SAND2016-5976R

Specification of Fenix MPI Fault Tolerance library

version 0.9

Marc Gamell, Rutgers Discovery Informatics Institute
Rob F. Van der Wijngaart, Intel Corporation
Keita Teranishi, Sandia National Laboratories
Manish Parashar, Rutgers Discovery Informatics Institute

June 21, 2016

Contents

1 Introduction

Fenix is a software library compatible with the Message Passing Interface (MPI) to support fault recovery without application shutdown.

Current implementation
This specification is derived from a current implementation of Fenix [1] that employs the User Level Fault Mitigation (ULFM) MPI fault tolerance module proposal. We only present the C library interface for Fenix; the Fortran interface will be added once the C version is complete.
End current implementation

Fenix is used (1) to repair communicators whose ranks suffered failure detected by the MPI runtime, and (2) to restore state to application variables and arrays from redundant data storage. Only communicators derived from the communicator returned by `Fenit_Init` are eligible for reconstruction. After communicators have been repaired, they contain the same number of ranks as before the triggering failure occurred, unless the user did not allocate sufficient redundant resources (*spare ranks*) and also did not instruct Fenix to create new ranks. In this case communicators will still be repaired, but will contain fewer ranks than before the failure occurred.

Fenix provides its own redundant data storage API to facilitate data recovery along with process recovery, but the user can choose other data recovery options to meet a variety of application needs. For example, data could be recovered by approximately interpolating values from unaffected, topologically neighboring ranks instead of by reading stored redundant data. In addition, the user may decide to use external libraries such as GVR (Global View Resilience [2]) or SCR (Scalable Checkpoint/Restart [3]) to restore rank data after a failure. The crux is that the program does not have to be restarted completely.

Current implementation
Fenix uses MPI's PMPI profiling interface. This currently means that it is incompatible with other software tools that need access to the profiling interface as well. It is expected that this restriction will be lifted soon via MPI extensions similar to that proposed by Schulz and De Supinski [4].
End current implementation

We will indicate for each library function argument whether it provides an input value (i.e. it is read by the function), an output value (i.e. it is set by the function), or both, using [IN], [OUT], and [INOUT], respectively. If a parameter is an opaque data type accessed by a handle and the handle itself is not changed by a Fenix function, but the contents of the data type may be, we still label the parameter as [INOUT], in keeping with the MPI specification.

Any Fenix function without a return type, e.g. `Fenix_Init`, may be implemented via macros, in which case it cannot be used to resolve function pointers. It is up to the implementation to decide which functions are macros.

Any Fenix function with optional parameters can be used as follows. Assume a function `f(a,b,c,d)`, with `a` being a mandatory parameter and `b`, `c`, and `d` being optional parameters can be called using only the following four combinations: `f(a)`, `f(a,b)`, `f(a,b,c)`, and `f(a,b,c,d)`. If only `a` and `c` are required by the application, the user will have to fill `b` with its default value.

2 Initialization, Rank Failure Recovery, and Teardown

2.1 Initialization

Fenix Init *(collective operation)*

```
void Fenix_Init(
        int *status,
        MPI_Comm comm,
        MPI_Comm *newcomm,
        int *argc,
        char ***argv,
        int spare_ranks,
        int spawn,
        MPI_Info info,
        int *error);
```

This function must be called by all ranks in `comm` after `MPI_Init` or `MPI_Init_thread`. All calling ranks must pass the same values for the parameters `comm`, `spare_ranks`, and `spawn`. This function is used (1) to activate the Fenix library, (2) to specify extra resources in case of rank failure, and (3) to create a logical resume point in case of rank failure.

The program may rely on the state of any variables defined and set before the call to `Fenix_Init`. But note that the code executed before `Fenix_Init` is executed by all ranks in the system (including spare ranks).

If any Fenix Data group (see Section 3) instances were created in the program following `Fenix_Init`, recovered ranks that experienced the failure, as well as surviving ranks, may be supplied with data from a valid and consistent state taken before the failure occurred. This behavior is controlled by the user. It is recommended to access `argc` and `argv` only after executing `Fenix_Init`, since command line arguments passed to this function that apply to Fenix may be stripped off by `Fenix_Init`.

- `status` [OUT] - upon return, contains one of the following values, indicating the current status of the calling rank:
  - `FENIX_STATUS_INITIAL_RANK` - this is the value returned to all ranks the first time the program is started (i.e. when the user invokes a program manager, e.g. mpirun, to launch it, not when individual ranks are reconstructed by Fenix to recover from a failure).

  - `FENIX_STATUS_RECOVERED_RANK` - this rank replaces a failed rank since the latest communicator restoration by Fenix. The rank was taken either from the pool of existing spare ranks managed by Fenix, or was newly created by Fenix using `MPI_Comm_spawn`.
  - `FENIX_STATUS_SURVIVOR_RANK` - this rank was not affected by the rank failure that triggered the latest communicator restoration by Fenix.

  The status parameter always indicates the status of a rank since the last exit from `Fenix_Init`. For example, assume a certain rank receives recovered status due to a failure. If it survives a subsequent failure, the `status` output parameter will indicate that this rank is now a survivor rank.

- `comm` [IN] - communicator that includes any spare ranks the user deems necessary. It will be used by Fenix to derive a new communicator that can be repaired. `MPI_COMM_WORLD` is a valid value for `comm`.

- `newcomm` [OUT] - Output communicator to be used by the application instead of `comm`. Let the number of ranks in `comm` be C, the number of spare ranks S, and the number of failed ranks thus far F ($F = 0$ for the first invocation of `Fenix_Init`). Upon exit from `Fenix_Init` `newcomm` contains:

  - $(C - S)$ ranks if `spawn = true`, and
  - $(C - S) - \max(F - S, 0)$ ranks if `spawn = false`.

  Ranks in `newcomm` are assigned in the same order as in `comm`. To enable successful recovery from failures via Fenix, the user should derive subsequent communicators only from `newcomm`.

- `argc` [INOUT] - pointer to the number of arguments provided by the `argc` argument to main, or NULL.

- `argv` [INOUT] - pointer to the argument vector provided by the `argv` argument to main, or NULL.

- `spare_ranks` [IN] - the number of ranks initially in `comm` that are exempted by Fenix in the construction of `newcomm`. These ranks are kept in reserve to substitute for failed ranks. When all spare ranks have been depleted and an additional failure occurs, Fenix will attempt to restore communicators to their original size by creating new ranks (e.g. using `MPI_Comm_spawn`), provided that spawning has explicitly been enabled with `spawn = true`. If this attempt fails, behavior is undefined and may result in all ranks aborting execution. If spawning is not enabled and spare ranks have been depleted, Fenix will repair communicators by shrinking them and will report such shrinkage in the `error` return parameter.

Ranks to be used as spare ranks by Fenix will be available to the application only before `Fenix_Init`, or after they are used to replace a failed rank. This document refers to the latter as *recovered* ranks.

Note that all spare ranks that have not been used to recover from failures (and, therefore, are still reserved by Fenix and kept inside `Fenix_Init`) will call `MPI_Finalize` and exit when all active ranks have entered the `Fenix_Finalize` call.

- `spawn` [IN] - used to determine whether Fenix should attempt to spawn new ranks or not.
  - If `spawn = false`, once spare ranks have been depleted, no new ranks will be spawned to fill out original communicators. Subsequent failures will be resolved by Fenix by "compacting" survivor ranks within their respective communicators, such that they retain the same order as before the failure, but they are numbered successively within the shrunk communicator.

    Note that this mode, in combination with requesting no spare ranks, can be used to obtain a shrinking communicator repair mechanism.
  - If `spawn = true`, once spare ranks have been depleted, new ranks will be spawned to fill out original communicators.
- `info` [IN] - a set of key-value pairs to further modify Fenix's process recovery behavior. Both key and value are strings, i.e. `null`-terminated `char *`. The application may pass `MPI_INFO_NULL` to indicate default behavior.

  At least the `"resume_mode"` key must be recognized by the Fenix implementation. This key is used to indicate where execution should resume upon rank failure (all active (non-spare) ranks in `newcomm` and in all of its derived communicators, not only those in communicators that failed). The following values associated to the `"resume_mode"` key must be supported.
  - `"fenix_init"` - execution resumes at logical exit of `Fenix_Init`.

  If Fenix uses `MPI_Comm_spawn` to spawn new processes (enabled by `spawn = true`), the library may include the entire key-value dictionary of the `info` parameter of `Fenix_Init` in the `info` parameter of `MPI_Comm_spawn`.
- `error` [OUT] - used to signal that a non-fatal error or special condition was encountered in the execution of `Fenix_Init`, or `FENIX_SUCCESS` otherwise. An example of such a condition is a communicator repair after all spare ranks have been depleted under a no spawning policy (i.e. `FENIX_WARNING_SPARE_RANKS_DEPLETED`).

`Fenix_Init` is called exactly once in a program. Spare ranks are not released from `Fenix_Init` until they have been used by Fenix to repair damaged communicators, or until `Fenix_Finalize` has been called by the active ranks (at which time remaining spare ranks automatically call `MPI_Finalize` and exit).

When a failure occurs and is recovered by Fenix, surviving ranks resume execution returning from `Fenix_Init` (or elsewhere depending on the `"resume_mode"` key in `info`). Replacement ranks that are created using `MPI_Comm_spawn` (invoked by the library once the spare ranks have been depleted, subject to the rank repair policy specified by the user) start execution at the lexical top of the program, including `MPI_Init` and `Fenix_Init` and any preceding statements. Consequently, spawned replacement ranks experience another control flow than survivor ranks or spare ranks, which may affect the correctness of MPI calls placed before `Fenix_Init`, especially collectives. It is the user's responsibility to avoid such problems.

Current implementation
Rank spawning in response to a failure is currently not supported.
End current implementation

No Fenix functions may be called before `Fenix_Init`, except `Fenix_Initialized`.

Fenix Initialized

```
int Fenix_Initialized(
        int *flag);
```

- `flag` [OUT] - true if `Fenix_Init` has been called and false otherwise.

2.2 Callback handler function recovery

Fenix Callback register

```
int Fenix_Callback_register(
        void (*recover)(int, MPI_Comm, int, void*),
        void *callback_data);
```

This function registers a callback to be invoked after a failure has been recovered by Fenix, and right before resuming application execution (e.g. returning from `Fenix_Init` by default). If this function is called more than once, the different callbacks registered will be called in the same order they were registered.

`Fenix_Callback_register` does not need to be called collectively. Callbacks will only be invoked by survivor ranks, since spare ranks or respawned ranks had no way to register them before a failure: they only execute code *after* `Fenix_Init` once the Fenix recovery procedure (which includes calling all registered callback functions) is completely finished.

`FENIX_ERROR_CALLBACK_NOT_REGISTERED` will be returned if there is an error while trying to register the callback function.

- `recover` [IN] - the callback function to be registered.
- `callback_data` [IN] - a pointer to application-specific data to be passed as the last parameter when calling the callback. Note that `NULL` is an acceptable value.

If a callback returns, Fenix will consider that no error occurred within the callback. If an error occurs, therefore, it needs to be either solved within the callback or escalated by using mechanisms such as `Fenix_Comm_invalidate` or `MPI_Abort`. Callback functions need to follow the following prototype:

```
void my_recover_callback(
        int status,
        MPI_Comm newcomm,
        int error,
        void *callback_data);
```

- `status` [IN] - contains the status of the rank in which this callback is called. All ranks in which a callback is called may only have a status equivalent to `FENIX_STATUS_SURVIVOR_RANK` See Section 2.1 for more details.
- `newcomm` [IN] - contains the resilient communicator returned by `Fenix_Init`.
- `error` [IN] - indicates any error that may have occurred during the recovery process. See Section 2.1 for more details.
- `callback_data` [IN] - contains the pointer passed when registering the callback (last parameter of `Fenix_Callback_register`). Note that this may be `NULL`.

2.3 Proactive rank removal

Fenix Comm invalidate *(collective operation)*

```
int Fenix_Comm_invalidate(
        MPI_Comm *comm,
        int mask);
```

This function must be called by all ranks in `comm` and informs the library that certain ranks within a communicator should be removed from the program execution. It can be used to remove ranks proactively before they experience a fatal error. It must be invoked with a resilient communicator managed by Fenix, or with `MPI_COMM_SELF`. All calling ranks must pass the same value for the parameter `comm`.

- `comm` [IN] - communicator whose rank(s) are slated for removal.
- `mask` [IN] - if non-zero, the calling rank will be removed.

2.4 Teardown

Fenix Finalize *(collective operation)*

```
int Fenix_Finalize(void);
```

This function must be called by all ranks in `newcomm` and cleans up all Fenix state, if any. If an MPI program using the Fenix library terminates normally (i.e., not due to a call to `MPI_Abort`, or an unrecoverable error) then each rank in the resilient communicator, `newcomm`, returned by `Fenix_Init` must call `Fenix_Finalize` before it exits. It must be called before `MPI_Finalize`, and after `Fenix_Init`. There shall be no Fenix calls after this function, except `Fenix_Initialized`.

As `Fenix_Init` notes, all spare ranks that have not been used to recover from failures (and, therefore, are still reserved by Fenix and kept inside `Fenix_Init`) will call `MPI_Finalize` and exit when all active ranks have called `Fenix_Finalize`.

3 Data Storage and Recovery

3.1 Overview

Fenix provides options for redundant storage of application data to facilitate application data recovery in a transparent manner. The library contains functions to control consistency of collections of such data, as well as their level of persistence. Functions with the prefix `Fenix_Data_` perform store, versioning, restore and other relevant operations and form the Fenix data recovery API. The user can select a specific set of application data, identified by its location in memory, label it with `Fenix_Data_member_create`, and copy it into Fenix's redundant storage space through `Fenix_Data_member_(I)store(v)` at a certain point in time. Subsequently, `Fenix_Data_commit` assures the consistency and the status of preceding `Fenix_Data_member_store` calls across MPI ranks, marking the data as consistent and, therefore, recoverable after a loss of ranks. Individual pieces of data can then be restored whenever they are needed with `Fenix_Data_member_(i)restore`, for example after a failure occurs. We note that the library's data storage and recovery facility aims primarily to support in-memory recovery.

Populating redundant data storage using Fenix may involve dispersion of data created by one rank to other ranks within the system (see e.g. [1]), making the store operation semantically a collective operation. However, Fenix does not require store and restore operations to be globally synchronizing. For example, execution of `Fenix_Data_member_store` for a particular collection of data could potentially be finished in some ranks, but not yet in others. And if certain ranks nominally participating in the storage operation have no actual data movement responsibility, the library is allowed to let them exit the operation immediately.

Consequently, Fenix data storage and retrieval functions should not be used for synchronization purposes.

Multiple distinct pieces (members) of data assigned to Fenix-managed redundant storage, can be associated with a specific instance of a Fenix *data group* to form a semantic unit. Committing such a group ensures that the data involved is available for recovery.

Appendix presents a diagram representing the data-centric view of the Data Storage and Recovery Fenix Interface.

3.2 Managing data storage and recovery constructs

3.2.1 Grouping data objects and ranks with *data groups*

A *Fenix data group* provides dual functionality. First, it serves as a container for a set of data objects (*members*) that are committed together, and hence provides transaction semantics. Second, it recognizes that `Fenix_Data_member_store` is an operation carried out collectively by groups of ranks, but not necessarily by all active ranks in the MPI environment. Hence, it adopts the convenient MPI vehicle of *communicators* to indicate the subset of ranks involved.

An instantiation of a data group is obtained with the following function.

Fenix Data group create *(collective operation)*

```
int Fenix_Data_group_create(
        int group_id,
        MPI_Comm comm,
        int start_time_stamp,
        int depth);
```

This function must be called by all ranks in `comm`. All calling ranks must pass the same values for all parameters.

- `group_id` [IN] - identifier of the group, unique among all active MPI ranks. If a group with this `group_id` was already created in the past and has not been deleted, the `start_time_stamp` and `depth` parameters of this invocation will be ignored, since Fenix automatically determines the correct values based on the previous invocation. The recreated group will logically be the same as the one previously in existence.

  Note that `group_id` functions as a handle to the group, to be used in creating data members associated with the group, storing these members, committing the group, as well as recovering data after a failure: the group is identified by `group_id` in `Fenix_Data_member_store` and `Fenix_Data_member_restore` calls, for example.

  The user-supplied `group_id` must be a nonnegative integer less than `FENIX_GROUP_ID_MAX`, with the latter value guaranteed to be at least 2^{30}.

- `comm` [IN] - all ranks inside this communicator need to call this function at the same logical time. They all participate as a logical unit in the storage and recovery of the data stored by the corresponding `Fenix_Data_member_store` call. `comm` should be a resilient communicator managed by Fenix, or derived from a communicator managed by Fenix (i.e. `newcomm` output parameter of `Fenix_Init`).

- `start_time_stamp` [IN] - subsequent commits related to this group have a sequence number that uniquely identifies the commit within this group. This unique logical sequence number is called the commit time stamp. The `start_time_stamp` is the sequence number of the first commit to be written to this group and can be defined by the user (for example, set to zero); this identifier will be incremented by one unit automatically by Fenix each time this group is committed.

  The user-supplied `start_time_stamp` must be a nonnegative integer less than `FENIX_TIME_STAMP_MAX`, with the latter value guaranteed to be at least 2^{30}.

- `depth` [IN] - the number of successive consistent commits (see `Fenix_Data_commit`) of this group whose associated members are retained by Fenix, in addition to the last one, and can be recovered by calling `Fenix_Data_member_(i)restore`. For example, a depth of 0 means Fenix will keep only the necessary members to restore the most recent consistent commit, while it will mark the previous ones for deletion. The data not necessary to restore older commit calls may be removed by Fenix. A depth of -1 means Fenix will not remove any committed data automatically.

The size and layout of the chosen communicator may affect the level of Fenix' fault tolerance capability.

Current implementation
Specifically, if the buddy rank mechanism is used for redundant data storage (the default method, see [1]), there have to be at least two ranks in the communicator to be able to recover data after a rank failure. However, if these ranks are collocated on the same processor or within the same node, they are more likely to fail together than if they are located on different nodes. In general, the communicator should be chosen such that it is possible to define a buddy rank that is outside the expected failure envelope of the rank that created the data to be stored.
End current implementation

The rationale behind the use of the `group_id` parameter is that it allows Fenix to cache information about the group and use that at a later time. After a loss of ranks, replacement ranks would not know about the group itself, but given that label `group_id` is a value set and known by the application, the application can query Fenix to retrieve the cached information and use it to reconstruct the group logically. The ranks can then use the group to retrieve redundantly stored application data.

The predefined constant `FENIX_DATA_GROUP_WORLD_ID` constitutes a `group_id` as if created by calling:

```
Fenix_Data_group_create(
        FENIX_DATA_GROUP_WORLD_ID, // group_id
        newcomm,                   // communicator
        0,                         // start_time_stamp
        0);                        // depth
```

where `newcomm` is the communicator returned by the last time `Fenix_Init` returned. In other words, this is a convenient constant to represent all ranks returned by `Fenix_Init` via a reserved `group_id`, an initial time stamp of zero, and garbage collection depth of zero (i.e. Fenix will keep only the last consistent commit).

Applications that do not need the flexibility of the more generic Fenix grouping mechanism can, therefore, avoid having to create a specific group and can use this generic group instead.

A Fenix data group can be deleted using the following functions. Along with the group, any application data associated with the group (see section 3.4.1) will also be deleted. Because this may take significant time, an asynchronous version is included.

Fenix Data group delete *(collective operation)*

```
int Fenix_Data_group_delete(
        int group_id);
```

- `group_id` [IN] - id of the group to be destroyed.

This function must be called by all ranks in the communicator `comm` associated with the group identified by `group_id`. All calling ranks must pass the same value for the parameter. When a data group is no longer needed, its resources can be released (and its `group_id` be made available for use in other groups) with this function. This function will recursively delete all its members and commits.

This collective operation marks the group object maintained by the library for deallocation. Any pending operations that use this group will complete normally; the object is actually deallocated only if there are no other active references to it.

Fenix Data group idelete *(collective operation)*

```
int Fenix_Data_group_idelete(
        int group_id,
        Fenix_Request *request);
```

- `request` [OUT] - handle to the asynchronous store operation.

This function has the same effect as `Fenix_Data_group_delete`, except that it returns immediately, possibly before the data or meta-data associated with the group have been deleted. The operation can be finalized by waiting on the returned request.

3.2.2 Describing application data with *data group members*

Fenix data groups are composed of members that describe the actual application data. A member joins a group with the following function.

Fenix Data member create *(collective operation)*

```
int Fenix_Data_member_create(
        int member_id,
        void *buffer,
        int count,
        MPI_Datatype datatype,
        int group_id);
```

This function must be called by all ranks in the communicator `comm` associated with the group identified by `group_id`. All calling ranks must pass the same values for the parameters `member_id`, `datatype`, and `group_id`.

- `member_id` [IN] - integer within the named group `group_id` that uniquely identifies the data in `buffer`.

  The user-supplied `member_id` must be a nonnegative integer less than `FENIX_MEMBER_ID_MAX`, with the latter value guaranteed to be at least 2^{30}.

- `buffer` [IN] - the address of the data to be copied to the redundant storage maintained by Fenix. Note that this parameter may also be specified using the function `Fenix_Data_member_set_attribute`. The latter is critical for non-survivor ranks (`FENIX_STATUS_RECOVERED_RANK`) after a failure. In that case data group members are *implicitly* recreated by the library when the programmer calls `Fenix_Data_group_create`, but any pointer to the application data is invalid and must be supplied explicitly by the user for each group member. Survivor ranks will use the buffer pointer specified before the failure, unless it is overwritten by `Fenix_Data_member_set_attribute`.

- `count` [IN] - maximum number of contiguous elements of type `datatype` of the data to be stored[1]. This parameter does not need to be the same in all ranks calling this function.

- `datatype` [IN] - data type of each element in buffer.

[1]To avoid problems related to using an `int` to identify sizes (such as 32-bit integers not being big enough to address all the memory, we will use `MPI_Count` once it is adopted by the MPI Forum.

- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - identifier of the group that will be associated with this member.

Fenix Data member delete *(collective operation)*

When a data group member is no longer needed, it may be deleted by the following functions. Along with the data group member, any application data associated with the member (see section 3.4.1) will also be deleted. Because this may take significant time, an asynchronous version is included.

```
int Fenix_Data_member_delete(
        int member_id,
        int group_id);
```

This function must be called by all ranks in the communicator `comm` associated with the group identified by `group_id`. All calling ranks must pass the same values for the parameters. This function marks all storage required to store data and meta-data related to `member_id` for deallocation, being the data in the calling rank, in any other rank, or in any other storage facility within the system. Any pending operations that use this member of the group will complete normally; the objects are actually deallocated only after all operations involving this member have completed.

This function needs to be called collectively, at the same logical time, by all ranks associated with the communicator used when creating `group_id`. All ranks must provide the same values for the parameters.

- `member_id` [IN] - unique integer within the named group that uniquely identifies the data in `buffer`.
- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - group id of collection of data.

Fenix Data member idelete *(collective operation)*

```
int Fenix_Data_member_idelete(
        int member_id,
        int group_id,
        Fenix_Request *request);
```

This function has the same effect as `Fenix_Data_member_delete`, except that it returns immediately, possibly before the data or meta-data associated with the member have been deleted. The operation can be finalized by waiting on the returned request. This operation has no optional parameters; all parameters are required.

- `request` [OUT] - handle to the asynchronous store operation.

Note that members can be created or deleted from a group (identified by `group_id`) at any point between the calls to `Fenix_Data_group_create` and `Fenix_Data_group_(i)delete` related to `group_id`.

3.2.3 Accessing redundancy policies

The resilience of data in Fenix' redundant data storage depends on the specified policy, which can be queried and set on a per-group basis using the following functions. This policy defines the default redundant storage policy for all group meta-data not explicitly stored in Fenix' redundant storage by the programmer, as well as for all group members' data.

Fenix Data group get redundancy policy

```
int Fenix_Data_group_get_redundancy_policy(
int group_id,
int policy_name,
void *policy_value,
int *flag);
```

This function is used to query the library for the type of policy it applies to safeguard all meta-data and application data (group members) by dispersing copies of that data.

- `group_id` [IN] - identifier of the group whose policy is sought.
- `policy_name` [IN] - name of policy whose value is sought.
- `policy_value` [OUT] - value of corresponding policy.
- `flag` [OUT] - true if a policy value was extracted; false if no policy is associated with the key.

Fenix Data group set redundancy policy *(collective operation)*

```
int Fenix_Data_group_set_redundancy_policy(
int group_id,
int policy_name,
void *policy_value,
int *flag);
```

This function must be called by all ranks in the communicator `comm` associated with the group identified by `group_id`. All calling ranks must pass the same values for the parameters `group_id`, `policy_name`, and the contents pointed by `policy_value`.

This function is used to define the type of policy the library applies to safeguard all meta-data and application data (group members) by dispersing copies of that data.

- `group_id` [IN] - identifier of the group whose policy is sought.
- `policy_name` [IN] - name of policy whose value is sought.

- `policy_value` [IN] - value of corresponding policy.
- `flag` [OUT] - true if a policy value was set; false if no policy is associated with the key, or if the policy is read-only (this could be a policy that is set at the time the library is built or initialized). Upon successful return of this function, all calling ranks are guaranteed to have the same value in the memory position pointed by `flag`.

At least the following policy name must be defined: `FENIX_DATA_POLICY_PEER_RANK_SEPARATION` which determines one of the simplest types of data redundancy, namely preserving a copy of the data on a peer rank within the same communicator used in the creation of a data group. In this case, the `policy_value` input parameter is the `rank_separation`, and has a default value equivalent to half of the size of the communicator associated with the group (size(`comm`)/2). A single copy of the data stored locally on rank `my_rank` will also be stored on rank `(my_rank+rank_separation) mod comm_size`, where `comm_size` equals the size of the communicator associated with the relevant data group. We note that depending on the layout of the ranks of the communicator across the physical resources of the system (nodes, racks, cabinets), different values of the `rank_separation` parameter should be selected to obtain the desired data resilience. For example, assuming a communicator spanning ranks mapped to nodes distributed in two physical cabinets (where ranks 0 to `cabinet_size`-1 are in one cabinet and ranks `cabinet_size` to (2*`cabinet_size`)-1 are in the other), `rank_separation` can be set to `cabinet_size` so that all stored members in the group are replicated in both cabinets.

Group redundancy policies can only be set before the first store operation of a member of `group_id`, or the first commit operation of the `group_id`. When a member is first stored or the group is first committed, group redundancy is considered frozen and cannot be changed even after a failure.

3.3 Probing and completing asynchronous operations

In many instances programmers can identify useful work to do by the application while a potentially costly Fenix operation is taking place. For this purpose Fenix supports asynchronous operations that return control to the application immediately, but that need to be probed and/or finished later. The functions needed, `Fenix_Data_wait` and `Fenix_Data_test`, are described here.

The user must always call `Fenix_Data_wait` in order to guarantee the successful completion of a non-blocking collective or non-collective operation (unless `Fenix_Data_test` returns with `flag = true`). Users should be aware that Fenix implementations are allowed, but not required, to synchronize ranks during the completion of a non-blocking collective operation.

Fenix Data wait

```
int Fenix_Data_wait(
        Fenix_Request request);
```

Waits for a non-blocking operation identified by `request`. One is allowed to call `Fenix_Data_wait` with a null or inactive request argument. In this case the operation returns immediately.

- `request` [IN] - handle to the asynchronous store operation.

Fenix Data test

```
int Fenix_Data_test(
        Fenix_Request request,
        int *flag);
```

Tests for the completion of a non-blocking operation identified by request.

- `request` [IN] - handle to the asynchronous store operation. One is allowed to call `Fenix_Data_test` with a null or already completed request. In such a case the operation returns with `flag = true`.
- `flag` [OUT] - The call returns immediately with `flag = true` if the operation is already completed. The call returns `flag = false`, otherwise.

3.4 Storing and committing application data

3.4.1 Storing group members

Fenix Data member store *(collective operation)*

```
int Fenix_Data_member_store(
        int member_id,
        int group_id,
        Fenix_Data_subset subset_specifier);
```

This function must be called by all ranks in the communicator `comm` associated with the group identified by `group_id`. All calling ranks must pass the same values for the parameters. This function is used to safeguard the data associated with a particular member of the data group. This function will place one or more copies of data residing in `buffer` (supplied in the call to the function `Fenix_Data_member_create`) in Fenix' redundant data storage.

Current implementation
After creating a local copy in memory of this member, Fenix will transfer this local copy to its final destination(s), e.g. non-volatile memory, a peer's memory, a file on a local hard disk.
End current implementation

This function may fail if not enough memory can be allocated to store data of the specified size. All ranks in the communicator that participated in `Fenix_Data_group_create` must call this function at logically the same time, with the same `member_id`.

When the call returns, the application can safely modify the data in `buffer` marked for safeguarding, since it has already been saved. The saved data, however, will only be available for recovery after being time stamped via commiting the group. This can be done using its group identifier, its member identifier, and the logical time stamp of the commit.

Multiple calls to `Fenix_Data_member_store` with the same `member_id` without intervening commits will lead to storing (parts of) the same application data object. Depending on the value of `subset_specifier`, this may lead to overwriting the data (loss of data), or incremental storage of the full data.

- `member_id` [IN] - integer label that uniquely identifies a member of the data group (see `Fenix_Data_member_create`). `FENIX_DATA_MEMBER_ALL` will store all members associated with the specified group.
- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - identifier of the group associated with this member.
- `subset_specifier` [IN] [optional; default value: `FENIX_DATA_SUBSET_FULL`] - specifier of the subset of data to be stored. The choice of this parameter needs to result in identical subsets in all ranks calling this function, which minimizes the need for the library to coordinate between the rank whose member needs to be safeguarded and the agent managing Fenix' non-local redundant data storage (which could be another rank in the system), thus resulting in performance improvement. Users are encouraged to use this function instead of `Fenix_Data_member_storev` (see below) whenever possible. When a `subset_specifier` different than `FENIX_DATA_SUBSET_FULL` is supplied, Fenix will only store the positions in the application buffer that are in the subset.

Fenix Data member storev *(collective operation)*

```
int Fenix_Data_member_storev(
        int member_id,
        int group_id,
        Fenix_Data_subset subset_specifier);
```

This function is the same as `Fenix_Data_member_store`, except that actual subsets realized by the choice of parameter `subset_specifier` can be different in different ranks.

Fenix Data member istore *(collective operation)*

```
int Fenix_Data_member_istore(
        int member_id,
        int group_id,
        Fenix_Data_subset subset_specifier,
        Fenix_Request *request);
```

This function has the same effect as `Fenix_Data_member_store`, except that it returns immediately, even before the data has been stored safely. Data in the application buffer marked for safeguarding may be overwritten once a call to `Fenix_Data_wait` on `request` has returned.

Current implementation
`Fenix_Data_member_istore` copies the application data into local memory before returning and starts the asynchronous transfer to its final destination. Therefore, in the current implementation, marked data in the application buffer may be overwritten once the call `Fenix_Data_member_istore` returns.
End current implementation

The result of multiple calls to `Fenix_Data_member_istore` with overlapping subsets and without intervening calls to `Fenix_Data_wait` is undefined.

This operation has no optional parameters; all parameters are required.

- `request` [OUT] - handle to the asynchronous store operation.

Fenix Data member istorev *(collective operation)*

```
int Fenix_Data_member_istorev(
        int member_id,
        int group_id,
        Fenix_Data_subset subset_specifier,
        Fenix_Request *request);
```

This function is the same as `Fenix_Data_member_istore`, except that actual subsets realized by the choice of parameter `count` in function `Fenix_Data_member_create` and parameter `subset_specifier` can be different in different ranks.

3.4.2 Making stored data recoverable with *data group commits*

Fenix Data commit *(collective operation)*

```
int Fenix_Data_commit(
        int *time_stamp,
        int group_id);
```

This function must be called by all ranks in the communicator `comm` associated with the group identified by `group_id`. All calling ranks must pass the same value for the parameter `group_id`. This function is used to freeze the current state of a data group, together with all its application data that has been stored in Fenix' redundant storage, and label it with a time stamp, thus creating a consistent snapshot of the stored application data. All ranks in the communicator that was used in the creation of the data group must call this function at the same logical time. Only data that has been committed is eligible for recovery through `Fenix_Data_member_restore`. An application needs to call `Fenix_Data_wait` for all pending asynchronous `Fenix_Data_member_istore` and `Fenix_Data_member_istorev` operations in the group before committing.

- `time_stamp` [OUT] [optional; default value: `NULL`] - absolute sequence number of the committed data. `NULL` is a valid parameter, in which case the automatically incremented sequence number is not returned to the application.

  The `time_stamp` parameter will be a nonnegative integer less than `FENIX_TIME_STAMP_MAX`, with the latter value guaranteed to be at least 2^{30}.

  Upon successful return of this function, all calling ranks are guaranteed to have the same value in the memory position pointed by `time_stamp`.

- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - identifier of the group to commit.

Note that not all members in the group need to be stored (with `Fenix_Data_member_store` or any other variant) in order for a commit to succeed. For example, the following scenario is valid.

```
// Create members
Fenix_Data_member_create(0, &a, 1, MPI_INT, mygroup);
Fenix_Data_member_create(1, &b, 1, MPI_INT, mygroup);
// Store members as part of commit with time stamp 0
a = myrank;
b = myrank+1;
Fenix_Data_member_store(0, mygroup);
Fenix_Data_member_store(1, mygroup);
Fenix_Data_commit(&ts, mygroup); // after this, ts=0
// Store only member 'b' for commit with time stamp 1
b = myrank+100;
Fenix_Data_member_store(1, mygroup);
Fenix_Data_commit(&ts, mygroup); // after this, ts=1
```

3.5 Recovering application data

After a failure is recovered and control is returned to the application (for example, by returning from `Fenix_Init`), the application may need to restore previously saved and committed data objects. The first step is to recreate the groups using the repaired communicators, which can be done using `Fenix_Data_group_create`, as explained in Section 3.2.1. Members, however, do not need to be recreated, since both their meta-data (in particular, the `member_id`, the `count`, and the `datatype`) and application data are saved in the redundant storage.

Fenix Data member restore *(collective operation)*

```
int Fenix_Data_member_restore(
        int member_id,
        void *data,
        int max_count,
        int time_stamp,
        int group_id);
```

This function must be called by all ranks in the communicator `comm` associated with the group identified by `group_id`. All calling ranks must pass the same values for the parameters `member_id`, `time_stamp`, and `group_id`. This function is used to retrieve explicitly stored and committed data. This function can only be used if the size of the communicator used to store the data is the same as that at the time of data recovery (this implies non-shrinking communicator recovery in case of a loss of rank).

The application will be able to recover explicitly stored and committed data group members by allocating a buffer and, using `Fenix_Data_member_restore`, requesting Fenix to fill it with the data from a particular member at a particular time stamp (commit).

If the size of the buffer to allocate is unknown for a particular rank, it can be queried by using the functions described in Section 3.7.3.

Parameters:

- `member_id` [IN] - this value must match the member id that was supplied when `Fenix_Data_member_store` was called.
- `data` [OUT] - the requested stored data will be written contiguously at this local address. If NULL, no attempt will be made to fetch and restore data. This is useful for selective recovery of application data. Each rank in the communicator associated with the data group will receive the selected data from the corresponding rank in the communicator used at the time the data was stored and committed.
- `max_count` [IN] - the requested stored data, if found, will only be recovered if its size is `max_count` times the size of `datatype` or less.
- `time_stamp` [IN] [optional; default value: `FENIX_LATEST_DATA_GROUP_COMMIT`] - the time stamp of the requested committed member. The special value of `FENIX_LATEST_DATA_GROUP_COMMIT` will always recover the latest committed data by Fenix in `group`.
- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - group that contains the requested data.

In case a member is selected for recovery in a commit that did not include that particular member, that data member will be restored using the value of the most recent commit C prior to the requested commit (`time_stamp`) that did include the member. In case the user explicitly deleted the member or commit where the loaded data was supposed to be found (C), the application data will not be restored and the contents of the memory pointed by `data` will not be modified.

The behavior when restoring members not included in a commit can be seen in lines 20 through 23 of the following scenario.

```
1  // Create members
2  Fenix_Data_member_create(0, &a, 1, MPI_INT, mygroup);
3  Fenix_Data_member_create(1, &b, 1, MPI_INT, mygroup);
4  // Store members for commit with time stamp 0
5  a = myrank;
6  b = myrank+1;
7  Fenix_Data_member_store(0, mygroup);
8  Fenix_Data_member_store(1, mygroup);
9  Fenix_Data_commit(&ts, mygroup); // after this, ts=0
10 // Store member 'b' for commit with time stamp 1
11 b = myrank+100;
12 Fenix_Data_member_store(1, mygroup);
13 Fenix_Data_commit(&ts, mygroup); // after this, ts=1
14 // Store member 'a' for commit with time stamp 2
15 a = myrank+200;
16 Fenix_Data_member_store(0, mygroup);
17 Fenix_Data_commit(&ts, mygroup); // after this, ts=2
18
19 // Restore members
20 Fenix_Data_member_restore(0, &new_a, 1, 1, mygroup);
21 // new_a now contains "myrank" (line 5)
22 Fenix_Data_member_restore(1, &new_b, 1, 1, mygroup);
23 // new_b now contains "myrank+100" (line 11)
```

When restoring a group member m with time stamp ts that contains a hole (i.e. in commit ts, a subset was used to store m that did not cover all elements of m), previous time stamps of m will be inspected, starting from $ts-1$ and working backwards, until a value v of the hole is found in $ts-i$ (being $i>0$). The hole in the buffer `data` will then be filled using v. If no value for the hole is found, the position of the hole in the application's buffer `data` will be not overwritten.

Fenix Data member irestore *(collective operation)*

```
int Fenix_Data_member_irestore(
        int member_id,
        void *data,
        int max_count,
        int time_stamp,
        int group_id,
        Fenix_Request *request);
```

This function has the same effect as `Fenix_Data_member_restore`, except that it returns immediately, possibly before the application data has been restored. Data in buffer `data` is not guaranteed to be consistent until a call to `Fenix_Data_wait` on the request has returned. This operation has no optional parameters; all parameters are required.

- `request` [OUT] - handle to the asynchronous data recovery operation.

The above functions assume that the size of the communicator used during storage of the data before the commit operation equals that present at the time of the restoration operation. All ranks within the communicator call this function, and Fenix can establish a one-to-one mapping between ranks that stored data before the commit and those that are requesting data at the time of the restoration. However, when the communicator has shrunk, such a mapping no longer exists. In this case, or in other instances in which more control is desired, the user can specify explicitly for each calling rank what is the source rank whose stored data needs to be retrieved from Fenix' redundant storage. This is accomplished by the following two functions.

Fenix Data member restore from rank *(collective operation)*

```
int Fenix_Data_member_restore_from_rank(
        int member_id,
        void *data,
        int max_count,
        int time_stamp,
        int group_id,
        int source_rank);
```

This function works the same way as `Fenix_Data_member_restore`, except that the source rank for the data to be recovered is specified explicitly.

Parameters:

- `source_rank` [IN] - specifies the rank (in the communicator associated with `group_id`) that performed the data store and whose data we are trying to recover.

Fenix Data member irestore from rank *(collective operation)*

```
int Fenix_Data_member_irestore_from_rank(
        int member_id,
        void *data,
        int max_count,
        int time_stamp,
        int group_id,
        int source_rank,
        Fenix_Request *request);
```

This function works the same way as `Fenix_Data_member_irestore`, except that the source rank for the data to be recovered is specified explicitly. This operation has no optional parameters; all parameters are required.

Parameters:

- `source_rank` [IN] - specifies the rank (in the communicator associated with `group_id`) that performed the data store and whose data we are trying to recover.

We note that these functions do not require that the communicator has shrunk, and can be used for any recovery pattern consistent with their definition, as long as the value for `source_rank` is valid.

3.6 Managing data subsets

Fenix data group members are used to provide resilient caches for sets of application data that are contiguous in memory. Each set is represented by a pair consisting of {`start_pointer`,`count`}. *Subsets* represent logical subsets of such sets. They allow the user to indicate which elements (zero or more elements between 0 and `count`) will be selected for a particular `Fenix_Data_member_store` operation or its variants (see example in Section 4.2). They provide a convenient mechanism to reduce the burstiness of data traffic to the final destination of stores (such as IO subsystems) accessed by `Fenix_Data_member_store` calls. They also provide a way to store only the elements of a group member that changed since the last commit.

An example of the usage of subsets id as follows. Assume an array of ten elements set initially to a particular set of values. An application iteratively changes the elements in the array, one element per iteration. In this scenario, the application can decide to initially store the entire array, and then, at a specific iteration, store only the changed element by selecting it with subsets.

Another example of an array in a contiguous memory layout is illustrated by Figure 1. In this example, the second and third `Fenix_Data_member_store` calls store subsets of an array by block patterns. Fenix provides a data type to allow users to define the relative location and size of individual blocks.

Current implementation
During the store call and its variants, Fenix decides how to perform the actual store, based on the data size and granularity of blocks, as well as the performance of underlying IO subsystems. See `Fenix_Data_member_store` for more details.
End current implementation

Fenix Data subset create

```
int Fenix_Data_subset_create(
        Fenix_Data_subset *subset_specifier,
        int num_blocks,
        int* array_start_offsets,
        int* array_end_offsets);
```

Creates a subset based on `num_blocks` pairs of {`start_offset`,`end_offset`}.

When applying a `Fenix_Data_subset` value to `Fenix_Data_member_store` calls, the values of `array_start_offsets` and `array_end_offsets` must be

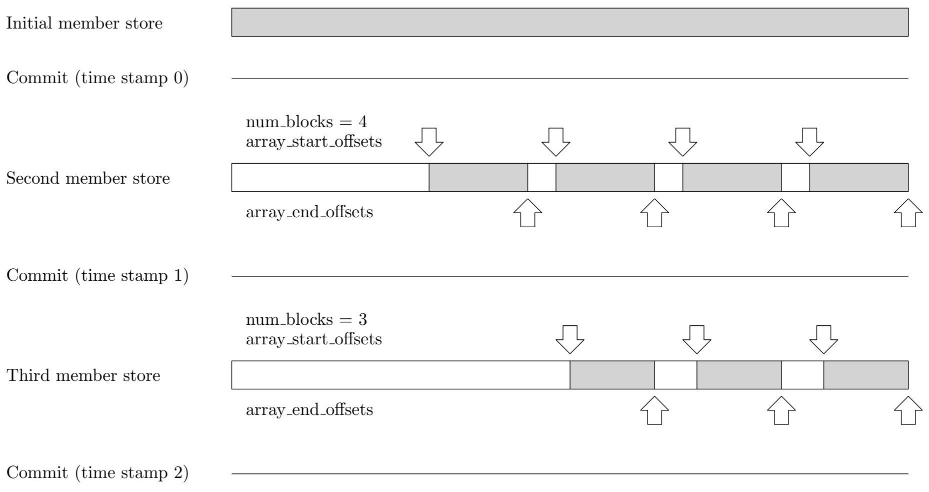


Figure 1: Incremental member store using *subsets*. Gray areas indicate the data being saved by `Fenix_Data_member_store` operations.

less than the count of the entire data object (value of `count`) defined by the corresponding `Fenix_Data_member_create` call.

- `subset_specifier` [OUT] - name of the subset specifier, to be used in storing data.
- `num_blocks` [IN] - the number of contiguous data blocks, which also defines the number of elements in `array_start_offsets` and `array_end_offsets`.
- `array_start_offsets` [IN] - an integer array, which indicates the index of the first elements for each data block (the `start_offset` in the pair {`start_offset,end_offset`}). The value indicates the number of data elements from the beginning of the data registered at `Fenix_Data_member_create`.
- `array_end_offsets` [IN] - an integer array, which indicates the index of the last element for each data block (the `end_offset` in the pair {`start_offset,end_offset`}). The value indicates the number of data elements from the beginning of the data registered at `Fenix_Data_member_create`.

The constant `FENIX_DATA_SUBSET_FULL` of type `Fenix_Data_subset` represents the default subset specifier; it selects all the data indicated by the user via the `count` parameter specified in the call to `Fenix_Data_member_create`.

Fenix Data subset delete

```
int Fenix_Data_subset_delete(
        Fenix_Data_subset *subset_specifier);
```

Deletes a previously-created subset.

- `subset_specifier` [INOUT] - name of the subset specifier, as returned by the `subset_specifier` parameter in `Fenix_Data_subset_create`. The handle is set to `FENIX_SUBSET_NULL`.

3.7 Accessing data storage and recovery constructs

These functions provide the means to access and alter the information and attributes for Fenix's data recovery and its internals. The status of individual stored objects can be queried by pointing to the corresponding Fenix data group and the `member_id`. Examples in Section 4.3 and Section 4.4 show how these functions can be used. All functions in this section have local completion semantics and do not have to be called collectively, except `Fenix_Data_member_get_attribute`.

3.7.1 Querying group members

Fenix Data group get number of members

```
int Fenix_Data_group_get_number_of_members(
        int *number_of_members,
        int group_id);
```

- `number_of_members` [OUT] - number of available distinct member of this group. Manually deleted members are not included in this number.
- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - Fenix data group whose information is sought.

Fenix Data group get member at position

```
int Fenix_Data_group_get_member_at_position(
        int position,
        int *member_id,
        int group_id);
```

- `position` [IN] - sequence number of the requested `Fenix_Data_member`. `position` must be a value between 0 and `number_of_members`-1, (`number_of_members` as returned by `Fenix_Data_group_get_number_of_members`). The member positions will be returned in the order the user added members to the Fenix data group, i.e. oldest first, newest last (e.g. the first member added by the user will have `position` 0). Deleted members will not be included in this list.
- `member_id` [OUT] - the unique identifier of the `Fenix_Data_member` sought.
- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - Fenix data group whose information is sought.

3.7.2 Querying committed data

Fenix Data group get number of commits *(collective operation)*

```
int Fenix_Data_group_get_number_of_commits(
        int *number_of_commits,
        int group_id);
```

This function must be called by all ranks in the communicator `comm` associated with the group identified by `group_id`. All calling ranks must pass the same value for the parameter `group_id`.

- `number_of_commits` [OUT] - number of available, distinct, consistent commits of this group. Deleted commits (either deleted manually or deleted through garbage collection –i.e. see `depth` of `Fenix_Data_group_create`) are not included in this number.

  Upon successful return of this function, all calling ranks are guaranteed to have the same value in the memory position pointed by `number_of_commits`.

- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - Fenix data group whose information is sought.

Fenix Data group get commit at position

```
int Fenix_Data_group_get_commit_at_position(
        int position,
        int *time_stamp,
        int group_id);
```

This function must be called after calling `Fenix_Data_group_get_number_of_commits` but before any commit is created or deleted in the group identified by `group_id`.

- `position` [IN] - sequence number of the requested `Fenix_Data_commit`. `position` must be a value between 0 and `number_of_commits`-1 (`number_of_commits` as returned by `Fenix_Data_group_get_number_of_commits`). The commit positions will be returned in the reverse order in which the user executed them, i.e. oldest last, newest first (e.g. the most recent completed commit will have `position` 0).

- `time_stamp` [OUT] - the unique identifier of the `Fenix_Data_commit` sought.

- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - Fenix data group whose information is sought.

3.7.3 Accessing group member attributes

Certain properties can be assigned to members of Fenix data groups. These properties, called attributes, can be queried and defined using the following functions.

Fenix Data member get attribute *(collective operation)*

```
int Fenix_Data_member_get_attribute(
        int member_id,
        int attribute_name,
        void *attribute_value,
        int *flag,
        int group_id,
        int source_rank);
```

This function must be called by all ranks in the communicator `comm` associated with the group identified by `group_id`. All calling ranks must pass the same values for the parameters `member_id`, `attribute_name`, and `group_id`.

Parameters:

- `member_id` [IN] - unique integer within group associated with `group_id` that identifies the data in Fenix's redundant data storage.
- `attribute_name` [IN] - name of the particular attribute, consisting of the prefix `FENIX_DATA_MEMBER_ATTRIBUTE_`, followed by a suffix. At least the following suffixes must be valid: `BUFFER`, `COUNT`, `DATATYPE`, and `SIZE`.
- `attribute_value` [OUT] - the attribute value of the particular member of the target data group.
- `flag` [OUT] - true if an attribute value was extracted; false if no attribute is associated with the key.
- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - Fenix data group whose information is sought.
- `source_rank` [IN] [optional; default value: the calling rank in the communicator associated with `group_id`] - for attributes that are rank-dependent (such as `FENIX_DATA_MEMBER_ATTRIBUTE_COUNT`), specifies the rank (in the communicator associated with `group_id`) that contains the attribute that is sought.

Fenix Data member set attribute

```
int Fenix_Data_member_set_attribute(
        int member_id,
        int attribute_name,
        void *attribute_value,
        int *flag,
        int group_id);
```

This function can be used to set an attribute related to a member. Attributes can only be set before the first store operation of `member_id` or commit operation of `group_id` that occur after returning from `Fenix_Init`. When a member is stored or a group is committed, attributes are considered frozen until the next failure occurs. After a failure, the execution will be returned from `Fenix_Init`, at which point attributes can be reset before any subsequent stores. In particular, at least the attribute `FENIX_DATA_MEMBER_ATTRIBUTE_BUFFER` must be writable after a failure is recovered.

- `member_id` [IN] - unique integer within group associated with `group_id` that identifies the data in Fenix's redundant data storage.
- `attribute_name` [IN] - name of the particular attribute. Attribute names with the suffix `COUNT` and `DATATYPE` are read only.
- `attribute_value` [IN] - the attribute value of the particular member of the target data group.
- `flag` [OUT] - true if the attribute value was set; false if no attribute is associated with the key or if the attribute is read-only.
- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - Fenix data group whose information is sought.

3.8 Removing stored application data

Data and meta-data associated with a specific commit operation can be manually marked for deletion using the following functions.

Fenix Data commit delete *(collective operation)*

```
int Fenix_Data_commit_delete(
        int time_stamp,
        int group_id);
```

This function must be called by all ranks in the communicator `comm` associated with the group identified by `group_id`. All calling ranks must pass the same values for the parameters. This function removes irretrievably the stored members and meta-data associated with a specific `Fenix_Data_commit` call. It can be used in addition to, or instead of, the implicit garbage collection that Fenix performs, which is controlled by the `depth` parameter in `Fenix_Data_group_create`.

- `time_stamp` [IN] - the time stamp of the requested commit. The special value of `FENIX_LATEST_DATA_GROUP_COMMIT` will always remove the latest data consistently committed by Fenix. The special value of `FENIX_ALL_DATA_GROUP_COMMIT` can be used to remove all data consistently committed by Fenix.

- `group_id` [IN] [optional; default value: `FENIX_DATA_GROUP_WORLD_ID`] - group whose time stamped, committed data should be removed.

Fenix Data commit idelete *(collective operation)*

```
int Fenix_Data_commit_idelete(
        int time_stamp,
        int group_id,
        Fenix_Request *request);
```

This function has the same effect as `Fenix_Data_commit_delete`, except that it returns immediately, possibly before the data has been removed. The operation can be finalized by waiting on the returned request. This operation has no optional parameters; all parameters are required.

- `request` [OUT] - handle to the asynchronous delete operation.

We note that redundant application data may also deleted as a side effect of the functions `Fenix_Data_group_(i)delete` and `Fenix_Data_member_(i)delete`. See Sections 3.2.1 and 3.2.2.

4 Examples

4.1 Protecting process and data with Fenix

This example shows two versions of the same mini-example application, a non-fault-tolerant version, and an augmented version with Fenix that tolerates failures in an on-line manner.

```
/* Non-fault-tolerant version */
int main()
{
  int it;
  int A[100], B[50];

  initialize(A, B);

  for(it=0 ; it<1000 ; it++) {
    work1(A, MPI_COMM_WORLD);
    if(A[0] > 200) {
      work2(A, B, MPI_COMM_WORLD);
    }
  }
}
```

```
/* Fault tolerant version with Fenix */
int main()
{
  int it;
  int A[100], B[50];
  int status, error;
  MPI_Comm new_comm_world;

  Fenix_Init(&status, MPI_COMM_WORLD, &new_comm_world,
             &argc, &argv,
             10, // num_spare_ranks
             0, // spawn
             MPI_INFO_NULL,
             &error);
  if( !error && status == FENIX_STATUS_INITIAL_RANK ) {
    /* no failure occurred */
    it = -1;
    initialize(A, B);
    Fenix_Data_member_create(990, &it, 1, MPI_INT);
    Fenix_Data_member_create(991, A, 100, MPI_INT);
    Fenix_Data_member_create(992, B, 50, MPI_INT);
    Fenix_Data_member_store(FENIX_DATA_MEMBER_ALL);
    Fenix_Data_commit();
  } else if(!error) {
    /* ranks recovered from a failure, now restore data */
    Fenix_Data_member_restore(990, &it, 1, FENIX_LATEST_COMMIT);
    Fenix_Data_member_restore(991, A, 100, FENIX_LATEST_COMMIT);
    Fenix_Data_member_restore(992, B, 50, FENIX_LATEST_COMMIT);
  } else {
    // There was an error in Fenix
    MPI_Abort(MPI_COMM_WORLD, -1);
  }

  for( ; it<1000 ; ) {
    it++;
    Fenix_Data_member_store(990);
    work1(A, new_comm_world);
    if(A[0] > 200) {
      work2(A, B, new_comm_world);
      Fenix_Data_member_store(992);
    }
    Fenix_Data_member_store(991);
    Fenix_Data_commit();
  }
}
```

4.2 Storing data objects with subsets

```
/* Non-fault-tolerant version */
int main()
{
  int it;
  double A[10000];
  const int lda = 100;

  initialize(A);

  for(it=0 ; it<100 ; it++) {
    work1(A[lda*it + it]);
  }
}
```

```
/* Fault tolerant version with Fenix */
int main()
{
  int it;
  int A[10000];
  int offsets[100];
  int sizes[100];
  int start_offset_A[100], end_offset_A[100];
  const int lda = 100;
  int status;
  Fenix_Data_subset subset_LU;

  Fenix_Init(&status, ...);
  if( status == FENIX_STATUS_INITIAL_RANK ) {
    /* no failure occurred */
    it = 0;
    initialize(A);
    Fenix_Data_member_create(990, &it, 1, MPI_INT);
    Fenix_Data_member_create(991, A, 10000, MPI_DOUBLE);
    Fenix_Data_member_store(FENIX_DATA_MEMBER_ALL);
    Fenix_Data_group_commit();
  } else {
    /* ranks recovered from a failure, now restore data */
    Fenix_Data_restore(990, &it, 1, MPI_INT, FENIX_LATEST_COMMIT
        );
    Fenix_Data_restore(991, A, 10000, MPI_DOUBLE,
        FENIX_LATEST_COMMIT);
  }

  for( ; it<100 ; it++) {
    Fenix_Data_member_store(990);
    /* Create a subset */
    for( j = it; j < 100; j++ ) {
       start_offset_A[j] = j*100 + j;
       end_offset_A[j] =  start_offset_A[j] + lda;
    }
    Fenix_Data_subset_create(&subset_LU, 100-it, start_offset_A,
         end_offset_A);

    work1(A[lda*it + it]);
    Fenix_Data_member_store(991, FENIX_DATA_GROUP_WORLD_ID,
        subset_A);

    Fenix_Data_group_commit();
    Fenix_Data_subset_delete(&subset_LU);
  }
}
```

4.3 Recovering one member of a data group

This example assumes that ranks have the knowledge of (1) the group identifier `group_id`, (2) the size of the communicator associated with that group (same size as `mycomm`), (3) the features of the member sought (in particular, `member_id`, `count`, and `datatype`) and (4) the specific time stamp `ts` of the sought consistent commit.

```
Fenix_Init(&status, MPI_COMM_WORLD, &new_comm_world,
            &argc, &argv,
            num_spare_ranks,
            0, // spawn
            MPI_INFO_NULL,
            &error);
if( !error && status != FENIX_STATUS_INITIAL_RANK ) {
  // Failure successfully recovered
  my_get_communicator_from_world(new_comm_world, &mycomm);
  Fenix_Data_group_create(group_id, mycomm,
            0,  // These last two params are ignored,
            0); // since group_id already existed
  int dt_size;
  MPI_Type_size(datatype, &dt_size);
  assert(size != MPI_UNDEFINED);
  uint8_t recovered_data = (uint8_t *) malloc(count*dt_size);
  Fenix_Data_member_restore(
            member_id, &recovered_data, count,
            ts, group_id);
  // At this point, the application has its recovered data in
  // all positions of member_pointers.
  // Now, the application should inspect these elements to try
  // and determine what to do with the recovered data.
}
```

4.4 Recovering all members of a data group

This example assumes that ranks have the knowledge of (1) the group identifier `group_id` as well as (2) the size of the communicator associated with that group (same size as `mycomm`).

This example assumes that the recovered rank have no knowledge about the application data contained in the members that were stored. This is a corner case, since the application should be aware of the data associated with a member identifier in a group.

```
Fenix_Init(&status, MPI_COMM_WORLD, &new_comm_world,
            &argc, &argv,
            num_spare_ranks,
            0, // spawn
            MPI_INFO_NULL,
            &error);
if( !error && status != FENIX_STATUS_INITIAL_RANK ) {
  // Failure successfully recovered
  my_get_communicator_from_world(new_comm_world, &mycomm);
  Fenix_Data_group_create(group_id, mycomm,
            0,  // These last two params are ignored,
            0); // since group_id already existed
  int number_of_members;
  Fenix_Data_group_get_number_of_members(
            &number_of_members, group_id);
  uint8_t **member_pointers = (uint8_t **)
            malloc(number_of_members*sizeof(uint8_t *));
  int *member_counts = (int *)
            malloc(number_of_members*sizeof(int));
  MPI_Datatype *member_datatypes = (MPI_Datatype *)
            malloc(number_of_members*sizeof(MPI_Datatype));
  for(int m=0 ; m<number_of_members ; m++) {
    int member_id;
    Fenix_Data_group_get_member_at_position(
            m,
            &member_id,
            group_id);
    int flag;
    Fenix_Data_member_get_attribute(member_id,
            FENIX_DATA_GROUP_MEMBER_ATTRIBUTE_COUNT,
            (void *) &member_counts[m], &flag, group_id);
    assert(flag);
    MPI_Datatype datatype;
    Fenix_Data_member_get_attribute(member_id,
            FENIX_DATA_GROUP_MEMBER_ATTRIBUTE_DATATYPE,
            (void *) &member_datatypes[m], &flag, group_id);
    int dt_size;
    MPI_Type_size(member_datatypes[m], &dt_size);
    assert(size != MPI_UNDEFINED);
    member_pointers[m] = (uint8_t *) malloc(count*dt_size);
    int commit_time_stamp;
    Fenix_Data_group_get_commit_at_position(
            0, &commit_time_stamp, group_id);
    Fenix_Data_member_restore(
            member_id, &(member_pointers[m]), member_counts[m],
            commit_time_stamp, group_id);
  }
  // At this point, the application has its recovered data in
  // all positions of member_pointers.
  // Now, the application should inspect these elements to try
  // and determine what to do with the recovered data.
}
```

Acknowledgments

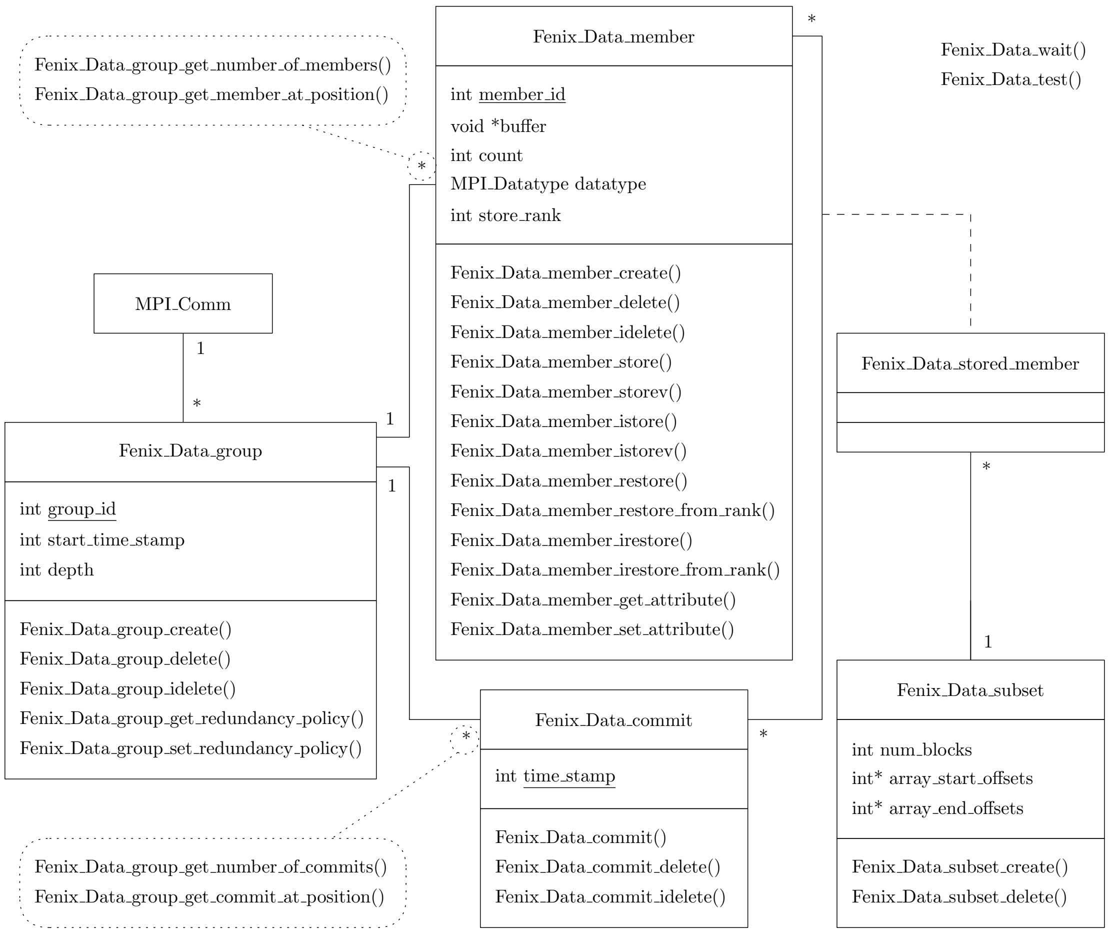
We thank Josep Gamell, Robert L. Clay, and Michael A. Heroux for their help, consistent support, and insightful discussions. We also thank George Bosilca and Ichitaro Yamazaki at University of Tennessee, Knoxville for the useful discussions in MPI-ULFM and realistic use cases of the persistent data storage interface of Fenix.

The research presented in this work is supported in part by National Science Foundation (NSF) via grants numbers CNS 1305375, ACI 1339036, ACI 1310283, ACI 1441376 and IIS 1546145, and by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the US Department of Energy Scientific Discovery through Advanced Computing (SciDAC) Institute for Scalable Data Management, Analysis and Visualization (SDAV) under award number DE-SC0007455, the DoE RSVP grant via subcontract number 4000126989 from UT Battelle, the Advanced Scientific Computing Research and Fusion Energy Sciences Partnership for Edge Physics Simulations (EPSI) under award number DE-FG02-06ER54857, the ExaCT Combustion Co-Design Center via subcontract number 4000110839 from UT Battle, via the SIRIUS grant number DE-SC0015160, and through a grant from Sandia National Laboratories. The research at Rutgers was conducted as part of the Rutgers Discovery Informatics Institute (RDI^2).

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

A Semantic picture of the Fenix Data recovery interface

The diagram below represents a data-centric view of the Data Storage and Recovery Fenix Interface.



Fenix_Data_group_get_number_of_members()
Fenix_Data_group_get_member_at_position()
Fenix_Data_member
int member_id
void *buffer
int count
MPI_Datatype datatype
int store_rank
Fenix_Data_member_create()
Fenix_Data_member_delete()
Fenix_Data_member_idelete()
Fenix_Data_member_store()
Fenix_Data_member_storev()
Fenix_Data_member_istore()
Fenix_Data_member_istorev()
Fenix_Data_member_restore()
Fenix_Data_member_restore_from_rank()
Fenix_Data_member_irestore()
Fenix_Data_member_irestore_from_rank()
Fenix_Data_member_get_attribute()
Fenix_Data_member_set_attribute()
Fenix_Data_wait()
Fenix_Data_test()
MPI_Comm
Fenix_Data_group
int group_id
int start_time_stamp
int depth
Fenix_Data_group_create()
Fenix_Data_group_delete()
Fenix_Data_group_idelete()
Fenix_Data_group_get_redundancy_policy()
Fenix_Data_group_set_redundancy_policy()
Fenix_Data_stored_member
Fenix_Data_commit
int time_stamp
Fenix_Data_commit()
Fenix_Data_commit_delete()
Fenix_Data_commit_idelete()
Fenix_Data_subset
int num_blocks
int* array_start_offsets
int* array_end_offsets
Fenix_Data_subset_create()
Fenix_Data_subset_delete()
Fenix_Data_group_get_number_of_commits()
Fenix_Data_group_get_commit_at_position()
1
*

Five types of logical data classes exist: `Fenix_Data_group`, `Fenix_Data_member`, `Fenix_Data_commit`, `Fenix_Data_stored_member`, and `Fenix_Data_subset`. For each data class, the middle part of each box describes user-accessible fields, in which underlined fields are the unique identifiers of particular instances of each data class. Particular instances from `Fenix_Data_stored_member` can be uniquely identified by the pair (`member_id`, `time_stamp`), since this data class can be seen as an associative class product of including particular members in a particular commit. The bottom part of each box includes functions used to create, delete, or manipulate different instances. Functions inside the two bubbles with dotted lines are functions associated with `Fenix_Data_group`, and can be used to discover the unique identifiers for `Fenix_Data_member` and `Fenix_Data_commit` associated with a particular group, respectively. Note that these data classes are not directly exposed to the user and Fenix implementations can actually choose to use a different layout for internal implementation. This diagram serves as a way to understand the effect and relationship of the different functions.

Fenix Function Index

Fenix Collective Function Index

References

[1] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE Press, 2014, pp. 895–906.

[2] A. Chien, P. Balaji, P. Beckman, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, R. Schreiber *et al.*, "Versioned distributed arrays for resilience in scientific applications: Global view resilience," *Journal of Computational Science*, 2015.

[3] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski, "Detailed modeling, design, and evaluation of a scalable multi-level checkpointing system," *Lawrence Livermore National Laboratory (LLNL), Tech. Rep. LLNL-TR-440491*, 2010.

[4] M. Schulz and B. R. De Supinski, "Pn mpi tools: A whole lot greater than the sum of their parts," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing.* ACM, 2007, p. 30.