# SANDIA REPORT

SAND2016-9171
Unlimited Release
Printed September 2016

# Specification of Fenix MPI Fault Tolerance library version 1.0

Marc Gamell, Rob F. Van der Wijngaart, Keita Teranishi and Manish Parashar

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

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.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.



Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2016-9171
Unlimited Release
Printed September 2016

# Specification of Fenix MPI Fault Tolerance library

version 1.0

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

**Abstract**

This document provides a specification of Fenix, a software library compatible with the Message Passing Interface (MPI) to support fault recovery without application shutdown. The library consists of two modules. The first, termed *process recovery*, restores an application to a consistent state after it has suffered a loss of one or more MPI processes (ranks). The second specifies functions the user can invoke to store application data in Fenix managed redundant storage, and to retrieve it from that storage after process recovery.

## Acknowledgment

We thank Josep Gamell, Robert L. Clay, Michael A. Heroux and Tim Mattson for their help, consistent support, and insightful discussions. We also thank George Bosilca, Aurélien Bouteiller and Ichitaro Yamazaki at University of Tennessee, Knoxville for the useful discussions on MPI-ULFM and realistic use cases of the persistent data storage interface of Fenix. Finally, we thank Eric Valenzuela for his work on the reference implementation of Fenix 1.0.

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.

# Contents

# Chapter 1

# Introduction

This document provides a specification of Fenix, 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 [2]. We only present the C library interface for Fenix; the Fortran interface will be added once the C version is complete.
*End current implementation*

## 1.1 Functionality

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.

**Process recovery.** Only communicators derived from the communicator returned by `Fenix_Init` are eligible for reconstruction. After communicators have been repaired, they contain the same number of ranks as before the failure occurred, unless the user did not allocate sufficient redundant resources (*spare ranks*) and instructed Fenix not to create new ranks. In this case communicators will still be repaired, but will contain fewer ranks than before the failure occurred.

To ease adoption of MPI fault tolerance, Fenix automatically captures any errors resulting from MPI library calls that experienced a failure due to a damaged communicator (other errors reported by the MPI runtime are ignored by Fenix and are returned to the application, for handling by the application writer). In other words, programmers do not need to replace calls to the MPI library with calls to Fenix (for example, `Fenix_Send` instead of `MPI_Send`).

*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 [3].
*End current implementation*

**Data recovery.** 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 [4]) or SCR (Scalable Checkpoint/Restart [5]) to restore rank data after a failure. The crux is that the program does not have to be restarted completely.

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.

*Current implementation*
Fenix currently does not have a thread safety model.
*End current implementation*

## 1.2 Terms and format

When describing Fenix functions, we will indicate for each 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.

For each function we list whether it is collective or not. If collective, all ranks in a specific communicator must call that function at logically the same time. A collective function may have local or non-local (potentially communicating with other ranks) completion semantics. If it is globally synchronizing, it is automatically non-local; no ranks in the associated communicator can complete the function until all ranks have called it.

This document contains sections that give advice to users, or that clarify the current implementation of Fenix. These sections are indicated by

*Advice to users*
Example advice.
*End advice to users*

and

*Current implementation*
Example clarification.
*End current implementation*,

respectively. They are not part of the specification.

# Chapter 2

# Initialization, Rank Failure Recovery, and Teardown

## 2.1 Initialization

**Fenix Init** *(collective operation)*

```
void Fenix_Init(
        MPI_Comm comm,
        MPI_Comm *newcomm,
        int *role,
        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`, `spawn`, and `info`. `Fenix_Init` must be called exactly once by each rank. 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 resumption 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, see below).

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 removed by `Fenix_Init`.

`Fenix_Init` is blocking, in the sense that no ranks are allowed to exit from it until all active ranks have returned control to it (provided the execution resumption point after an intercepted error has been set to `fenix_init` in the `info` parameter, which is also the default behavior, see below).

*Current implementation*
While `Fenix_Init` is called explicitly only once in a user code, control is transferred into it indirectly by the Fenix library whenever an active rank calls an MPI function whose associated resilient communicator has been damaged–a condition automatically intercepted by Fenix, based on ULFM's failure notification mechanism. Consequently, the function may experience delay if certain ranks do not call MPI functions for a long time, or only call MPI functions involving communicators that were not affected by the error.
*End current implementation*

Parameters:

- `comm` [IN] - communicator that includes any spare ranks the user deems necessary. It will be used by Fenix to derive a *resilient communicator*. We define a resilient communicator as one whose accesses are monitored by Fenix, which will repair it in case it suffers failed ranks; any communicator derived from a resilient communicator is automatically resilient itself. `MPI_COMM_SELF` is not a valid value for `comm`.

  *Advice to users*
  `MPI_COMM_WORLD` is a valid value for `comm`.
  *End advice to users*

- `newcomm` [OUT] - resilient output communicator, managed by Fenix and derived from `comm`, to be used by the application instead of `comm`. Let the number of ranks in `comm` be $C$, the number of spare ranks $S$ (equals `spare_ranks`), and the number of ranks failed thus far $F$ ($F$ is assumed 0 at the first invocation of `Fenix_Init`). Upon exit from `Fenix_Init` `newcomm` contains:

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

  If `newcomm` equals `NULL`, Fenix will tacitly replace every occurrence of communicator `comm` in the code with `Fenix_Init`'s resilient output communicator.

  *Advice to users*
  If `newcomm` equals `NULL`, the user can utilize `MPI_COMM_WORLD` as a resilient communicator. This constitutes a significant convenience, since the user would not need to replace occurrences of `MPI_COMM_WORLD` in the code explicitly with a different resilient communicator.
  *End advice to users*

  *Current implementation*
  The current implementation uses the PMPI interface. If `newcomm` equals `NULL`, the interface will intercept any use of `comm` and replace it with the resilient communicator (which is stored inside Fenix and is not visible to the user).
  *End current implementation*

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

- `role` [OUT] - upon return, contains one of the following values, indicating the most recent history of the calling rank:

  – `FENIX_ROLE_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_ROLE_RECOVERED_RANK` - this rank replaces a failed rank since the latest resilient 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_ROLE_SURVIVOR_RANK` - this rank was not affected by the rank failure that triggered the latest resilient communicator restoration by Fenix.

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

- `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 in `comm` that are exempted by Fenix in the construction of the resilient communicator by `Fenix_Init`[1]. These ranks are kept in reserve to substitute for failed ranks. Failed ranks in resilient communicators are replaced by spare or spawned ranks.

  *Current implementation*
  First, spare ranks are used to substitute failed ranks. When these are depleted, either (1) failed ranks are substituted with spawned ranks (if `spawn` equals `true`), or (2) survivor ranks are compacted to shrink the resilient communicator (if `spawn` equals `false`).
  *End current implementation*

  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 automatically call `MPI_Finalize` and exit when all active ranks have entered the `Fenix_Finalize` call.

- `spawn` [IN] - used to specify whether Fenix may attempt to spawn replacement ranks or not.

  - If `spawn` equals `false`, and insufficient spare ranks are available to replace all failed ranks, no new ranks will be spawned to fill out original communicators. Failures will be resolved by Fenix by "compacting" survivor ranks within their respective resilient communicators, such that they retain the same order as before the failure, but they are numbered successively and contiguously within the shrunk communicator.

    Note that this mode, in combination with requesting no spare ranks, can be used to force a shrinking communicator repair mechanism.

  - If `spawn` equals `true`, and insufficient spare ranks are available to replace all failed ranks, some or all required new ranks will be spawned, using `MPI_Comm_spawn`, to fill out original communicators. Fenix includes the entire key-value dictionary of the `info` parameter of `Fenix_Init` in the `info` parameter of `MPI_Comm_spawn`.

- `info` [IN] - `MPI_Info` object to further modify Fenix's process recovery behavior. 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 for all active (non-spare) ranks in any resilient communicators, not only for those ranks in communicators that failed. The following values associated with the `"resume_mode"` key must be supported.

  - `"fenix_init"` - execution resumes at logical exit of `Fenix_Init`.

- `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. It has the same value across all ranks released by `Fenix_Init`. If spawning is explicitly disabled (`spawn` equals `false`) and spare ranks have been depleted, Fenix will repair resilient communicators by shrinking them and will report such shrinkage in the `error` return parameter through the value `FENIX_WARNING_SPARE_RANKS_DEPLETED`. If spawning is enabled, but fails to create the required new ranks in the absence of a sufficient number of spare ranks, Fenix will repair resilient communicators by shrinking them and will report such shrinkage in the `error` return parameter through the value `FENIX_ERROR_SPAWNING_FAILED`.

Spare ranks are not released from `Fenix_Init` until they have been used by Fenix to repair damaged resilient 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 Fenix once the spare ranks

[1]While it may be more accurate to use the term spare *MPI processes*, since spare *ranks* taken from `comm` are to be used for substitution in other communicators, we stick with the shorter term for readability.

have been depleted, subject to the rank repair policy specified by the user) start executing the main 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 collective communications. It is the user's responsibility to avoid such problems.

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.

*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)(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 assume that no error occurred within the callback. Therefore, if an error does occur, it needs to be either solved within the callback or escalated by using mechanisms such as `MPI_Abort`. Callback functions need to observe the following prototype:

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

- `newcomm` [IN] - contains the resilient communicator returned by `Fenix_Init`. When the callback is invoked, this communicator has already been repaired by Fenix and, therefore, `comm` is identical to `newcomm` as 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`.

Since the registration of callback functions is not collective, the callback itself should not issue any MPI collective communication.

## 2.3 Querying active ranks

Even though the application can obtain information about the roles of ranks after a failure, that may require a collective communication among ranks in the target resilient communicator. Fenix has access to this information locally and the application can access it by using the following operations.

**Fenix Get number of ranks with role**

```
int Fenix_Get_number_of_ranks_with_role(
        MPI_Comm comm,
        int role,
        int *number_of_ranks);
```

This function returns the total number of ranks in resilient communicator `comm` that have a particular `role`. This function is performed locally and involves no communication with other ranks.

- `comm` [IN] - communicator whose ranks are being queried.
- `role` [IN] - queried role. See the description of the `role` output parameter of `Fenix_Init` for a clarification of the possible values.
- `number_of_ranks` [OUT] - number of ranks in `comm` whose role equals `role`.

**Fenix Get role**

```
int Fenix_Get_role(
        MPI_Comm comm,
        int rank,
        int *role);
```

This function can be used to query a particular rank in resilient communicator `comm` about its role. This function is performed locally and involves no communication with other ranks.

- `comm` [IN] - communicator whose rank is being queried.
- `rank` [IN] - rank whose role is requested.
- `role` [OUT] - the role of the queried rank. See the description of the `role` output parameter of `Fenix_Init` for a clarification of the possible values.

## 2.4 Teardown

**Fenix Finalize** *(collective operation)*

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

This function must be called by all ranks in the resilient communicator returned by `Fenix_Init` 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 such rank 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`.

*Advice to users*
Sometimes users may want to remove ranks proactively from the execution, for example because monitoring data shows that failure of a rank is imminent. This can be accomplished simply by calling `exit` on the targeted ranks, followed by an invocation of `MPI_Barrier`. The removed ranks will not reach the barrier, causing an error among the remaining ranks in the resilient communicator supplied to the barrier function. This error will be intercepted by Fenix, which will attempt to repair the affected communicator, excluding any eliminated ranks[2].

The smaller the comunicator used in the invocation of the barrier is chosen, the slower the effect of removing ranks from that communicator may percolate to other ranks.
*End advice to users*

[2] An out-of-band solution could also be sending a `SIGKILL` signal to the targeted ranks.

# Chapter 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. Fenix 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` finalizes all preceding Fenix store operations involving this data group and assigns a unique time stamp to the resulting data *snapshot*, marking the data as potentially recoverable after a loss of ranks. Individual pieces of data can then be restored whenever they are needed with `Fenix_Data_member_restore`, for example after a failure occurs. We note that the Fenix'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, Fenix 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, which has local completion semantics, must be called by all ranks in resilient communicator `comm`. All must pass the same values for all parameters.

- `group_id` [IN] - identifier of the group, unique among all active MPI ranks in the application. 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. It must be a nonnegative integer less than `FENIX_GROUP_ID_MAX`, with the latter value guaranteed to be at least $2^{30}$.

- `comm` [IN] - ranks in `comm` must 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_(i)store(v)` call.

- `start_time_stamp` [IN] - each subsequent data snapshot of this group has an index that uniquely identifies the snapshot within the group. This index is called a *time stamp*. The `start_time_stamp` is the index of the first data snapshot of this group to be written, and can be defined by the user (for example, set to zero); this value will be incremented by one automatically each time the 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 data snapshots (see `Fenix_Data_commit`) of this group that are retained by Fenix, in addition to the last one, and can be recovered by calling `Fenix_Data_member_restore`. For example, a depth of 0 means Fenix will keep only the necessary data to restore the most recent snapshot, while it will mark older snapshots for deletion. These may be removed by Fenix' garbage collector functions, see Section 3.4.3. A depth of $-1$ means Fenix will not remove any older snapshots automatically (applies to `Fenix_Data_barrier`, `Fenix_Data_cleanup`, `Fenix_Data_commit_barrier`, and `Fenix_Data_commit_cleanup`, see Sections 3.4.2 and 3.4.3). In that case only explicit, manual deletion of out of date snapshots is possible, see Section 3.8.

*Current implementation*
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 resilient 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 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
        comm,                      // communicator
        0,                         // start_time_stamp
        0);                        // depth
```

where `comm` is the resilient communicator produced when `Fenix_Init` returned most recently. In other words, `FENIX_DATA_GROUP_WORLD_ID` is a convenient constant to represent a data group involving all active ranks via a reserved `group_id`, an initial time stamp of zero, and a garbage collection depth of zero (i.e. Fenix will keep only the last snapshot).

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.

Any Fenix data group except `FENIX_DATA_GROUP_WORLD_ID` can be deleted, using the following function. Along with the group, any locally stored application data associated with the group (see section 3.4.1) will also be deleted.

**Fenix Data group delete**

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

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

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. It will recursively delete all its members, including data snapshots, on the calling rank.

### 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 group_id,
        int member_id,
        void *source_buffer,
        int count,
        MPI_Datatype datatype);
```

This function must be called by all ranks in the resilient communicator 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`.

- `group_id` [IN] - identifier of the data group containing the member.
- `member_id` [IN] - integer within the named group `group_id` that uniquely identifies the data in `source_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 $2^{30}$.
- `source_buffer` [IN] - address of data to be copied to redundant storage maintained by Fenix. Note that this parameter may also be specified using the function `Fenix_Data_member_attr_set`. The latter is critical for non-survivor ranks (`FENIX_ROLE_RECOVERED_RANK`) after a failure. In that case data group members are *implicitly* recreated by Fenix 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 source buffer pointer specified before the failure, unless it is overwritten by `Fenix_Data_member_attr_set`.

- `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 `source_buffer`.

**Fenix Data member delete**

When a data group member is no longer needed, it may be deleted on the calling rank by the following function. Along with the data group member, any locally stored application data associated with the member (see section 3.4.1) will also be deleted.

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

- `group_id` [IN] - identifier of the group containing this member.
- `member_id` [IN] - unique integer within the named group that identifies the data in `source_buffer`.

Note that members can be added to or deleted from a group at any point between the calls to `Fenix_Data_group_create` and `Fenix_Data_group_delete`.

### 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.

**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 Fenix 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)*

[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.

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

This function with local completion semantics must be called by all ranks in the resilient communicator associated with the group identified by `group_id` at logically the same time. All calling ranks must pass the same values for the parameters `group_id`, `policy_name`, and the contents of `policy_value`.

This function is used to define the type of policy Fenix 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 Fenix 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 resilient communicator corresponding to the 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. 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, not 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 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 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 group_id,
        int member_id,
        Fenix_Data_subset subset_specifier);
```

This function has non-local completion semantics. It must be called by all ranks in the resilient communicator associated with the group identified by `group_id` at the same logical time for the same `member_id`. While it is logically collective, it does not require all ranks within the communicator to synchronize with each other, so the call should not be used for global synchronization. This function is used to safeguard the data belonging to a particular member of the data group. It places one or more copies of data residing in `source_buffer` (supplied in the call to the function `Fenix_Data_member_create`) in Fenix' redundant data storage.

*Current implementation*
After creating a copy of this member in the calling rank's memory, 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. When the call returns, the application can safely modify the data in `source_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. Such recovery requires the group identifier, the member identifier, and the logical time stamp of the saved data.

Multiple calls to `Fenix_Data_member_store` with the same `member_id` without intervening commit calls 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.

- `group_id` [IN] - identifier of the group associated with this member.
- `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.
- `subset_specifier` [IN] - specifier of the subset of data to be stored. The choice of this parameter, while in principle strictly local, needs to result in subsets of identical extent in all calling ranks. When a `subset_specifier` different than `FENIX_DATA_SUBSET_FULL` is supplied, Fenix will only store the positions in the application source buffer that are in the subset. When `subset_specifier` equals `FENIX_DATA_SUBSET_EMPTY`, no data will be stored.

*Advice to users*
The requirement on resultant subset extent 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.
*End advice to users*

**Fenix Data member storev** *(collective operation)*

```
int Fenix_Data_member_storev(
        int group_id,
        int member_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` and parameter `count` in the call to `Fenix_Data_member_create` can be different in different ranks.

**Fenix Data member istore** *(collective operation)*

```
int Fenix_Data_member_istore(
        int group_id,
        int member_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 source 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 source 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.

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

**Fenix Data member istorev** *(collective operation)*

```
int Fenix_Data_member_istorev(
        int group_id,
        int member_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 group_id,
        int *time_stamp);
```

This function must be called by all ranks in the group's resilient communicator at the same logical time. 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 snapshot of the stored application data.

- `group_id` [IN] - identifier of the group to commit.
- `time_stamp` [OUT] - pointer to index of the committed data. `NULL` is a valid value, in which case the automatically incremented index is not returned to the application.

  The `time_stamp` parameter will be a nonnegative integer no larger than `FENIX_TIME_STAMP_MAX`, with the latter value guaranteed to be at least $2^{30}$.

While the commit is logically collective, it has local completion semantics, so it cannot be used for synchronization. Consequently, there is no guarantee that a data member in a snapshot created with `Fenix_Data_commit` is consistent, which is a requirement for being recoverable. A data snapshot is *consistent* with respect to a group member if all ranks in the group's communicator have committed their stores to that group with the same time stamp, and if the member existed on all ranks at the time of the commit. Consistency can be ensured by calling the globally synchronizing function `Fenix_Data_barrier`, see section 3.4.3.

For convenience, Fenix provides the following function, which combines the time stamp function of `Fenix_Data_commit` with the consistency enforcement of `Fenix_Data_barrier`.

**Fenix Data commit barrier** *(collective operation)*

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

Upon completion of this call, only completely consistent snapshots of the named group remain in Fenix' redundant storage, and any snapshots older than the depth specified in the call to `Fenix_Data_group_create` have been removed (the latter is ignored if depth equals $-1$).

If consistency is not a concern, for example because the application itself is regularly globally synchronizing and does not perform any selective member removals, there could still be the need to remove out of date snapshots. This can be done by calling `Fenix_Data_cleanup`, see section 3.4.3.

For convenience, Fenix provides the following function, which combines the time stamp function of `Fenix_Data_commit` with the garbage collection of `Fenix_Data_cleanup`. While collective, it has local completion semantics. If the depth equals $-1$, no snapshot is ever considered out of date, so none will be removed by this function.

**Fenix Data commit cleanup** *(collective operation)*

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

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(v)` operations in the group before committing.

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

### 3.4.3 Data consistency and garbage collection

**Fenix Data barrier** *(collective operation)*

```
int Fenix_Data_barrier(
        int group_id);
```

This function is collective and globally synchronizing across the data group's resilient communicator. It performs garbage collection and enforces consistency of data and meta-data for group with label `group_id`. It will remove consistent snapshots that are older than the depth specified in the creation of the group, as well as any inconsistent snapshots, to reduce storage pressure. In addition, it removes across all time stamps any members from the group that are not present in all calling ranks. If the group is not present in all calling ranks, it will remove the entire group. If the depth equals $-1$, no snapshot is ever considered out of date, so none will be removed on that basis by this function.

- `group_id` [IN] - Fenix data group

**Fenix Data cleanup** *(collective operation)*

```
int Fenix_Data_cleanup(
        int group_id);
```

This function is collective but has local completion semantics. It removes any snapshots of the named group that are older than the depth specified in the call to `Fenix_Data_group_create`. If the depth equals $-1$, this function cannot be used to remove old snapshots, and the only method to accomplish that is via manual deletion using `Fenix_Data_snapshot_delete`, see Section 3.8.

- `group_id` [IN] - Fenix data group

## 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 previous data snapshots. 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 Fenix' redundant storage.

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

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

This is a blocking collective function. All calling ranks must pass the same values for the parameters `group_id`, `member_id`, and `time_stamp`. This function is used to retrieve data from consistent snapshot members. Any inconsistent or out of date snapshots may be removed by Fenix at this time. 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 rank loss).

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

Parameters:

- `group_id` [IN] - group that contains the requested data.
- `member_id` [IN] - this value must match the member id that was supplied when `Fenix_Data_member_store` was called.
- `target_buffer` [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 calling rank will receive the selected data from the corresponding rank in the communicator used at the time the snapshot was taken.
- `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] - time stamp of the first snapshot to be inspected for the presence of valid recovery data. Fenix will inspect successively older available consistent snapshot members until it has found for each element of the requested member a valid recovery value. The availability of such data depends on the choice of subsets used in data storage calls, and potentially selective member removal or time stamp skipping. If no value is found, the corresponding element of the receiving buffer is left unchanged. The special time stamp value of `FENIX_DATA_SNAPSHOT_LATEST` will always identify the group's latest consistent snapshot.

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

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

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

```
int Fenix_Data_member_restore_from_rank(
        int group_id,
        int member_id,
        void *target_buffer,
        int max_count,
        int time_stamp,
        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 by each calling rank.

Parameters:

- `source_rank` [IN] - specifies the rank (in the resilient communicator associated with `group_id`) that performed the data store and whose data is being recovered. Its value can be set independently by all ranks in the communicator.

We note that this function does not require that the resilient communicator has shrunk, and can be used for any recovery pattern consistent with its 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.

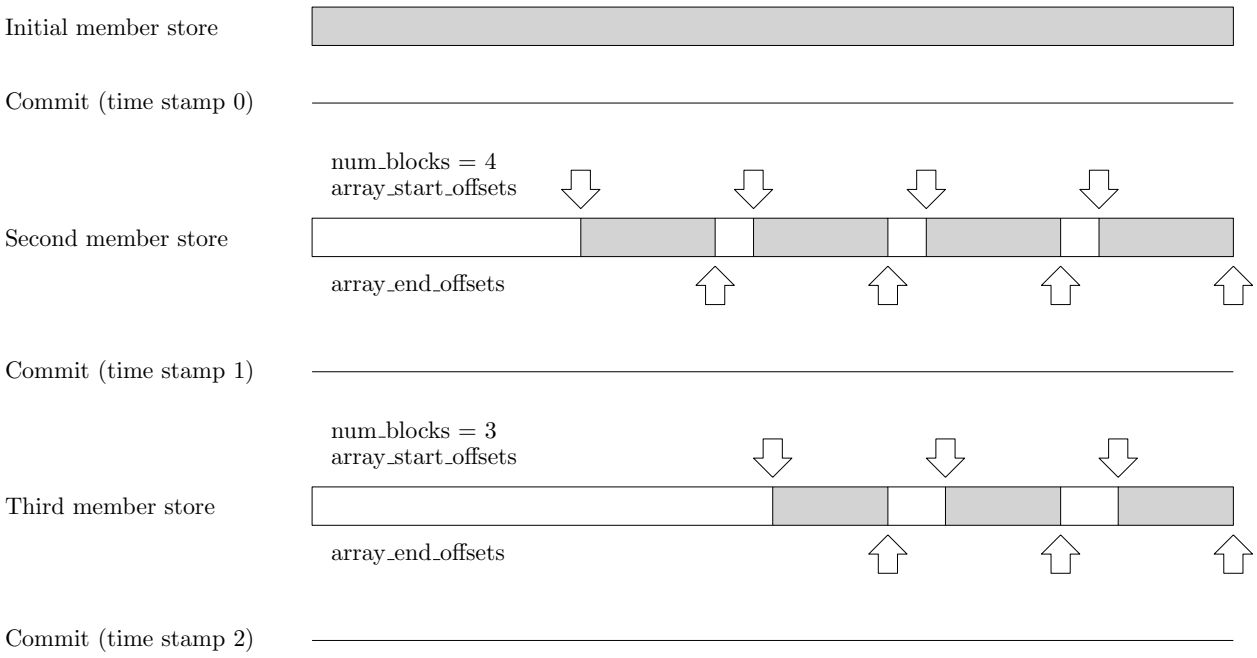


**Figure 3.1.** Incremental member store using *subsets*. Gray areas indicate the data being saved by `Fenix_Data_member_store` operations.

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 3.1. 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(
        int num_blocks,
        int* array_start_offsets,
        int* array_end_offsets,
        Fenix_Data_subset *subset_specifier);
```

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

- `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`.
- `subset_specifier` [OUT] - name of the subset specifier, to be used in storing data.

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 Fenix Data 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.

### 3.7.1 Querying group members

**Fenix Data group get number of members**

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

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

**Fenix Data group get member at position**

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

- `member_id` [OUT] - the unique identifier of the `Fenix_Data_member` sought.
- `group_id` [IN] - Fenix data group whose information is sought.
- `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.

### 3.7.2 Querying snapshots

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

```
int Fenix_Data_group_get_number_of_snapshots(
        int group_id,
        int *number_of_snapshots);
```

This function must be called by all ranks in the group's resilient communicator. It has non-local completion semantics, but is not necessarily globally synchronizing. All calling ranks must pass the same value for the parameter `group_id`.

- `group_id` [IN] - Fenix data group whose information is sought.
- `number_of_snapshots`[OUT] - number of available, distinct, consistent snapshots of this group. Inconsistent or deleted snapshots (removed either manually or through garbage collection, see `depth` in `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 to by `number_of_snapshots`.

**Fenix Data group get snapshot at position**

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

- `group_id` [IN] - Fenix data group whose information is sought.
- `position` [IN] - sequence number of the requested consistent snapshot. `position` must be a value between 0 and `number_of_snapshots`-1. Snapshot positions will be returned in the reverse order in which the user committed them, i.e. oldest last, newest first (e.g. the most recent completed consistent snapshot will have `position`=0).
- `time_stamp` [OUT] - the unique index of the snapshot 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 attr get** *(collective operation)*

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

This function must be called by all ranks in the resilient communicator 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:

- `group_id` [IN] - Fenix data group whose information is sought.
- `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.
- `source_rank` [IN] - for attributes that are rank-dependent (such as `FENIX_DATA_MEMBER_ATTRIBUTE_COUNT`), specifies the rank (in the resilient communicator associated with `group_id`) that contains the attribute that is sought.

**Fenix Data member attr set**

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

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_SOURCE_BUFFER` must be writable after a failure is recovered.

- `group_id` [IN] - Fenix data group whose information is sought.

- `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.

## 3.8 Removing stored application data

The following function removes irretrievably a specific snapshot of a data group. 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`.

**Fenix Data snapshot delete**

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

- `group_id` [IN] - group whose snapshot(s) should be removed.
- `time_stamp` [IN] - the time stamp of the requested snapshot. The special value of `FENIX_DATA_SNAPSHOT_LATEST` will always remove the latest snapshot. The special value of `FENIX_DATA_SNAPSHOT_ALL` can be used to remove all snapshots.

We note that redundant application data may also deleted as a side effect of the functions `Fenix_Data_group_delete` and `Fenix_Data_member_delete`. See Sections 3.2.1 and 3.2.2.

# Chapter 4

# Examples

For the Fenix codes we use the following shorthands for convenience:

```
#define BUF          FENIX_DATA_MEMBER_ATTRIBUTE_SOURCE_BUFFER
#define FULL         FENIX_DATA_SUBSET_FULL
#define LATEST       FENIX_DATA_SNAPSHOT_LATEST
#define FENIX_WORLD  FENIX_DATA_GROUP_WORLD_ID
#define COUNT        FENIX_DATA_GROUP_MEMBER_ATTRIBUTE_COUNT
#define TYPE         FENIX_DATA_GROUP_MEMBER_ATTRIBUTE_DATATYPE
#define VP           void *
```

## 4.1 Protecting process and data with Fenix

We show two versions of the same mini-example application, one without fault tolerance, and one augmented with Fenix that tolerates failures in an on-line manner.

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

  MPI_Init(&argc, &argv);
  initialize(A, B);

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

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

  MPI_Init(&argc, &argv);
  Fenix_Init(MPI_COMM_WORLD, &new_comm_world, &role,
             &argc, &argv,
             10,     // num_spare_ranks
             0,      // spawn
             MPI_INFO_NULL,
             &error);
  /* regardless of role, we (re)create the data group */
  Fenix_Data_group_create(
    66,             // group_id
    new_comm_world, // resilient communicator
    0,              // starting index for snapshots
    0);             // depth
  if( !error && role == FENIX_ROLE_INITIAL_RANK) {
```

```
    /* no failure occurred */
    it = 0;
    initialize(A, B);
    Fenix_Data_member_create(66, 90, (VP)&it, 1, MPI_INT);
    Fenix_Data_member_create(66, 91, (VP)A, 100, MPI_INT);
    Fenix_Data_member_create(66, 92, (VP)B, 50,  MPI_INT);
    /* make B recoverable */
    Fenix_Data_member_store(66, 92, FULL);
    Fenix_Data_commit_barrier(66, NULL);
  } else if(!error) {
    /* ranks recovered from a failure, now restore data */
    Fenix_Data_member_restore(66, 90, (VP)&it, 1, LATEST);
    Fenix_Data_member_restore(66, 91, (VP)A, 100, LATEST);
    Fenix_Data_member_restore(66, 92, (VP)B,  50, LATEST);
    /* need to say where member data lives, for future stores */
    Fenix_Data_member_attr_set(66, 90, BUF, (VP)&it, &flag);
    Fenix_Data_member_attr_set(66, 91, BUF, (VP)A,   &flag);
    Fenix_Data_member_attr_set(66, 92, BUF, (VP)B,   &flag);
  } else {
    // There was an error in Fenix
    MPI_Abort(MPI_COMM_WORLD, -1);
  }

  /* it may be initialized or recovered */
  for( ; it<1000; it++ ) {
    Fenix_Data_member_store(66, 90, FULL);
    work1(A, new_comm_world);
    if(A[0] > 200) {
      work2(A, B, new_comm_world);
      Fenix_Data_member_store(66, 92, FULL);
    }
    Fenix_Data_member_store(66, 91, FULL);
    Fenix_Data_commit_barrier(66, NULL);
  }
  Fenix_Finalize();
  MPI_Finalize();
}
```

## 4.2 Storing data objects with subsets

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

  MPI_Init(&argc, &argv);
  initialize(A);

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

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

  MPI_Init(&argc, &argv);
  Fenix_Init(MPI_COMM_WORLD, &new_comm_world, &role, ...);
```

```
  /* regardless of role, we (re)create the data group */
  Fenix_Data_group_create(
    66,             // group_id
    new_comm_world, // resilient communicator
    0,              // starting index for snapshots
    0);             // depth
  if(!error && role == FENIX_ROLE_INITIAL_RANK) {
    /* no failure occurred */
    it = 0;
    initialize(A);
    Fenix_Data_member_create(66, 90, (VP)&it, 1,     MPI_INT);
    Fenix_Data_member_create(66, 91, (VP)A,   10000, MPI_DOUBLE);
    Fenix_Data_member_store(66, FENIX_DATA_MEMBER_ALL, FULL);
    Fenix_Data_commit(66, NULL);
  } else {
    /* ranks recovered from a failure, now restore data */
    Fenix_Data_restore(66, 90, (VP)&it, 1,     LATEST);
    Fenix_Data_restore(66, 91, (VP)A,   10000, LATEST);
  }

  for( ; it<100 ; it++) {
    Fenix_Data_member_store(90);
    /* 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(100-it, start_offset_A, end_offset_A, &subset_LU);

    work1(A[lda*it + it]);
    Fenix_Data_member_store(66, 91, FENIX_WORLD, subset_LU);

    Fenix_Data_commit(66, NULL);
    Fenix_Data_subset_delete(&subset_LU); // garbage collection
  }
}
```

## 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 `new_comm_world`), (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 snapshot.

```
Fenix_Init(MPI_COMM_WORLD, &new_comm_world, &role,
           &argc, &argv,
           num_spare_ranks,
           0, // spawn
           MPI_INFO_NULL,
           &error);
if( !error && role != FENIX_ROLE_INITIAL_RANK ) {
  // Failure successfully recovered
  Fenix_Data_group_create(group_id, new_comm_world,
           0,  // These last two params are ignored,
           0); // since group_id already existed
  MPI_Type_size(datatype, &dt_size);
  uint8_t recovered_data = (uint8_t *) malloc(count*dt_size);
  Fenix_Data_member_restore(
           group_id, member_id, (VP)&recovered_data, count, ts);
  // 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 `new_comm_world`).

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(MPI_COMM_WORLD, &new_comm_world, &role,
           &argc, &argv,
           num_spare_ranks,
           0, // spawn
           MPI_INFO_NULL,
           &error);
MPI_Comm_rank(new_comm_world, &my_rank);
if( !error && role != FENIX_ROLE_INITIAL_RANK ) {
  // Failure successfully recovered
  Fenix_Data_group_create(group_id, new_comm_world,
           0,  // These last two params are ignored,
           0); // since group_id already existed
  Fenix_Data_group_get_number_of_members(
           group_id, &number_of_members);
  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++) {
    Fenix_Data_group_get_member_at_position(
           group_id, &member_id, m);
    Fenix_Data_member_attr_get(group_id, member_id, COUNT,
           (VP)&member_counts[m], &flag, my_rank);
    Fenix_Data_member_attr_get(group_id, member_id, TYPE,
           (VP)&member_datatypes[m], &flag, my_rank);
    MPI_Type_size(member_datatypes[m], &dt_size);
    member_pointers[m] = (uint8_t *) malloc(count*dt_size);
    Fenix_Data_group_get_snapshot_at_position(
           group_id, 0, &time_stamp);
    Fenix_Data_member_restore(
           group_id, member_id, (VP)&(member_pointers[m]),
           member_counts[m], time_stamp);
  }
  // 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.
}
```

# 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] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *International Journal of High Performance Computing Applications*, p. 1094342013488238, 2013.

[3] 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.

[4] 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.

[5] 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.

DISTRIBUTION:

,

1 MS 0899 Technical Library, 8944 (electronic copy)

Sandia National Laboratories