily strict: multiple operations can get the same clock value. The advantage of a clock history is that it is more scalable than a FOL history.

Note that clock, as opposed to epoch, contains operations from a single node. An epoch can contain operations from multiple nodes.

An operation typically belongs to multiple histories. For example, a simplest file system call WRITE, would produce a nucleus operation containing updates from the following histories with corresponding updates:

history	update	description
inode	ctime update	change file attributes as part of write
file data extent	new data	change file data
local transaction	no update	adding local transaction history to WRITE

		operation allows tracking all updates from the transaction
fol	no update	all operations are recorded in the FOL
distributed transaction	no update	adding distributed transaction history to the operation allows DTM to determine boundaries of distributed transactions during recovery
application domain	no update	linking all operations in the domain to the domain history allows domains to be rolled back (and forward) individually
epoch	no update	linking all operations to the epoch history allows epoch boundaries to be identified

Some of the histories for an operation are determined by the operation semantics (inode and data page histories in the example above are), others are determined by the DTM layers outside of the nucleus.

A history is called *shared* if other DTM instances maintain histories of the same object. A history is called *local* otherwise. For example, an RPC slot history is always shared, because the sequence of RPC items sent through the slot is maintained by both peers. A local transaction history is always local.

A history is called *full* if it contains all operations for the object. Otherwise a history is called *sparse*. A local history is always full, a shared history can be either sparse or full. For example, an RPC slot history is full, whereas an [epoch](#) history, which is shared, is sparse.

In addition to the result, an operation has a boolean *result flag*. This flag is true iff the operation has been executed somewhere in the system and its result was reported to a user. Result flag is always true by the time the operation reaches VOLATILE state. It can be set to true when the last operation update moves from INPROGRESS to VOLATILE state or it can be already set when the operation arrives to the nucleus. The latter is possible when one nucleus sends already executed updates from a shared history to another nucleus.

An operation or update can change before it reaches VOLATILE state. For an update, its [update rule](#) can change. For an operation, the set of updates it consists

of can change, its result and result flag can change. Once the update or operation leaves enters VOLATILE state, it becomes immutable.

The following notation is used to describe relations between nucleus objects:

H.cver: the current version number of history H;
H.pver: the persistent version number of history H;
H.sver: the stable version number of history H;
H.shared: true iff history H is shared;
H.full: true iff history H is full;
H[I]: I-th update in history H, where the 0 corresponds to the current version, each earlier update has larger index, future updates have negative indices;
U.hi: the history the update U is belongs to;
U.op: the operation the update U is part of;
U.state: U's state (FUTURE, VOLATILE, PERSISTENT, STABLE);
U.ver: the version the history has after U is applied to it;
U.prev: previous (earlier) update in U.hi;
U.next: next (later) update in U.hi;
U.rule: U's update rule (see [below](#));
O[I]: I-th update in operation O;
O.rc: the result of operation O, or NULL if no result.

Updates

An update in a history is associated with a version number. These version numbers are specified according to an *update rule*.

update rule	description	applicability
INC	the update increments the current version number by 1	$C + 1 == U$
SET	the update sets the current version number to the specified value	$C < U$
NOT	the update doesn't change the version number	$C == U$
UNK	the version is not known	true
APP	the update sets the current version number to the specified value	provided

Applicability column specifies when the update can be applied to the history. In this column, C is the current history version number (UPDATE.hi.cver) and U is the update's version number (UPDATE.ver). Note additionally to the condition written, updates must always respect chronology ($C \leq U$), as stated earlier.

INC rule is used for incremental updates, which modify part of the object. SET rule is used for total updates, which completely overwrite the state of the object. An update with the SET rule can be executed before all previous updates are executed, because it overwrites the entire object. UNK rule is used for an update when the version is not known (see the [intent operation use case](#) below). NOT rule is used for an update that doesn't change the state of the object. Such updates are needed, for example, to provide stronger transaction isolation by ordering read-only queries w.r.t. updates. APP updates are equipped with a user provided applicability condition, that can examine state of objects to determine whether the update is applicable. Nucleus requires that after an APP update is executed, its applicability condition becomes false.

An update with the NOT rule is called a *weak update* (which is less confusing than a more precise appellation of a read-only update). Otherwise the update is called *strong*. Strong updates change the state of the object they are applied to. An update with any rule except UNK is called *ordered*.

A nucleus update is called *detailed* if it contains enough information to redo and undo the actual update to the state. Otherwise, the update is called *descriptive*.

Descriptive updates exist due to the [r.m0.dtx.large] requirement: in some cases the amount of information that must be transferred and stored to redo and undo the actual update to the state is too large. A typical example of a descriptive update appears in the parity de-clustered write operation: each service involved in the operation knows about updates made by the other services as part of the same distributed transaction, but it doesn't have the actual data written by the other services. A typical example of a detailed update is a meta-data operation, e.g., MKDIR. In this case each service involved in the operation has full information about updates executed by other services.

Nucleus maintains for an operation a *persistence mask* that conservatively estimates how many detailed copies of the operation updates exist in the system and how persistent these copies are.

For example, when the operation arrives to its originating client DTM instance, each update has a single detailed volatile copy. When one of the updates is re-integrated to a remote DTM instance and reply is received, the mask is modified to indicate that this update now has 2 volatile replicas. The same happens when other updates are re-integrated. When persistency notification is received for an update, the persistency mask is modified accordingly.

Associated with an operation is a *persistency threshold* attribute that determines when the operation becomes persistent. For example, consider an MKDIR operation, containing 2 detailed updates. The persistency threshold for this operation is to have a persistent copy of every update. That is, the operation becomes persistent when both updates are persistent. On the other hand, consider a write to a RAID1 mirrored file. This operation consists of 2 detailed updates, writing identical data to the mirrored replicas. This operation is persistent when any of updates is persistent and so its persistency threshold is to have a persistent copy of 1 update. DTM determines operation persistency by matching persistency mask against persistency threshold. The notions of persistency mask and threshold can be easily generalised to distinguish, for example, between server and client volatile store and different types of persistent store.

Nuclear interface

Nucleus interacts with the rest of the system through the following interface:

- nucleus entry points, called from the outside:
 - `history_add(history, flags, hi_vec)`: creates a new history, with specified flags (local vs. shared and full vs. sparse). `hi_vec` is a vector of functions called by nucleus to notify the user about events related to the history;
 - `op_add(op, update[], op_vec)`: creates a new operation specifying an array of updates. For each update, the history is specified. `op_vec` contains functions that nucleus calls when the operation's state changes;
 - `op_done(op)`: notifies nucleus that the operation execution completed;
 - `op_persistent(op)`: notifies nucleus that the operation became persistent;

- `op_stable(op)`: notifies nucleus that the operation became stable;
- operation functions (`op_vec`) provided by the caller of `op_add()` and called by the nucleus:
 - `ready(op)`: the operation is ready for execution;
 - `persistent(op)`: the operation is persistent;
 - `stable(op)`: the operation is stable;
- history functions (`hi_vec`) provided by the caller of `history_add()` and called by the nucleus:
 - `persistent(h)`: the history persistent version number changed;
 - `stable(h)`: the history stable version number changed;
 - `dry(h)`: the history has no volatile or future operations.

Note that there are incoming and outgoing notifications about persistency and stability, this is explained in the use cases below.

Nucleus interface is used in the following ways:

- when a new operation is added to a history, its updates are initially in FUTURE state. The updates remain in this state until the applicability conditions for all update are satisfied. When the applicability conditions are satisfied, the nucleus notifies the user that the operation can be executed, by changing the update state to INPROGRESS and invoking `op_vec::ready(op)` function. When the execution completes, the user calls `op_done(op)` and the nucleus atomically changes the current version of all histories the operation belongs to and moves the operation updates (and the operation itself) to the VOLATILE state;
- when the nucleus receives a notification that an operation became persistent, `op_persistent(op)` (see below for possible sources of this notification), nucleus goes through the histories the operation belongs to. If the update increases the history version, the nucleus calls history `hi_vec::persistent(h)` function:

```
op_persistent(op) {
```

```

        for u in op.u[] {
            u.state = PERSISTENT;
            if (u.hi.pver < u.ver)
                u.hi.pver = u.ver;
                u.hi.hi_vec.persistent(h);
            }
        }
    }

```

DTM and nucleus

When a client adds a new operation to a distributed transaction, DTM instance builds corresponding nucleus operation. This operation consists of updates of 2 kinds:

- semantic updates, determined by the action type. For example, a MKDIR operation consists of the following semantic updates:
 - update parent directory attributes (ctime and nlink);
 - add new directory entry to the parent directory;
 - allocate new inode number for the newly created child directory;
 - setup new directory inode with supplied attributes (permission mask);
 - add dot and dotdot entries to the child directory.

Each semantic update belongs to a nucleus history. For example, the list above assumes that the DTM instance maintains separate histories for directory inode, inode allocation table and directory "body" (where directory entries are stored) or, alternatively, for each page of directory body.

The nomenclature and granularity of histories are determined to satisfy concurrency and memory usage trade-offs, which are beyond the scope of the present document.

- control updates, which are added to track transaction boundaries. At the very least DTM instance adds a weak update to the distributed transaction. Depending on the configuration, DTM instance can add more weak control updates:
 - to an application domain object. This update is added when the distributed transaction, which the operation is part of, is opened

against an application domain;

- to an epoch object. See [below](#) for more details;
- to an FOL. FOL linearly orders all operations executed by a given Mero instance and associates a version number in the FOL history, also referred to as an LSN or log sequence number to every operation.

FOL is used for variety of purposes, see the [HLD of FOL](#).

Control updates added by the DTM instance do not update anything. Their undo and redo actions are no-op, they are used as placeholders to record relationships between other objects.

User selects [update rule and version](#) for a semantic update.

When the operation is re-integrated, the DTM instance puts in the operation's FOP enough information to reconstruct the nucleus operation. No additional information is necessary for the semantic updates, because they are completely described by the operation parameters.

When the client closes the distributed transaction, the DTM instance constructs a special CLOSE operation, containing a strong update of the distributed transaction history. This operation is internally re-integrated by the DTM to all services hosting histories involved in the transaction, if possible by piggy-backing the CLOSE fop to the last fop sent as part of the transaction. Again, CLOSE operation doesn't change any state, it only records transaction boundary.

When the service receives the FOP with an operation description, the corresponding nucleus operation is created from the description. This nucleus operation contains semantic and control updates as in the client case. In addition to the other control updates, the operation created by the service also contains a weak update of a local transaction within which the operation is executed.

When the local transaction on the service is closed, the DTM instance constructs a special CLOSE operation, containing a strong update of the local transaction history.

When the local transaction becomes persistent, DTM calls `op_persistent(CLOSE)`. If

nucleus call `hi_vec.persistent()` of a shared history, the DTM instance sends notification to the remote DTM instance maintaining a copy of the history (subject to certain batching policy). When a DTM instance receives such notification, it calls `op_persistent()` for the matching operation.

It is assumed that local transactions modifying the same object become persistent in the order of such modifications [**u.m0.txn.ordered**]. This immediately means that for a given history H, it is exactly all updates not later than H.pver that are persistent. Recalling the definition of persistency, this means that after a DTM instance fails and restarts with lost volatile store, the persistent store contains prefixes of histories up to and including their persistent versions at the moment of failure.

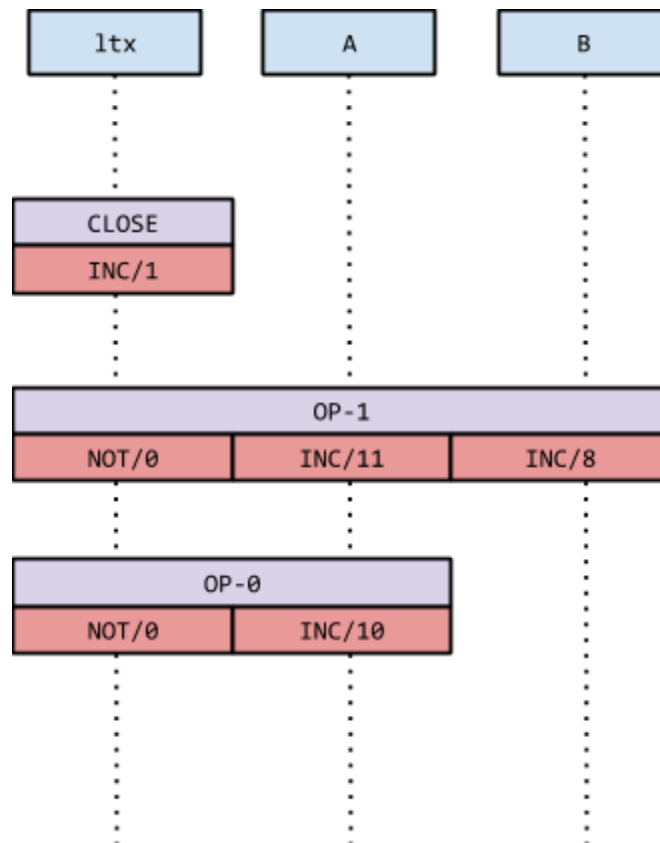
Service

This sub-section considers a typical use case of DTM nucleus on service side.

When an operation arrives to a service for execution, the DTM instance creates a nucleus operation, which is linked into multiple histories.

Originally, all updates are in the FUTURE state. As [described above](#), when applicability conditions of all updates are satisfied, nucleus moves the updates in the INPROGRESS state.

Some histories the operation belongs to, as mentioned above, are determined by the operation type, some by the DTM configuration. Specifically, the service groups operations it executes in local transactions (subject to a certain policy, details of which do not at the moment matter. For example, every operation can be executed as a separate local transaction). To a local transaction corresponds a nucleus history, to which every operation executed as part of the local transaction belongs to:



On this picture, a local transaction (ltx) consists of 2 user operations: OP-0 and OP-1. OP-0 modifies object A, increasing its version to 10. OP-1 modifies objects A and B, increasing their versions to 11 and 8 respectively. Each operation also contains a weak update, linking the operations to the local transaction history. These updates don't change the version of the local transaction history.

The local transaction is not necessarily known when the operation arrives to the service, but it must be known before the operation leaves the INPROGRESS state and becomes VOLATILE.

When the local transaction is closed, DTM adds a special CLOSE update to the history corresponding to the local transaction. This update has INC rule and changes the local transaction history version from 0 to 1.

Some time after the local transaction is closed, it becomes persistent and the local transaction mechanism invokes a commit hook ([u.M0.BACK-END.COMMIT.CALL-BACK ST]), provided by the DTM instance. This call-back calls op_persistent(CLOSE). op_persistent() increases ltx.pver to 1 and calls

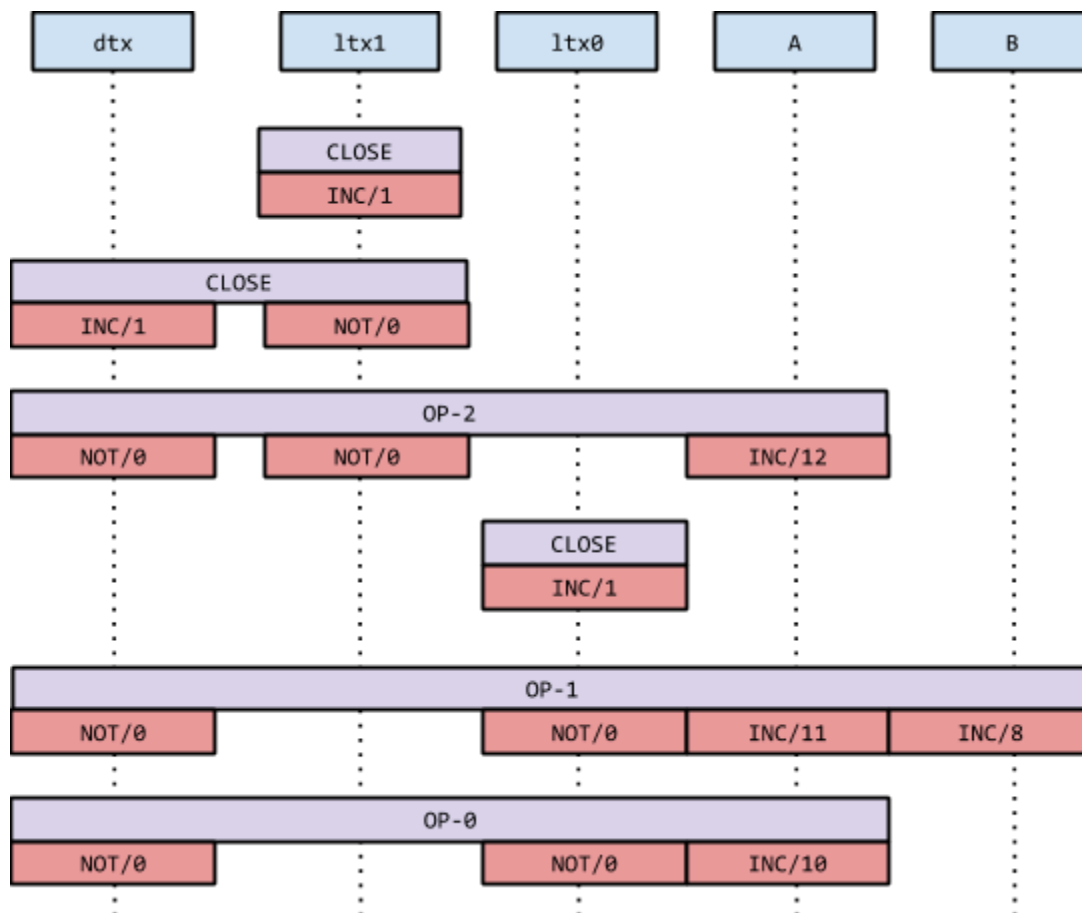
ltx.hi_vec.persistent(ltx), because CLOSE increases version number.

ltx.hi_vec.persistent() is a function, provided by DTM that calls op_persistent() for each operation belonging to ltx history:

```
ltx_persistent(h) {  
    for (i = 0; h[i] exists; ++i)  
        op_persistent(h[i].op);  
}
```

These calls increase A.pver to 11, B.pver to 8 and call A.hi_vec.persistent() and B.hi_vec.persistent().

Now, consider a more complex case, where operations are part of a distributed transaction:



A distributed transaction (dtx) is executed in 2 local transactions: ltx0 and ltx1,

possibly separated in time. ltx0 is the same as in the previous case, except that all operations in ltx0, excluding the CLOSE operation, contain an additional weak update linking them to the dtx history.

ltx1 contains a single user operation OP-2, modifying object A.

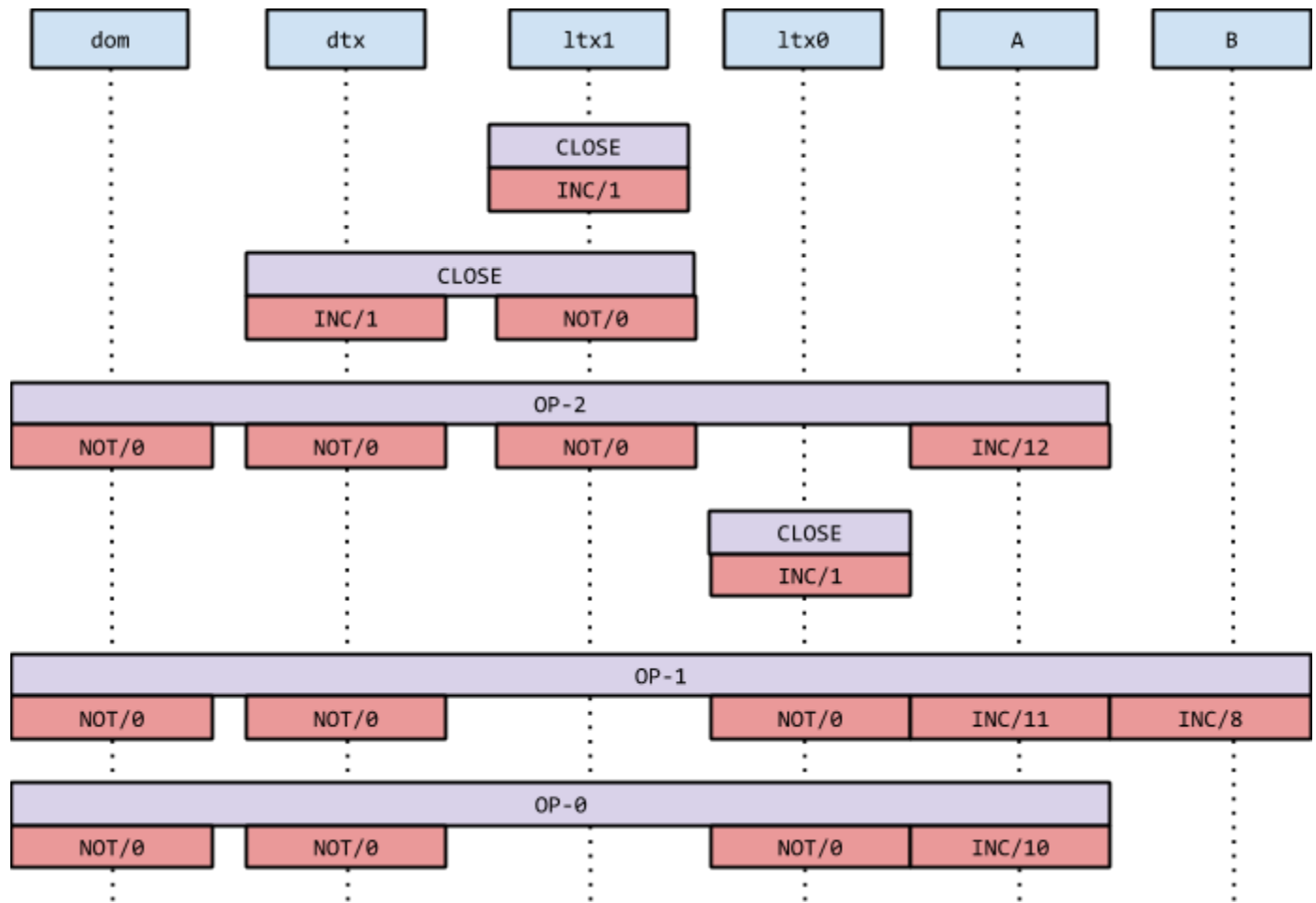
The distributed transaction is closed as part of ltx1, by the CLOSE operation, that increases dtx version to 1, but doesn't change ltx1 version. This arrangement is not the only one possible: a distributed transaction can be closed in a separate local transaction.

Finally, ltx1 is closed.

When ltx0 becomes persistent, A and B persistent versions are increased, but dtx.hi_op.persistent() is not called, because all ltx0 updates to dtx are weak.

It is only when ltx1 becomes persistent, dtx.hi_op.persistent() is invoked.

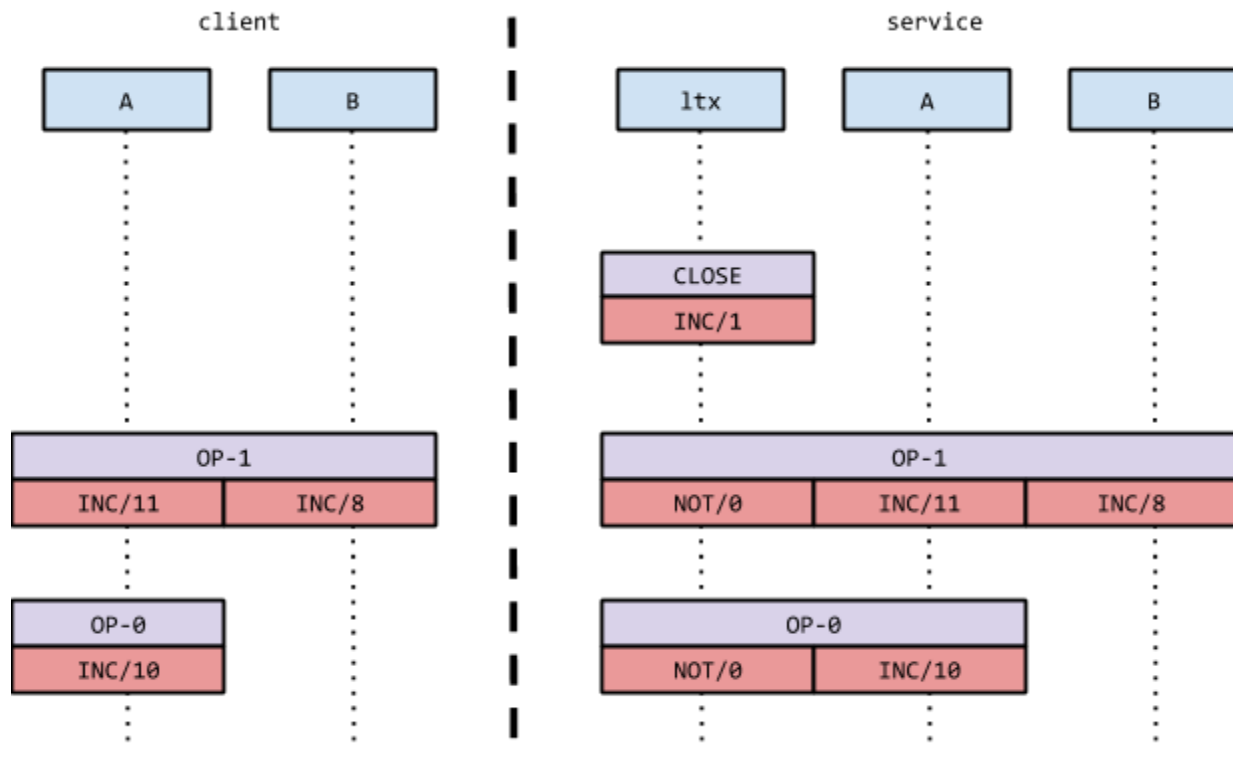
Final case adds an [application domain](#):



Every user operation in the application domain includes a weak update to the domain history. Because domain version number is never changed, domain history is never notified about changes in its persistency. The domain history is used to track all operations executed in the domain.

Client

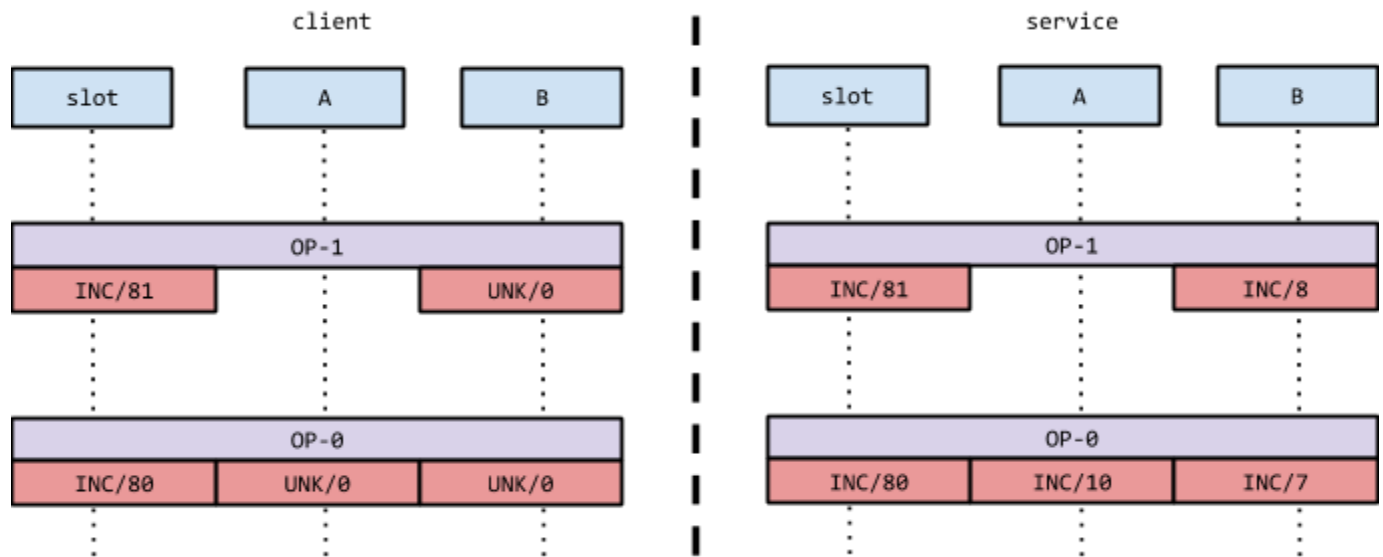
The simplest case of interaction between 2 DTM instances is a "write-back cache" client. Such client keeps its own cache where operations are done locally before re-integration (this requires appropriate write credits from the resource manager):



Client knows versions of objects in its cache, so it can assign correct versions to the updates. The updates are transmitted to the service (through fop mechanism), where they are executed (distributed transactions are omitted for simplicity). When local transaction (ltx) becomes persistent, `ltx.hi_vec.persistent()` is called, advancing `A.pver` and `B.pver`. Because A and B are shared histories, DTM propagates notification about persistency change to all other DTM instances that maintain copies of A and B (the client node in our case). This propagation is done by sending a 1-way rpc to the client. On receiving such notification, the client DTM instance calls `op_persistent(OP-0)` and `op_persistent(OP-1)`.

As in the service case, updates of an operation in the write-back cache are originally in the **FUTURE** state. They are moved to **INPROGRESS** state when re-integration starts. When the reply to the re-integration message is received, the update moves to the **VOLATILE** state. Eventually, it is moved to the **PERSISTENT** and **STABLE** states.

Next use case is so-called "intent operation" where the client keeps no cache. Instead of sharing object histories with servers, the client shares a special object called "a slot". A slot is used to generate version numbers and to order operation execution (local and distributed transactions are omitted for simplicity):



Instead of version numbers, the client uses UNK rule for all updates in an intent operation, except for the slot update, for which it uses known slot version number, increased on each operation (the client can safely manipulate the slot version number, because the slot history is full: no other DTM instance changes it).

When, as part of re-integration, the operation is added to the service DTM nucleus, slot version number is used to determine when the operation is ready to be executed (UNK updates have trivial applicability condition). When the operation is executed, DTM replaces UNK/0 versions with the version numbers that resulted from the update execution. These actual version numbers are sent to the client together with the operation result. The client DTM stores actual version numbers: they are needed to order re-execution of the operations in case of service failure and restart.

Persistent format

As operations are added, the nucleus histories grow longer and cannot always be stored in the primary (volatile) store. For this reason, nucleus information must be encodable in a format suitable for secondary (persistent) store.

Naïve encoding of a history as a sequence of updates indexed by version numbers and of an operation as a tuple of updates, is prohibitively expensive and wasteful. A minute contemplation would convince oneself that all nucleus algorithms traverse histories only pastward. Adding to that an obvious observation that an update in the middle of a history is never deleted, it's easy conclude that a history can be

efficiently represented as a single-linked list.

The last remaining ingredient of an efficient storage representation is FOL: every operation executed by a DTM instance belongs to the instance's FOL and, hence, is uniquely identified by its LSN. An update belongs to an operation and is, therefore, uniquely identified by operation's LSN and an index within the operation.

Thus, every history, except for the FOL, can be represented as a header:

```
struct m0_dtm_history_bin {
    m0_lsn_t h_cver; /* last update */
    m0_lsn_t h_sver; /* last stable update */
};
```

Note that there is, obviously, no need to store "last persistent" update on the persistent store.

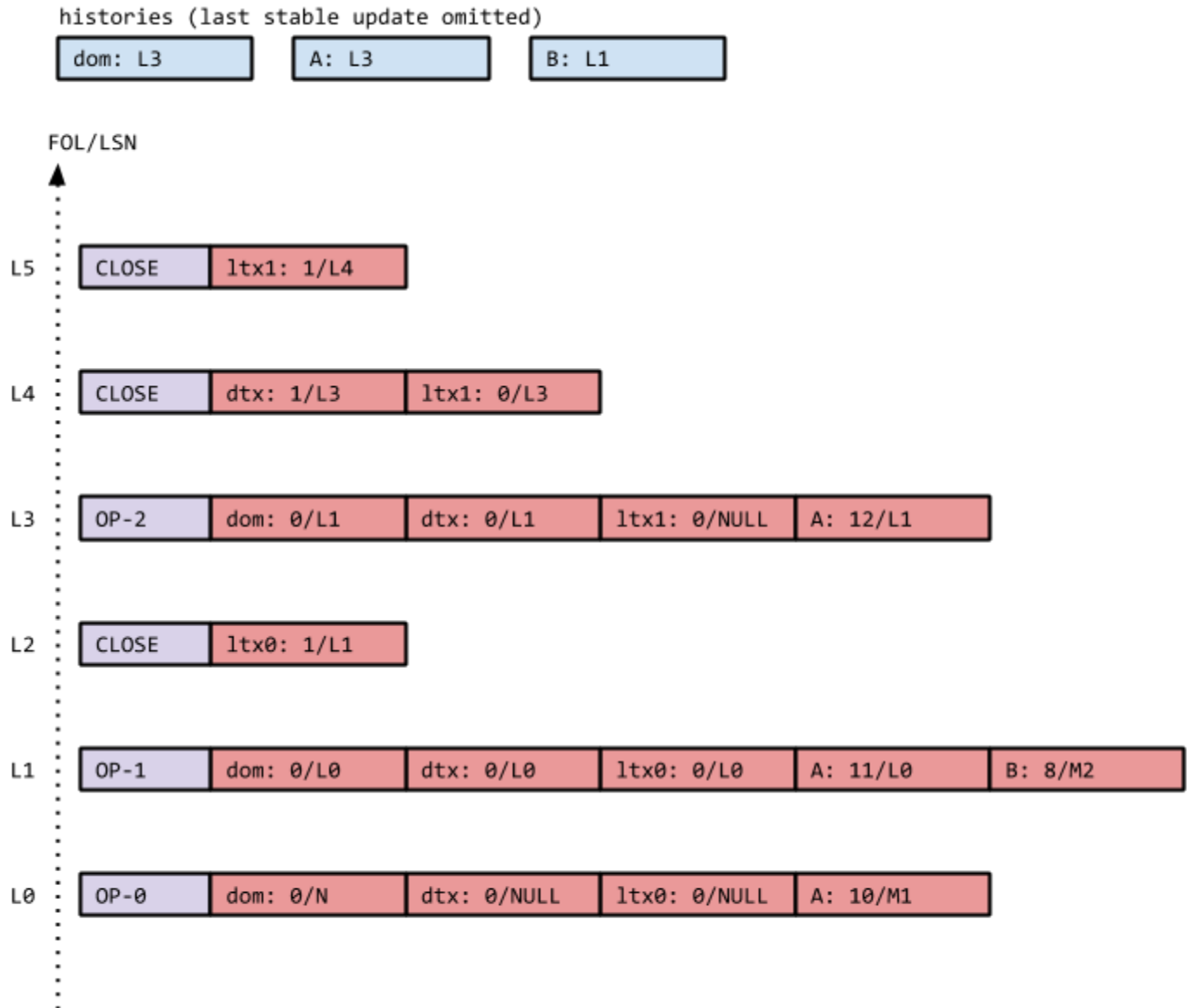
An operation, stored in the FOL, is represented as

```
struct m0_dtm_op_bin {
    /* number of updates */
    uint32_t o_nr;
    /* array of updates */
    struct {
        /* version after the update */
        uint64_t u_ver;
        /* previous update in the same history */
        m0_lsn_t u_prev;
    } o_updates[0];
};
```

FOL representation is up to the local transaction manager.

The actual storage format would include parameters of each update.

For example, the [diagram](#) above can be represented as



Where H: V/P means an update to history H, changing the version to V and where P is the LSN of the previous update in H. P of NULL means the first update in H.

A part of a history that became persistent can be deleted from the primary store. It is not necessary to delete it immediately, some caching policy can be applied. When history has to be scanned past its fragment maintained in the primary store, history is read from the persistent store and converted to the primary store format.

Eventually, histories on persistent store are truncated, as described [below](#).

Shared histories

Recall that a history is shared when multiple DTM instances maintain it. These DTM instances are called *participants* in the shared history.

It is assumed that the list of participants is known to every participant when the history is added to the participant and, moreover, that the list is at least as stable as the history itself, *i.e.*, when any record of the history becomes persistent, the list is recorded in persistent store. For example, in the persistent format, described above, the participants list can be encoded in struct `m0_dtm_history_bin`.

DTM keeps shared histories consistent in the following sense:

- if multiple participants contain 2 at least VOLATILE updates, all participants agree about the ordering of updates;
- if multiple participants contain an instance of at least VOLATILE operation, all participants agree on the operation result.

The first condition is non-trivial because of un-ordered updates (*i.e.*, updates with UNK rule). DTM guarantees that an un-ordered update is replaced with the ordered one before the update reaches the VOLATILE state. Note, however, that the first condition doesn't require that all participants see the same version numbers: it is relative ordering that must be preserved.

Both condition are also non-trivial because of DTM instance failures, which lose volatile state, including parts of shared histories.

Two DTM mechanisms maintain history consistency:

- when an operation moves from INPROGRESS to VOLATILE state, operation's result is sent (subject to some batching policy) to the DTM instances hosting any other full instance of any shared history the operation belongs to. For example, when a service completes execution of an operation, received from a client, it sends the result back to the client. But, when the client moves the operation to VOLATILE after receiving the last reply, it doesn't send the results back to the services (from where the result was received);
- when a version of a shared history becomes persistent, a notification about this is posted to a *persistence aggregator*. A DTM instance has a persistence aggregator for every other DTM instance it shares histories with.

Persistency aggregator is used (as opposed to directly sending persistency notifications to every participant), because sometimes it is possible to compactly encode a large number of persistency notifications at a slight increase in latency. For example, assuming that every operation executed by the DTM instance has an update to the shared FOL history, it is possible to roughly describe the state of all histories by a single number: the smallest LSN of a non-persistent operation. When a participant receives this single number it goes through all its histories shared with the sender and marks as persistent all updates with FOL LSN less than the received number.

Similarly, a clock history can be used to aggregate a body of persistency notifications to a single number. More elaborate scenarios are described [elsewhere](#).

Note that a list of participants can be "everybody". For example, an [epoch](#) history can be shared with every node in the cluster. To send results or persistency notifications to a history shared with everybody or, more generally, to a "widely-shared history", DTM uses a scalable broadcast interface [**u.m0.ha.scalable-broadcast**].

Network failures

Assume that any operation added to a distributed transaction contains, for each DTM instance the operation is re-integrated to, at least one ordered update to a history shared with the DTM instance [**u.m0.dtx.ordered-update**]. This condition assures that every DTM instance will be able to insert the operation in the correct place in corresponding histories.

If, for a particular DTM instance, an operation contains multiple ordered updates of shared histories, they must be ordered consistently. This is typically guaranteed by the resource manager.

Looking at the applicability conditions for *strong* ordered updates, it's easy to observe that once nucleus [updates the history current version](#) after a strong ordered update is executed (moves from INPROGRESS to VOLATILE state), the update's applicability condition is false (formally, substituting U for C in the strong ordered update [applicability condition](#) produces false).

This leads to the following simple applicability check: when an operation arrives to a DTM instance, take any strong ordered update U in the operation and

- go through all INPROGRESS updates for $U.hi$. If the update is found, reject the operation with "already in progress" error; otherwise
- if $U.ver < U.hi.cver$ or user provided applicability check for the APP update returns ALREADY, reject the operation with "already executed" error; otherwise
- queue the operation for execution.

This check can be further simplified, by maintaining an additional field $U.hi.aver$, where the version number of the latest update accepted for execution is stored.

For strong ordered updates this check guarantees exactly once execution (EOS). For weak updates, this check guarantees at least once execution, which is also fine, because weak updates, by definition, can be executed multiple times.

This check, together with resend on timeout, handles all allowed network failures.

Node failures

Recovering consistency after a node failure and restart is the *raison d'être* of DTM.

The general process of a recovery after a node failure is

- redo step. If a detailed update from a shared history is lost by some participants of the history, but retained by some other, these others resend the update so that it can be re-executed;
- undo step. If an update U from a shared history is lost by all participants, it cannot be recovered, which means that all later updates not commuting with U must be removed from all instances of $U.hi$. Which means that all updates of all operations of all distributed transactions containing removed updates must also be removed, leading to more update removals, &c.

Second step builds an *undo closure* of the failure. Similarly to [stability detection](#), there are multiple algorithms to build the undo closure different in their communication and storage overhead, and in their granularity (some algorithms

remove more updates than strictly necessary). Unsurprisingly, stability detection and undo closure algorithms come in pairs.

In general, recovery can always be done by executing an undo step, followed by a redo step. However, an additional redo step at the very beginning, can shrink the closure, reducing the number of undone transactions.

Redo

Taking the **[r.m0.dtx.large]** requirement into account, updates are classified in *small* and *large*.

For a large update in a shared history, some participants have only descriptive update.

This restriction essentially mandates the implementation of the redo step for large updates: the participant with the detailed update, must keep the update in its nucleus history until the update becomes persistent on all participants.

If a participant with the descriptive update fails and loses the update, the participant with the detailed update goes through the history pastward and changes states of updates to FUTURE until it passes the lost detailed update. This causes all updates, including the lost one to be re-integrated in the right order.

If all participants with the detailed update fail and lose the update, the update is completely lost and redo step cannot help.

A better scheme is possible for a certain class of small updates.

For a strong ordered update U call its *prerequisite sequence* the minimal sequence of updates that must be executed "from the beginning of the Universe" before U can be executed.

The prerequisite sequence is defined recursively as the union of

0. the minimum sequence of updates that must be applied in the initial state of U_{hi} before U can be executed (call this sequence $S(U)$) and
1. the prerequisite sequence of update that created U_{hi} , if any.

The sequence $S(U)$ defined in the step (0) depends on U 's update rule and is itself

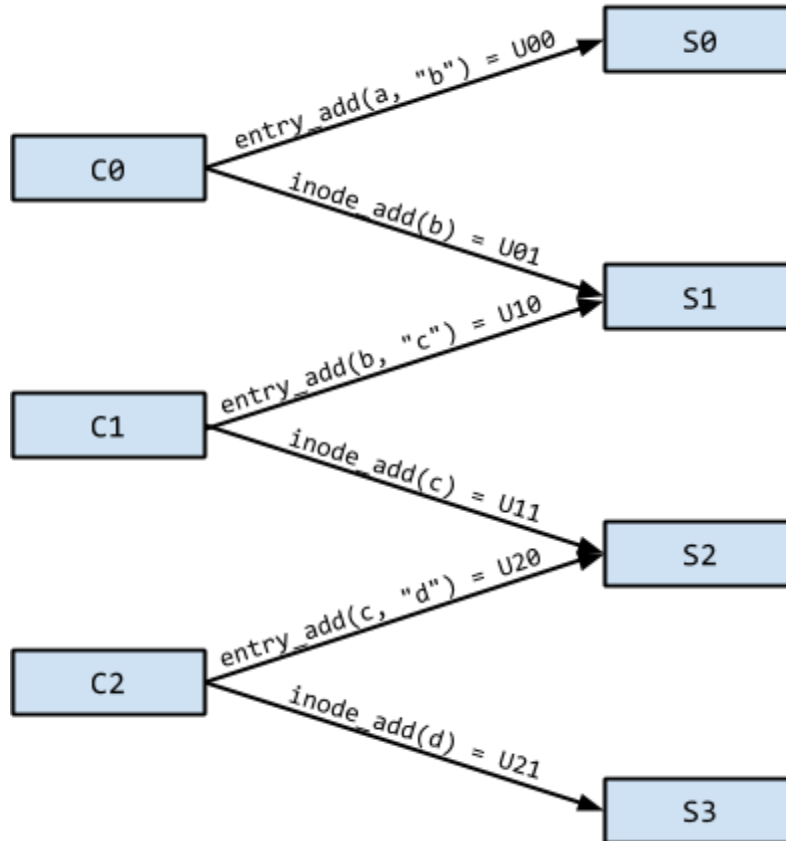
defined recursively

- INC: union of U_{prev} and $S(U_{\text{prev}})$;
- SET: empty (SET update can be applied to the initial state of the history);
- APP: depends on the applicability rule. For example, an operation creating a new file can be executed after the parent directory has been created. An operation to unlink a file can be executed after the parent directory has been created and the file, being unlinked, has been created.

A sub-sequence of U 's prerequisite sequence consisting of still not persistent updates is called U 's *redo tail*. Redo tail of an update contains all still not persistent updates which must be executed before the update in question can be executed.

The following situation will be used as the running example:

- client C_0 gets an MKDIR("a/b") request that must be re-integrated to 2 servers S_0 and S_1 (e.g., the parent directory "a" is on S_0 and new sub-directory "b" is created on S_1) as a pair of updates U_{00}, U_{01} ;
- client C_1 gets an MKDIR("a/b/c") request that must be re-integrated to 2 servers S_1 and S_2 as a pair of updates U_{10}, U_{11} ;
- client C_2 gets an MKDIR("a/b/c/d") request that must be re-integrated to 2 servers S_2 and S_3 as a pair of updates U_{20}, U_{21} .



Assuming that none of the U_{ij} is persistent and the operation that created directory "a" is stable, the redo tail of each update is the following:

- $U_{00}.rtrail = U_{01}.rtrail = \{\}$;
- $U_{10}.rtrail = U_{11}.rtrail = \{U_{00}, U_{01}\}$;
- $U_{20}.rtrail = U_{21}.rtrail = \{U_{00}, U_{01}, U_{10}, U_{11}\}$;

Restrict the attention to the small updates with SET rule, and to the histories which are either never created or created by means of small SET updates. In this case, the prerequisite sequence of U has a very simple structure: it's a list of consisting of the update P that created $U.hi$, the update GP that created $P.hi$, the update GGP that created $GP.hi$, &c. until history that wasn't ever created is reached.

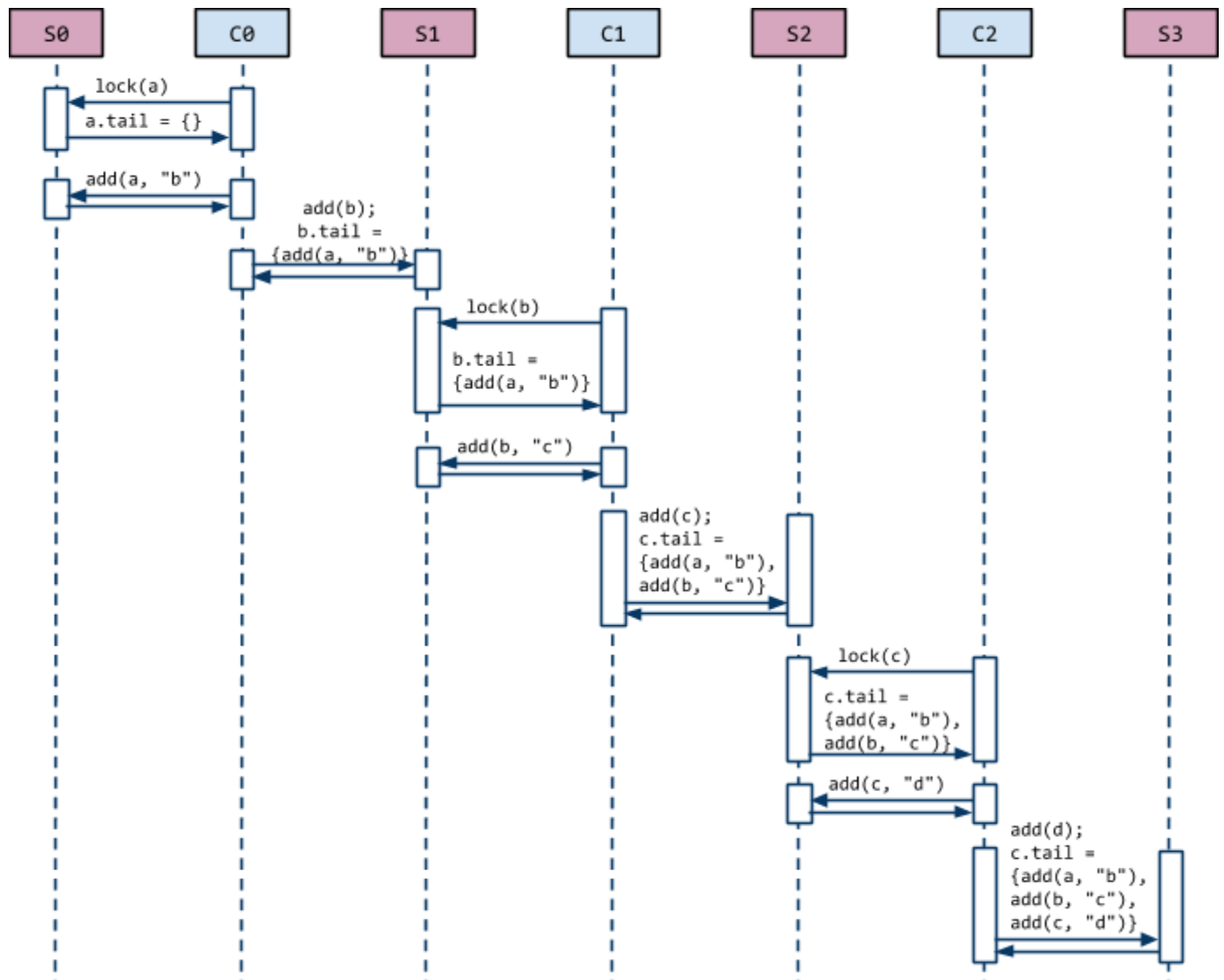
This class of updates and histories includes the running example and, in general, all POSIX meta-data operations on hierarchical directory trees and file attributes.

Within this class, redo-tail can be associated with a history rather than an update within the history. In addition, note that because all updates must be ordered,

clients must use [write-back cache](#) (as opposed to [intent operations](#)). Under these conditions, redo tail can be maintained according to the following rules:

- when a client requests a write lock on an object, the redo-tail of the object is sent to the client together with the lock;
- when a client creates a new child object, it adds the redo-tail of the parent object, together with the update that created the parent object to the redo-tail of the child;
- when a client re-integrates updates or surrenders the lock, it sends redo-tails of involved objects to the service.

In the running example, the propagation of redo-tails looks like this:



In the described scenario, each service maintains in its histories and redo-tails enough to redo all updates. For example, should S2 fail in the running example, S3 could re-integrate `add_entry(c, "d")` and `add_inode(c)`, because their are stored in redo-tails on S3.

In this situation, stability is equivalent to persistency: if any update of an operation, together with its redo-tail is persistent, operation is stable, because the redo-tail contains enough information to execute the operation.

Undo closure

Undo closure determines the set of updates that should be undone to restore consistency after a node restart.

The [redo-tail mechanism](#) described above provides for the simplest possible undo closure algorithm: nothing should be undone, the undo closure is empty. In other words, redo-tail supports *redo-only recovery*.

In other cases, some updates must be undone. If an update is undone, the operation the update is part of must be undone. If an operation is undone, the distributed transaction the operation is part of must be undone. In addition, if an update U is undone, all later updates from U.hi not commuting with U must be undone.

It is possible to directly implement the description above as a distributed algorithm that would iteratively build undo closure by exchanging information about updates, operations, distributed transactions and histories. However, such algorithms won't be practically usable, because the amount of information that must be exchanged through network would be prohibitively large and, in addition, the number of iterations (each involving message round-trips) is unbounded.

Epochs

Practical undo closure algorithms are based on ordering of distributed transactions. Distributed transactions are naturally partially ordered: $T_0 < T_1$ when they update the same history H and any T_0 's update to H is earlier than any T_1 's update to H.

Undo closure algorithm can be constructed given a monotone mapping F from distributed transactions to some partially ordered set of *epochs* such that if $T_0 \leq T_1$ then $F(T_0) \leq F(T_1)$. Epochs must form a [lower semilattice](#), that is every pair of epochs has the greatest lower bound (infimum, inf). The set of epochs is chosen such that ordering in it and infimums are easy to compute.

A distributed transaction is *closed* on a DTM instance X when the CLOSE operation for the transaction has been executed by X. An epoch M is *hereditarily closed* on X, denoted HC(M), when all transactions on X with epochs less than or equal to M are closed on X:

$$HC(M) \equiv \forall t.(t \in X.dtx \wedge F(t) \leq M) \rightarrow t \text{ is closed}$$

On a failure, each DTM instance X reports the *maximum hereditarily closed epoch* MHC(X) that is a hereditarily closed epoch such that any larger epoch contains open transaction. Any transaction with the epoch less or equal to the MHC(X) is closed on

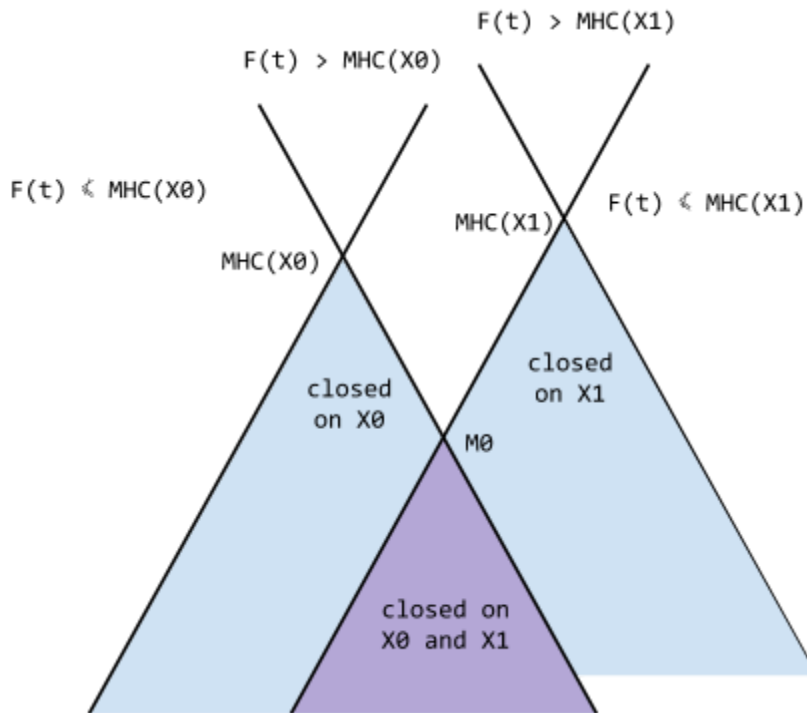
X.

$$F(t) \leq MHC(X) \rightarrow t \text{ is closed}$$

The undo closure is built by taking the infimum of $MHC(X)$ reported by all instances

$$M_0 = \inf \{MHC(X) \mid X \text{ is a DTM instance}\}$$

and by undoing all updates from transactions t such that $F(t) \not\leq M_0$. Note that epochs are, in general, only partially ordered and the last condition is not equivalent to $F(t) > M_0$.



In the diagram above, everything outside of the violet cone is undone.

Correctness of this algorithms is easy to observe:

- by the definition of $MHC(X)$, all transactions t , such that $F(t) \leq MHC(X)$ are closed on X ;
- this means that all transactions such that $F(t) \leq M_0$ are closed on all DTM instances. This guarantees that the set of not undone updates consists of

entire operations and transactions;

- if an update U is undone and an update V is later than U , then $V.op.dtx \geq U.op.dtx$ by the definition of the natural partial ordering of transactions and $F(V.op.dtx) \geq F(U.op.dtx)$ by the monotonicity of F . This means that V is also undone.

As an example, let epochs to be the set of integers and $F(T) = 0$ for all transactions. If at least a single update is lost in a failure, one transaction remains unclosed on some DTM instance and this instance returns MHC of, say, -1. The resulting undo closure consists of all transactions. This algorithm is obviously correct, because it restores the system to its initial state, but it is not very practical, because it doesn't have any liveness guarantees: transactions never stabilise and can always be undone.

This example demonstrates that undo closure algorithms can undo more updates than strictly necessary. It [can be shown](#) that by taking epochs to be the set of n -tuples of integers, where n is the number of histories in the system and using coördinate-wise partial ordering, a *precise* undo closure algorithm can be designed that undoes no more transactions than strictly necessary.

As the next example, consider a system with perfectly synchronised clocks on each DTM instance. Then, taking the set of possible clock readings as the set of epochs and $F(T) = \text{time-when-}T\text{-was-opened}$ obtains a simple undo closure algorithm, provided that clock granularity is sufficient to maintain the defining property of F (specifically, if no transaction can be opened and closed faster than the clock ticks).

This general mechanism can be applied to the [standard scenarios](#):

- **local operations** and **local-distributed**. Undo closure algorithms is the same in these cases. Because distributed transactions are not actually distributed (all updates arrive to the same DTM instance), DTM instance FOL can be used for transaction ordering: F maps a transaction to the LSN of its first update. On a node failure and restart, the DTM instance goes through the FOL undoing all updates until all open transactions are completely undone (note that some closed transactions can be undone in the process);
- **application domain**. An application domain has a single DTM user, called owner. Only the owner re-integrates operations on domain objects.

Introduce a special *distributed transaction sequence object* (DTSO) whose history is shared by all services in the domain and by the owner. Each distributed transaction started by the owner, contains an INC update, incrementing the version of the DTSO by 1. Effectively, distributed transactions in the domain are sequentially ordered by DTSO versions numbers.

Epochs are integers. F maps a transaction to the version number of DTSO the transaction sets.

The undo closure construction is the following:

- if the owner is alive, it contacts all the services in the domain and asks them for the version number of the DTSO in their nuclei. The service undoes still open transactions before returning the DTSO version number.
- the owner finds the minimum of the version numbers returned by the alive services;
- the undo closure consists of all distributed transactions with the DTSO version number larger than the minimum.

If the owner is dead, no recovery takes place, until the owner restarts.

This algorithm, while obviously correct, has a drawback: if some service in the domain didn't participate in transactions for some time before the failure, the domain will be restored to the last transaction this idle service participated in. To avoid unnecessary undo, every distributed transaction in the domain can be lazily broadcast to each service, that is, dummy updates to some dummy objects are added to every transaction so that it has updates on every service. These dummy updates can be accumulated and re-integrated lazily. This modification puts a bound on the amount of undone transactions.

general. In the general case, multiple client DTM instances concurrently re-integrate updates from multiple distributed transactions updating a shared set of objects. This document gives only the brief overview of the implementation, see the References, specifically [\[Lustre-devel\] global epochs](#)

[\[an alternative proposal, long and dry\]](#) and [Report: distributed transactions stabilization \(epochs\)](#) for more details. The undo closure mechanism for the general case is based on associating a variable epoch with a DTM instance. An operation re-integrated as part of a distributed transaction is tagged with the epoch at the moment when the transaction was opened. Instance epochs are maintained in such a way, that if 2 distributed transactions update the same history, the epochs are in the same order as updates versions. This property is achieved by using some logical clock algorithm (scalar, vector, matrix, &c.). Epochs are intervals in the [logical time](#), which are used to detect stability of a large number of transactions at once. The important properties of epochs are:

- epochs can be ordered (totally or partially, depending on the used clock algorithm) and
- an operation in an epoch cannot depend on an operation in the later (according to the ordering) epoch;
- all operations of a distributed transaction belong to the same epoch.

Because of these properties, if all operations in an epoch are persistent and all operations in all previous epochs are persistent too, all operations in the epoch are stable.

To construct the undo closure after a restart, DTM instances find out (by scanning their FOLs) the largest epoch, such that all transactions with this or lesser epochs are closed (MHC). The infimum of MHC-s is calculated and all updates from transactions with the epoch neither less than nor equal to the infimum are undone.

The scenarios described above include a sub-task of "scanning the FOL pastward until all open transactions are scanned". To limit this scan a special history can be maintained in the FOL:

0. record in the volatile store all transactions open at this moment in time;
1. write to the FOL a special record, containing a pointer to the previous record produced by this process;
2. wait until all transactions in the list recorded at the step (0) are closed;
3. repeat from the step (0).

To scan past all open transactions it's sufficient to scan the FOL until the second record, produced by the process above is met *i.e.*, to scan one interval between 2 records, because all transactions open at the beginning of such interval are closed by its end.

Stability detector

Examples above demonstrate dissemination of information about persistency of histories, originating from nucleus and local transaction mechanism. Persistency, however, is very different from stability, because even an operation that is persistent on all services where its updates were executed is not necessarily stable, because some previous operation on which the operation in question depends can be non-persistent.

Stability detection is needed for 2 purposes:

- a user might be interested in knowing when the data are stable (fsync(2) system call in POSIX);
- redo information, including redo tails can be discarded once it is known that it will be never used by recovery. To avoid redo information growing without bound, the information about stable updates should be discarded.

There are multiple mechanisms that detect stability of operation (based on persistency). Efficiency of these mechanisms depends critically on the system configuration and work-loads. Because of this, stability detection mechanism is abstracted as a *stability detector*. Multiple stability detectors can co-exists simultaneously in the system. They differ in granularity (how many operations they declare stable in one go), latency and overhead. Multiple stability detectors can be applicable to the same entity. The first one to declare the entity stable, wins.

From the definition of stability, it is clear that stability detector depends on the undo closure algorithm used:

- for redo-tail recovery, persistency is equivalent to stability and an entity can be removed from the redo queue the moment nucleus informs about its persistency;

- for epoch recovery, the same assignment of epochs to distributed transactions (and DTM instances) is used for stability detection.

In the latter case, call an epoch hereditarily durable on a DTM instance, when it is hereditarily closed and hereditarily persistent on the instance. DTM instance monitors its maximum hereditarily durable epoch and when this epoch changes, it notifies the *stability detector* about the change. Stability detector calculates the infimum of hereditarily durable epochs reported by the DTM instances and notifies the instances when the infimum changes. All epochs less than or equal to the infimum are stable and can be pruned from redo queues.

The stability detector is the same entity that coördinates the undo closure construction:

- in the **local** and **local-distributed** cases the DTM instance is the stability detector for itself;
- for the **application domain** case, the application domain owner is the stability detector;
- in the **general** case, the infimum must be calculated across all DTM instances. Thanks to the nice properties of the infimum function (idempotence, commutativity and associativity) this can be efficiently done by organizing DTM instances in a tree and calculating infimum bottom-to-top in this tree.

5.1. Conformance

[For every requirement in the Requirements section, this sub-section explicitly describes how the requirement is discharged by the design. This section is part of a requirements tracking mechanism, so it should be formatted in some way suitable for (semi-)automatic processing.]

- **[R.M0.DTX]** distributed transactions are supported.

Distributed transactions support is explicitly and extensively described by the specification.

- **[R.M0.DTX.APPS]** storage application (FDML extension plugin) activity is

executed as a part of the same dtx as the original file system operation. This guarantees that if file system operation is later rolled back due to a failure, storage application state is also rolled back, keeping the latter consistent with the core.

When a DTM instance determines that an arrived operation from a certain distributed transaction contains an update that must be delivered to an FDMI extension plugin, the instance

- delivers the update to the plugin together with the transaction description so that the plugin can participate in the same transaction and can notify the originating DTM instance about changes in persistency and
 - appends to in its reply to the originating DTM instance description of the plugin so that the originating instance would watch for the persistency notifications from the plugin.
- **[R.M0.DTX.ATOM]** Dtx can be used to implement ATtributes On Meta-data service (ATOM).

ATOM requires transactions containing operations separated by potentially long intervals of time. In the particular case of attribute-on-mds, a client transactionally writes to ioservice and updates file attributes on mdservice, where update is sent to the mdservice long time after the ioservice update, for example, when the file is closed.

This type of transactions can be implemented through redo-tail mechanism: detailed mdservice update is sent to the ioservice together with the main update. The operation becomes persistent when these updates become persistent on the ioservice. Should client fail before it sends the update to mdservice, the ioservice would send this update.

- **[R.M0.DTX.AVAILABILITY]** Non-blocking availability layer can be based on dtx.

Transaction containing updates to multiple meta-data tables are supported.

- **[R.M0.DTX.BACK-ENDS]** a distributed transaction can contain updates for

state on multiple back-ends both local and remote.

Explicitly supported.

- **[R.M0.DTX.COUPLING]** a degree of coupling can be specified for a distributed transaction.

Supported through persistence masks and thresholds.

- **[R.M0.DTX.FSYNC]** A dtx stabilization can be forced.

In the simplest way, transaction stabilization can be "forced" by waiting until it becomes stable naturally. More proactive mechanism can easily be implemented.

- **[R.M0.DTX.GROUP]** a collection of file system updates can be grouped into a distributed transaction.

Explicitly supported.

- **[R.M0.DTX.ISOLATION]** a degree of isolation can be specified for a distributed transaction.

Achieved by resource manager.

- **[R.M0.DTX.MULTIPLE]** a distributed transaction can contain updates for multiple objects both data and meta-data.

Explicitly supported.

- **[R.M0.DTX.SERIALIZABLE]** Dtxs are "weakly serializable": all conflicting updates from a pair of dtxs are applied in the same order.

Achieved by resource manager.

- **[R.M0.DTX.VARIABILITY]** DTM transparently takes advantage of new facilities (e.g., of a new fast logging device) or adapts when facilities are removed.

Achieved by the separation between DTM nucleus that manages transaction state and the rest of DTM that configures nucleus to use available facilities.

- **[r.m0.dtx.clovis]** Clovis transactions are directly implementable on top of DTM.
- **[r.m0.dtx.large]** Large operations are supported through undo-redo recovery. Small operations can use redo-only recovery (redo-tail).
- Standard scenarios **[r.m0.dtx.local-operation]**, **[r.m0.dtx.local-distributed]**, **[r.m0.dtx.application-domain]** and **[r.m0.dtx.general]** are explicitly supported as described in the [Undo closure](#) and [Stability detector](#) sub-sections.
- **[r.m0.dtx.pre-established]** A participant in a pre-established transactions knows all other participants and knows the transaction identity. Service DTM instances which execute updates from pre-established transactions notify all participants about persistency updates.

5.2. Dependencies

[This sub-section enumerates other system and external components the component depends on. For every dependency a type of the dependency (uses, generalizes, etc.) must be specified together with the particular properties (requirements, invariants) the design depends upon. This section is part of a requirements tracking mechanism.]

- HA
 - [u.m0.ha.scalable-broadcast]
- FOL and local transactions
 - [u.m0.txn.ordered]
 - [u.M0.FOL ST]
 - [u.M0.FOL.REDO ST], [u.M0.FOL.UNDO ST]
 - [u.M0.FOL.EPOCHS ST]
 - [u.M0.BACK-END.COMMIT.CALL-BACK ST]
- DTM users
 - [u.m0.dtx.ordered-update]

5.3. Security model

[The security model, if any, is described here.]

DTM relies on other sub-systems for authorisation and authentication. Malign servers can be detected and isolated by using replication and quorums. Malign clients (sending partial or inconsistent transactions) can be confined by eviction and cryptographically signed capabilities.

5.4. Refinement

[This sub-section enumerates design level requirements introduced by the design. These requirements are used as input requirements for the detailed level design of the component. This sub-section is part of a requirements tracking mechanism.]

The refinement will be done as part of DLD.

The following changes to the design are made on the DLD level:

- to improve concurrency, an intermediate PREPARE state is added between FUTURE and INPROGRESS states. PREPARE state is used, for example, to take locks necessary to order operation execution. Operations in INPROGRESS state are executed concurrently as far as DTM is concerned (locks taken in PREPARE state should be sufficient for concurrency control).

6. State

[This section describes the additions or modifications to the system state (persistent, volatile) introduced by the component. As much of component behavior from the logical specification should be described as state machines as possible. The following sub-sections are repeated for every state machine.]

See the [Analysis](#) section for states, transitions and invariants.

6.1. Concurrency control

[This sub-section describes what forms of concurrent access are possible and what forms on concurrency control (locking, queuing, etc.) are used to maintain consistency.]

Concurrency control will be described in the DLD.

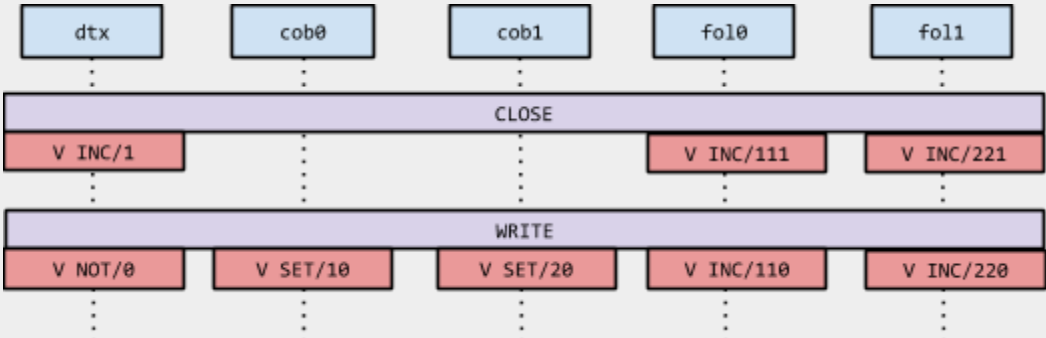
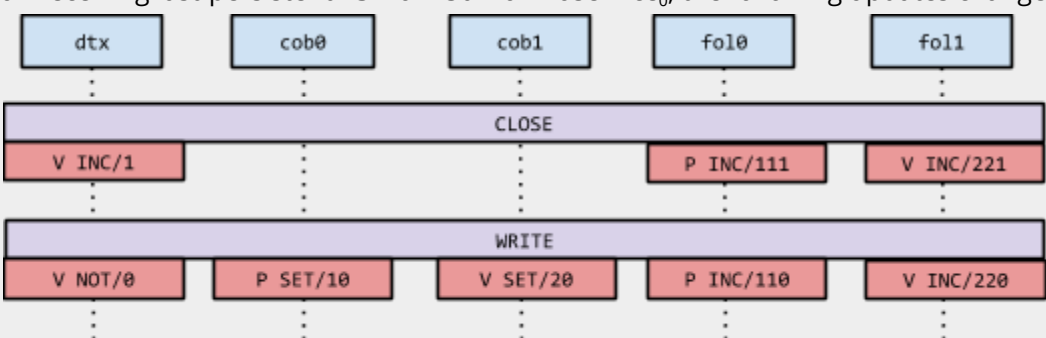
7. Use cases

[This section describes how the component interacts with rest of the system and with the outside world.]

7.1. Scenarios

[This sub-section enumerates important use cases (to be later used as seed scenarios for ARID) and describes them in terms of logical specification.]

Scenario	[usecase.dtm.wbc]
Quality attributes	scalability
Stimulus	distributed transaction containing a single operation: write to a parity de-clustered file
Stimulus source	Mero client
Environment	Mero client caching file data and ioservices hosting file's cobs.
Artifact	<p>The operation in the client's DTM nucleus looks like (cob₀: [FUTURE, SET, verno₀], cob₁: [FUTURE, SET, verno₁], ..., cob_n: [FUTURE, SET, verno_n], fol₀: [FUTURE, ANY, 0], fol₁: [FUTURE, ANY, 0], ...) where cob_i-s are shared histories of component objects. Client sends updates to each ioservice. The re-integration message has a form: ([update-id, SET, verno, service-id, cob-id, offset, length, data], [update-id, service-id], [update-id, service-id], ...)</p> <p>where first is the detailed update for the target ioservice and the rest are descriptive sibling updates. "data" are passed separately through bulk mechanism. verno contains new versions of data pages. Some flavours of read-modify-write for parity de-clustered layout can use INC update rule instead of SET (in a case where the data or parity difference is sent).</p> <pre>sequenceDiagram participant dtx participant cob0 participant cob1 participant fol0 participant fol1 Note over dtx, cob0, cob1, fol0, fol1: CLOSE dtx->>cob0: F INC/1 dtx->>fol0: F UNK/0 dtx->>fol1: F UNK/0 Note over dtx, cob0, cob1, fol0, fol1: WRITE dtx->>cob0: F NOT/0 dtx->>cob1: F SET/10 dtx->>fol0: F SET/20 dtx->>fol1: F UNK/0</pre>
Response	<p>the service, on executing the detailed update, sends back a reply message: (update-id, rc, LSN) where "rc" is the return code and "LSN" is the FOL version assigned to the operation. The</p>

	<p>client, on receiving the reply, modifies the original operation by adding the update to the shared ioservice FOL history (with LSN as the version) to the operation. Eventually, when all replies are received, the operation on the client looks like (cob₀: [VOLATILE, SET, verno], cob₁: [VOLATILE, SET, verno], ..., fol₀: [VOLATILE, SET, LSN], fol₁: [VOLATILE, SET, LSN], ...)</p>  <p>When the update becomes persistent on an ioservice, the service sends back to the client persistency notification message of the form (last-persistent-LSN). The client goes through the corresponding fol history and marks as persistent all updates in this history with LSNs not greater than the last persistent and sibling cob updates from the same operations. <i>E.g.</i>, on receiving last-persistent-LSN of 150 from ioservice₀, the following updates change state:</p> 
Response measure	
Questions and issues	Instead of FOL and LSN a monotone clock can be used.

Scenario	[usecase.dtm.intent]
Quality attributes	scalability
Stimulus	a distributed transaction with a single operation, write to a parity de-clustered file
Stimulus source	a Mero client
Environment	a Mero client without data cache and ioservices hosting the file cob-s
Artifact	The client uses a slot, that is a special object shared with the remote service and accessible only to the client and the service. The slot is effectively cached by the client and is used to

	<p>order the operations. Using the [usecase.dtm.wbc] example, the original nucleus on the client would look as</p> <p>Here, slot₀ is used to communicate with ioservice₀ and slot₁ with ioservice₁. Note that updates to cob histories use UNK rule, because the client doesn't cache the data and cannot assign the version numbers to the updates.</p>
Response	<p>When ioservice is executing an update, it assigns rules and version numbers to all UNK updates and sends these data with the reply:</p> <p>Persistence notification happens as in the previous case: through FOL LSN.</p>
Response measure	
Questions and issues	

Scenario	[usecase.dtm.pre-arranged]
Quality attributes	usability
Stimulus	
Stimulus source	
Environment	multiple DTM users coöperatively participate in the same distributed transaction, without addiiothnal communication
Artifact	multiple DTM users, say, SNS repair copy machine replicas, re-integrate operations to multiple services. These operations together constitute a distributed transaction. Each user participating in such pre-arranged transaction must in advance know identities of all other participants and identities of involved services. For example, in the case of SNS repair, a distributed transaction consists of writes to one of more re-constructed units in a parity group. Multiple replicas contribute data toward this reconstruction. Each replica knows the identities of other replicas contributing data for the transaction and the identities of the services from file layout information.
Response	On receiving an update, together with transaction description, including identities of all other participants, a service records this information in distributed transaction history. When distributed transaction persistency changes, the service propagates the notification

	to all participants.
Response measure	
Questions and issues	

Scenario	[usecase.dtm.lost]
Quality attributes	fault tolerance
Stimulus	a network packet carrying a re-integration message is lost
Stimulus source	network failure
Environment	transient network failure
Artifact	sender RPC resends the message after a timeout, until the reply is received
Response	eventually the message is received and executed. If message is resent after that point, due to a lost reply, the receiver is able to detect this by using version numbers (see [u.m0.dtx.ordered-update]).
Response measure	
Questions and issues	

Scenario	[usecase.dtm.duplicate]
Quality attributes	fault tolerance
Stimulus	a network packet carrying a re-integration message is duplicated
Stimulus source	network failure
Environment	transient network failure
Artifact	receiver detects duplication by analysing version numbers (see [u.m0.dtx.ordered-update])
Response	update is executed at most once
Response measure	
Questions and issues	

Scenario	[usecase.dtm.re-order]
----------	------------------------

Quality attributes	fault tolerance
Stimulus	network packets carrying re-integration messages, possibly from different senders, are lost
Stimulus source	network failure
Environment	transient network failure
Artifact	
Response	suppose that updates that logically should be executed in the order U0 then U1 are received in the opposite order. On receiving U1, the DTM instance would queue it instead of scheduling immediately for the execution (see [u.m0.dtx.ordered-update]). When U0 is received, it plugs the gap in the version number sequences and is immediately scheduled for execution. After U0's execution completes, U1 is ready for execution.
Response measure	
Questions and issues	

Scenario	[usecase.dtm.redo]
Quality attributes	fault tolerance
Stimulus	a DTM instance with persistent store fails and re-starts
Stimulus source	hardware failure
Environment	single failure
Artifact	on restart DTM instance notifies all other DTM instances with which it shares histories about the last survived version of each history. This can be done by sending a single number: last LSN in the FOL to all other DTM instances.
Response	On receiving such notification, a DTM instance scans its histories changes status of all lost updates to FUTURE. These updates are re-integrated as usual.
Response measure	
Questions and issues	

Scenario	[usecase.dtm.redo-lost]
Quality attributes	fault tolerance
Stimulus	a DTM instance with persistent store fails and re-starts, in addition, a re-integration message reply sent by the failed instance was lost by network

Stimulus source	hardware failure
Environment	double failure
Artifact	on restart DTM instance notifies all other DTM instances with which it shares histories about the last survived version of each history. This can be done by sending a single number: last LSN in the FOL to all other DTM instances.
Response	On receiving such notification, a DTM instance scans its histories changes status of all lost updates to FUTURE. These updates are re-integrated as usual. The update for which the reply was lost, is also moved from INPROGRESS to FUTURE state. This has to be done separately, because in the absence of the reply the update cannot be ordered by LSN. When this update is re-integrated, the receiving DTM instance can properly identify it by using version numbers and its result flag (the latter is false the update with lost reply and true for the updates which received replies).
Response measure	
Questions and issues	

[[UML use case diagram](#) can be used to describe a use case.]

7.2. Failures

[This sub-section defines relevant failures and reaction to them. Invariants maintained across the failures must be clearly stated. Reaction to [Byzantine failures](#) (*i.e.*, failures where a compromised component acts to invalidate system integrity) is described here.]

This document is all about failures and their handling.

8. Analysis

The standard way to precisely describe and analyse system behaviour is by presenting it as a state machine.

As an example, consider the process of re-integration of a single update. The state of re-integration can be described by the following Boolean attributes:

S: the update re-integration message was sent at least once;

D: the update re-integration message was delivered at least once;

E: the update was executed by the receiver and this execution is reflected in the history and object state;

R: reply was delivered;
P: update is persistent;
A: the update re-integration message was re-sent after a timeout (at least once).

This gives the following transition table for the non-deterministic state machine:

SDERPA		SDERPA	
(----0-)	-> s(u)	(1-----)	; send for the first time or after failure
(1-----)	-> d(u)	(11-----)	; deliver update, maybe multiple times
(-10---)	-> e(u)	(-11---)	; execute delivered update
(--1---	-> d(r)	(--11--)	; deliver reply, maybe multiple times
(--1-0-)	-> p(u)	(--1-1-)	; update becomes persistent
(1--0--)	-> s(u)	(1--0-1)	; resend on timeout
(--1-0-)	-> c	(--0-0-)	; volatile update is lost in crash

The transition table is given as a set of (source mask) -> event (target mask) rules. Source mask defines the set of states where the transition is possible. '-' means any value of the corresponding Boolean attribute. If an attribute is '-' in both source and target masks, its value doesn't change during transition.

Events in this state machine are:

- s(x): sending message x, where x is either update re-integration (u) or reply (r);
- d(x): delivery of message x to the peer;
- e(u): execution of the update;
- p(u): update becomes persistent;
- c: receiver crashes, losing the update.

It's easy to see that the state machine represents an unreliable network that can lose and duplicate messages and a very simple "rpc" layer on top of the network, that repeatedly re-sends the message after a timeout.

Dually, the table above can be thought of as a grammar describing the language of possible event sequences, *e.g.*,

s(u) d(u) e(u) d(r) c s(u) s(u) s(u) ...

While It is possible to formally reason about the properties of the state machine

and corresponding language, the number of states grows very rapidly when a larger fragment of the system, including multiple histories with multiple updates and version numbers, is formalised.

Instead of using a state machine formalism, the DTM and its environment (stores and network) are modelled as a collection of communicating concurrent programs in a simple imperative language. See [this reference](#) for an accessible introduction to this method. The model is in a [separate document](#). *A fortiori*, this model can be used to simulate DTM.

10. References

[References to all external documents (specifications, architecture and requirements documents, etc.) are placed here.]

- See the DTM section in the [Mero reading list](#);
- [High level design of version numbers](#);
- [Mero Detailed QAS](#);
- [Recovery presentation](#);
- [Mero architecture changes overview](#)
- [HLD of FOL](#)
- [FOL overview](#)
- [Report: distributed transactions stabilization \(epochs\)](#)
- [\[Lustre-devel\] global epochs \[an alternative proposal, long and dry\]](#)
- [\[Lustre\] Meta-data Write-back Cache](#)
- [Verifying Concurrent Processes Using Temporal Logic](#) (a downloadable version of the book can be found with enough diligence)

[HLD of distributed transaction manager](#)

[Introduction](#)

[Definitions](#)

[2. Requirements](#)

[Standard scenarios](#)

[3. Design highlights](#)

[4. Functional specification](#)

[5. Logical specification](#)

[Nucleus](#)

[Updates](#)

- [Nuclear interface](#)
- [DTM and nucleus](#)
- [Service](#)
- [Client](#)
- [Persistent format](#)
- [Shared histories](#)
- [Network failures](#)
- [Node failures](#)
- [Redo](#)
- [Undo closure](#)
- [Epochs](#)
- [Stability detector](#)
- [5.1. Conformance](#)
- [5.2. Dependencies](#)
- [5.3. Security model](#)
- [5.4. Refinement](#)
- [6. State](#)
 - [6.1. Concurrency control](#)
- [7. Use cases](#)
 - [7.1. Scenarios](#)
 - [7.2. Failures](#)
- [8. Analysis](#)
- [10. References](#)