Hewlett Packard Enterprise

OpenFAM

An API for programming Fabric Attached Memory

Sharad Singhal and Kim Keeton
4-11-2018

Document History

Document Name: OpenFAM
File: openFAM-API-v104.docx
Version: 1.04
Created: 3/27/2018 12:48:00 PM by Sharad Singhal
Last saved: 4/11/2018 5:46 PM by Keeton, Kimberly
This copy printed: Wednesday, April 11, 2018

Contributors

Name	Organization	Telephone	Email
Sharad Singhal	Hewlett Packard Labs	+1 (650) 258-3346	sharad.singhal@hpe.com
Kim Keeton	Hewlett Packard Labs	+1 (650) 258-7944	kimberly.keeton@hpe.com

This document incorporates feedback and suggestions from

Name	Organization	Name	Organization
Keith Packard	Hewlett Packard Enterprise		
Rocky Craig	Hewlett Packard Enterprise		
Nic McDonald	Hewlett Packard Enterprise		
Haris Volos	Hewlett Packard Enterprise		

Version Notes

Version	Notes
1	Initial version for distribution
1.01	Updates to Atomics API, changed implicit lengths to uint_64.
1.02	Change parameters to uint64_t etc. to be explicit about word lengths. Deleted register API, set_options(), added fam_copy()
1.03	Added fam_quiet(), return values for fam_{get,put,gather,scatter,copy,resize_region,change_permissions}. Changed semantics for fam_fence() and name for fam_compare_swap(). Added discussion section for implementation specifics and open questions.
1.04	Added explicit non-blocking variants of fam_get(), fam_put(), fam_gather() and fam_scatter(); added discussion of failure handling (fail-fast and failure-reporting) models; expanded name service to be metadata service and revised discussion topics.

Note that this document is under active revision. Issues that need addressing or discussion items are marked using colored text in brackets such as <comment or issue yet to be resolved, or text yet to be written>. Please address any comments or questions on this document to openfam@groups.ext.hpe.com.

Contents

Introduction

Recent technology advances in high-density, byte-addressable non-volatile memory (NVM) and low-latency interconnects have enabled building large-scale systems with a large disaggregated fabric-attached memory (**FAM**) pool shared across heterogeneous and decentralized compute nodes. In this model, compute nodes are decoupled from FAM, which allows separate evolution and scaling of processing and memory. Thus, the compute-to-memory ratio can be tailored to the specific needs of the workload. Compute nodes fail independently of FAM, and these separate fault domains provide a partial failure model that avoids a complete system shutdown in the event of a component failure. When a compute node fails, updates propagated to FAM remain visible to other compute nodes.

The traditional shared-nothing model of distributed computing partitions data between compute nodes. Each compute node "owns" its local data and relies on heavyweight two-sided message passing and data copying to coordinate with other nodes. Data owners mediate access to their data, performing work on behalf of the requester. This model suffers mediation overheads and doesn't sufficiently leverage the data sharing potential of FAM systems.

In contrast, the large capacity of the FAM pool means that large working sets can be maintained as in-memory data structures. The fact that all compute nodes share a common view of memory means that data sharing and communication may be done efficiently through shared memory, without requiring explicit messages to be sent over heavyweight network protocol stacks. Additionally, data sets no longer need to be partitioned between compute nodes, as is typically done in clustered environments, and they can avoid message-based coordination overheads. Any compute node can operate on any data item, which enables more dynamic and flexible load balancing. More generally, sharing permits new approaches to cooperation.

In this document, we describe an application programming interface (API) for use in systems that contain FAM. The API is close to and is patterned after APIs provided by one-sided partitioned global address space (PGAS) libraries such as OpenSHMEM[1]. This maintains ease of use for application writers already familiar with those libraries. As such, we refer to this API specification as the OpenFAM API. The primary distinctions between this API and those provided by other PGAS libraries are:

1. Our focus is on fabric-attached disaggregated memory, rather than remote node memory. Since FAM is no longer associated with a specific PE, it can be addressed directly from any PE without the cooperation and/or involvement of any other PE.
2. Because state in FAM can survive program termination, additional interfaces that associate user-friendly names to data in FAM are defined to simplify data management beyond the lifetime of a single program.
3. Disaggregated memory can provide non-functional attributes (such as hardware-level redundancy) that are not normally present in DRAM. A two-level allocation scheme is introduced to take advantage of such characteristics.
4. The independence of state maintained in FAM provides additional capability for managing application availability in the presence of component failure.
5. We include an API for mapping portions of fabric-attached disaggregate memory into a PE's virtual address space, and permit direct load/store access to leverage the capabilities provided by some memory-semantic fabrics.

[1] OpenSHMEM Application Programming Interface version 1.4: http://www.openshmem.org/site/Specification

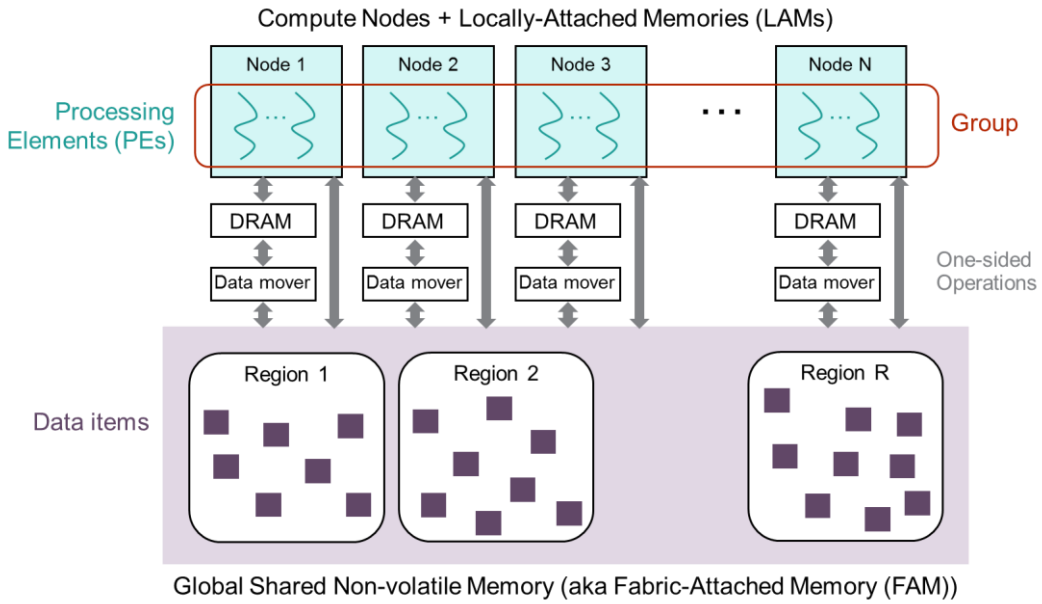

Figure 1: System view in OpenFAM

Figure 1 shows the system model assumed in OpenFAM. The system consists of a multi-OS environment where each compute node runs a separate operating system instance, with locally attached memory that is "private" to the OS instance. In addition, the system provides fabric attached memory that is also directly addressable (as memory) in a Global Address Space (GAS). Programmable data movers, e.g., direct memory access (DMA) engines, support efficient high-speed movement of data between local memory and fabric attached memory, and within different parts of fabric attached memory at a hardware level. To distinguish references to the two types of memory, we will use the term **FAM** to refer to fabric attached memory and **local memory** to refer to the DRAM (or persistent memory) attached locally to a processing node in the discussion below. We also assume in the API that parts of FAM may be persistent to enable data to be shared not only within a running program, but also across program instances in larger computational workflows.

The application spans compute nodes, and is composed of a group of processing elements (**PE**s) that cooperate with one another. Each PE represents a thread of execution that uses both local memory and FAM to perform its tasks. FAM acts as a shared (and potentially persistent) space where PEs may place and access data. As in other PGAS programming models, the application is responsible for coordination of accesses between PEs to any shared data and for managing data consistency in FAM.

We assume that the system manages FAM in a two level hierarchy. At the coarser level programs can create FAM **regions**, which are large blocks of memory. Regions may have non-functional properties (e.g., persistence or resilience properties) associated with them. At the more granular level, memory managers can allocate **data items** that correspond to program data structures within the regions. Data items inherit the non-functional characteristics of the regions within which they are allocated.

FAM is addressed by **descriptors**, which are opaque read-only data structures within applications, and contain sufficient information in them to uniquely locate the corresponding region or data item in FAM. Descriptors can be freely copied and shared across processing elements by the program, and are also portable across program instances. A **metadata service** is used primarily to maintain mappings from user-friendly names to descriptors that locate data in FAM, and also to maintain limited metadata (e.g., region and data item permissions and sizes, region

redundancy levels). Currently, the API leaves the structure of names open. Each region is required to have a unique name associated with it. Data items may optionally have associated names. In line with the two-level memory hierarchy represented by regions and data items, data items within a given region must be named uniquely relative to other data items in that region. If a hierarchical name space is desirable, it can be accommodated within the metadata service implementation with no modifications of the API.

Both regions and data items have access permissions associated with them. Currently, the API supports UNIX®-like permissions for access control to any data item that is long-lived. It leaves open the possibility of using secret authentication tokens, key pairs, or even PKI-based certificates for access control based on security needs of the implementation.

Language Bindings and Conventions

This version of the OpenFAM API uses C11 bindings. Extensions to C++ are expected to be straightforward from the specification, and as the API stabilizes, we expect that bindings in other languages can be provided in the future. Also note that the API in this document should be considered EXPERIMENTAL/UNSTABLE, since implementation and usage will undoubtedly require changes to the API as implementers gain experience with the API.

It is assumed in the API that a recent C11 compiler is available, and generics are used to avoid insertion of data types in method names unless necessary. Method and data type names in the API use underscore separated lower-case words. Data types start with initial upper case letters, while method names and method arguments start with lower case, as do fields within data types. Method parameters and fields within data types use camelCase naming. To maintain separation from other programs, all type and method names have the prefix "Fam_" or "fam_" depending on whether the name represents a data type or a method name. Where multiple methods have the same functionality, and the method signature is also identical (with the exception of data types), we use the C convention of adding "_TYPE" as a suffix to separate out the methods.

In the following sections, we provide the details of the API organized in functional groups. In each case, a high level conceptual representation of the API is provided to give an overview of the anticipated functionality behind the API, followed by the detailed API specified in C11. Throughout the discussion, we assume that the application has appropriate permissions needed to access the regions and data items and that mechanisms are available to handle exceptions if they occur.

We follow the functional categorization provided in the OpenSHMEM specification to segment the API. Note that the description in this document only specifies an API, and implementations may optimize this API for different hardware environments. Also note that we attempt to keep the API ***minimal*** in this specification. As we gain experience with both implementing the libraries, and using them, it is expected that the API will evolve.[2]

[2] Since it is easier to retain backward compatibility when adding functionality rather than removing it, we expect the API to expand over time, rather than trying to cover all possible corner cases up front, and be left with a cumbersome API.

Initialization and Finalization

These routines initialize the OpenFAM environment for an application and enable access to FAM. The routines are accompanied by finalize routines that allow a PE to gracefully terminate its participation in the application. A high-level summary of the initialization and finalization API is shown in Figure 2.

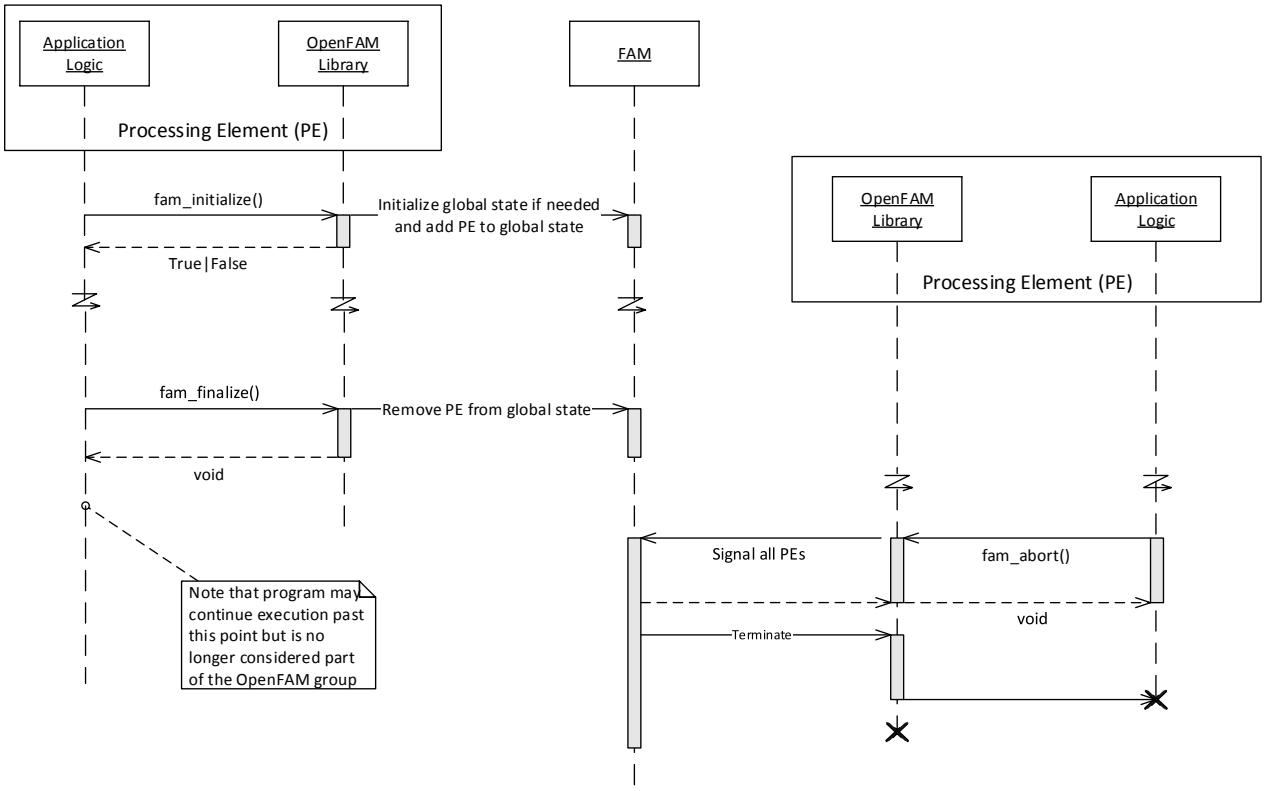

Figure 2: Initialize and Finalize API

At the start of the program the PE calls `fam_initialize()` and provides a group name and other options that are application (or PE) specific. Options represent an extensible data structure determined by the implementation that contains library-specific variables (e.g., those traditionally provided by environment variables). The library creates FAM-resident data structures that are shared across the group for internal use by the OpenFAM library. If the library finds that the required data structures already exist, it simply adds information about the current PE to the data structures. The library returns success or failure to allow graceful termination in case of errors.

When the program finishes (or when a PE wants to leave the overall group) the PE calls the `fam_finalize()` method. This disconnects the PE from the overall FAM environment, although it may continue to run without further involvement of the OpenFAM library.

Finally, a PE may invoke `fam_abort()` with a status flag. This causes a signal to be propagated asynchronously to all PEs in the group indicating that termination of the application is desired. The OpenFAM library will forcibly terminate a PE as the signal is received at that PE. Since data consistency in FAM cannot be assured by the library under this scenario, the programmer should not use this mechanism as a normal termination mechanism—each PE should invoke `fam_finalize()` to gracefully disconnect from the group when it is finished, and terminate using normal process exit mechanisms.

fam_initialize

Initialize the OpenFAM library.

Synopsis

```
int fam_initialize(char *groupName, Fam_Options *options);
```

Description

This method is required to be the first method called when a processing element uses the OpenFAM library. It initializes internal data structures within the library that allow the processing element to participate in the application. Normally, called by all PEs at the start of the program.

Input Arguments

Name	Description
groupName	Name that defines an application group. All cooperating PEs must provide the same group name.
options	Pre-defined options that a processing element provides to the OpenFAM library.

Return Values

Integer value following normal C convention of 0 for successful completion, 1 for unsuccessful completion, and a negative number in case of exception.

Notes

Fields may be specified in options in the future to support parameters or other variables that have traditionally been provided to the runtime via environment variables.

Example

```
#include<stdlib.h>
#include<stdio.h>
#include "fam.h"
int main(void){
	Fam_Options *fm = malloc(sizeof(Fam_Options));
	// assume that no specific options are needed by the implementation
	if(fam_initialize("myApplication", fm) == 0){
		printf("FAM initialized\n");
		fam_finalize("myApplication"); // terminate gracefully
		return 0;
	}
	printf("FAM Initialization failed\n");
	fam_abort(-1); // abort the program
	// note that fam_abort currently returns after signaling
	// so we must terminate with the same value
	return -1;
}
```

fam_finalize

Finalize the OpenFAM library.

Synopsis

```
void fam_finalize(char *groupName);
```

Description

This method is the final method called when a processing element disconnects from the OpenFAM library. It removes the processing element from the group, and frees underlying PE entries in the group's OpenFAM library data structures. Normally called by all PEs when a program ends.

Input Arguments

Name	Description
groupName	Name of the group for the program. Should be the same as that provided to `fam_initialize()`.

Return Values

None.

Example

See `fam_initialize`

fam_abort

Forcibly terminate the program.

Synopsis

```
void fam_abort(int status);
```

Description

This method is used to signal to the OpenFAM library that the PE wishes to abort the program.

Input Arguments

Name	Description
status	Termination status desired by the PE.

Return Values

None.

Notes

The OpenFAM library asynchronously notifies all other cooperating PEs when `fam_abort()` is called, and the method returns before other PEs terminate. Since data consistency in FAM cannot be guaranteed by the library, this method should not be used as the normal exit.

Example

See `fam_initialize`

Query State and Names

These routines provide any PE the ability to query the state being maintained by the OpenFAM library environment.

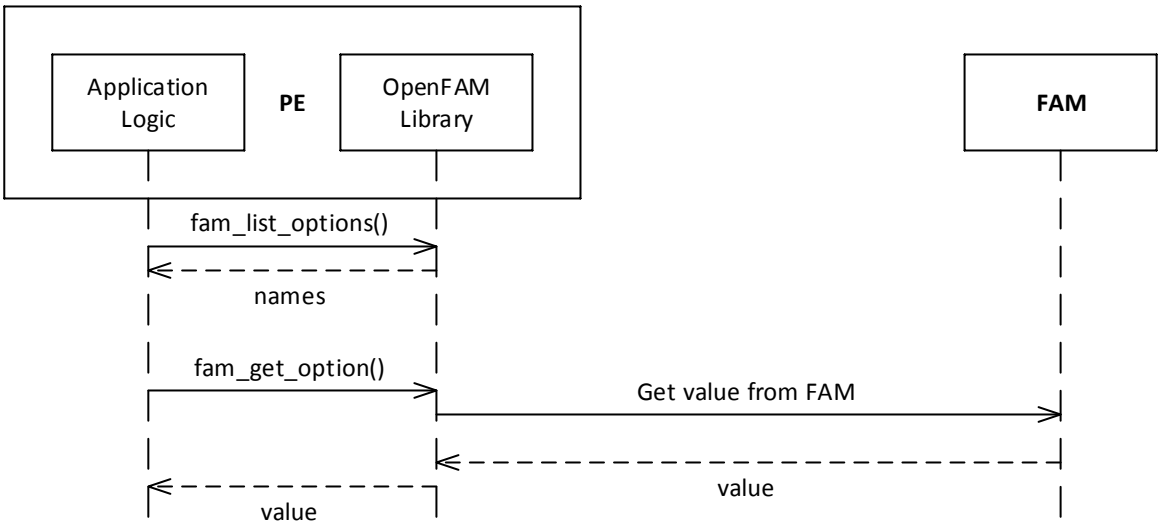

Figure 3: API for querying options

The maintained state may include library and system information (e.g., library version, number of nodes, size of memory etc.) or additional information specific to the executing program available via the OpenFAM library (e.g., number of PEs that have joined the group, maximum region size possible etc.). Rather than embedding specific information within function names, for flexibility and extensibility, the API provides an generic interface for querying such state as {name, value} pairs. However, it should be recognized that the interface accesses a set of pre-configured state variables, and does *not* represent a generic key/value store for use by the application. Figure 3 outlines this API.

`fam_list_options()` allows the application to query the library for currently defined state variables. Any PE can query for the value of the corresponding variable using `fam_get_option()`.

In addition, the query group provides an interface to the metadata service, wherein names may be associated with data items (or regions) in FAM (see allocate APIs later), and the corresponding descriptors be retrieved both across multiple PEs, as well as across program invocations. The API is summarized in Figure 4.

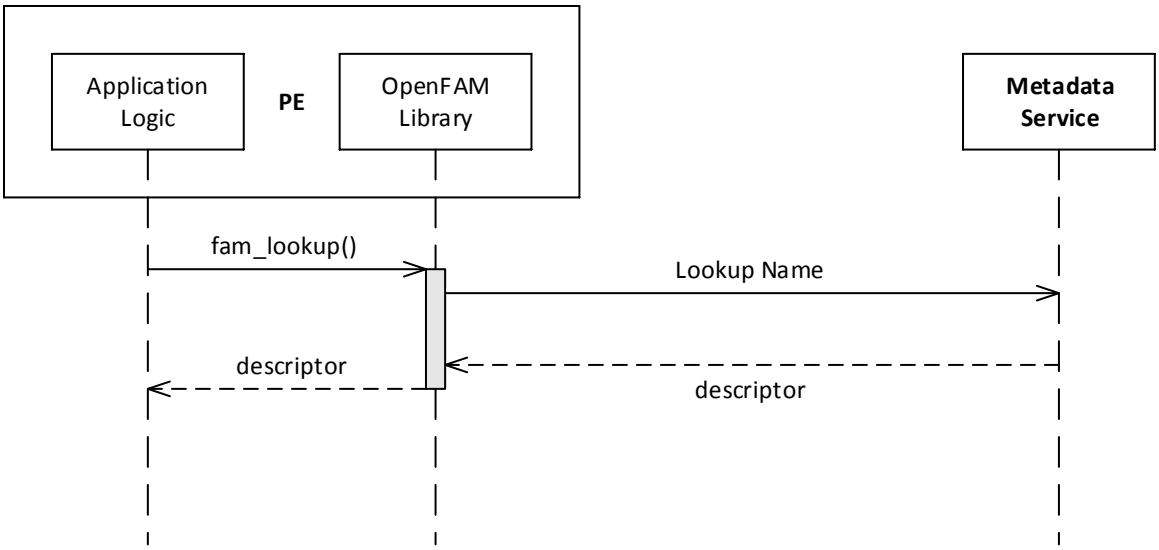

Figure 4: Metadata service API

Regions are automatically registered with the metadata service during creation and unregistered when they are destroyed. Data items can optionally be associated with a name for later retrieval (either within the executing program or within a different program) via `fam_lookup()`. Corresponding to the two-level memory hierarchy (regions and data items), data item names are scoped within the region within which they are created. *Note that if a descriptor is not registered with the metadata service during allocation, then the program itself is responsible for saving and tracking the descriptor if required later, since the data in FAM will be unreachable without the descriptor.* Implementations may optionally choose to automatically register unnamed data items during creation using special names, and/or allow background services to eventually garbage collect data items if their descriptors are not named. However, such services and mechanisms for garbage collection are outside the scope of this specification.

fam_list_options

Get the list of names for pre-defined names in the FAM library.

Synopsis

```
const char **fam_list_options(void);
```

Description

This method can be used to retrieve the names of global variables used within OpenFAM or set using the options argument to `fam_initialize()`. The list is implementation specific.

Input Arguments

None.

Return Values

A null terminated list of names that can be used to retrieve option values.

Example

```
#include<stdio.h>
#include "fam.h"

int main(void){
	// ... initialization code here
	const char ** optionName = fam_list_options();
	while(*optionName != NULL){	// while we have more names
		printf("%s\n", *optionName);	// print the name
		optionName++;
	}
	// ... finalization code here
}
```

fam_get_option

Get the value for a pre-defined option in the FAM library.

Synopsis

```
const void *fam_get_option(char *optionName);
```

Description

This method can be used to retrieve values of state variables maintained within OpenFAM or set using the options argument to `fam_initialize()`. The supported names (and returned types) are implementation specific.

Input Arguments

Name	Description
optionName	Name of the option to be retrieved.

Return Values

A pointer to the value of the option.

Notes

None.

Example

```
#include<stdio.h>
#include "fam.h"

int main(void){
	// ... initialization code here

	// assumes that library version is one of the supported options
	const char * libraryVersion = fam_get_option("VERSION");
	printf("%s\n", libraryVersion);

	// ... finalization code here
}

```

fam_lookup

Look up a descriptor to a region or data item in the metadata service.

Synopsis

```
Fam_Region_Descriptor *fam_lookup_region(char *regionName);
Fam_Descriptor *fam_lookup(char *itemName, char *regionName);
```

Description

These methods look up a descriptor registered with the metadata service using its name.

Input Arguments

Name	Description
itemName	Name of the data item to be looked up.
regionName	Name of the region, or region containing the item.

Return Values

A descriptor to the value in FAM. Null is returned if the caller does not have access rights or if the item or region name was not found by the metadata service.

Notes

See the `fam_create_region()` and `fam_allocate()` APIs later in this document for discussion of regions and data items.

Example

```
#include "fam.h"
int main(void){
	// ... initialization code here

	// get a region descriptor from the metadata service
	Fam_Region_Descriptor *region = fam_lookup_region("myRegion");

	// Get a descriptor (created by another PE) via the metadata service
	Fam_Descriptor *arrayDescriptor = fam_lookup("myArray", "myOtherRegion");

	// ... continuation code here
}
```

Memory Allocation and Deallocation

These routines provide mechanisms for allocating/deallocating FAM regions as well as data items within those regions. The API provides a two-level hierarchy for memory allocation:

1. **Regions** represent (large) containers within FAM that have associated non-functional properties, such as desired resilience levels, which may be specified by the user when the region is created. All data items within a region have the same resilience as the region. Regions are identified using region identifiers maintained by the underlying system. Management services use regions as the allocation units to reserve large blocks of FAM. Quotas may be enforced by management services to limit the total size of FAM allocated to a given user or a given application. Users may request additional space (or may reduce space) for a region by resizing the region.
2. **Data items** represent (smaller) areas of FAM that are allocated within the regions. Each region provides a heap allocator abstraction, which can be used to allocate individual data items. Management services may delegate per-region heap allocator functionality to distributed brokers within operating systems or other middleware. Currently the OpenFAM API does not support resizing (or automated re-packing) of data items within the region. This constraint may be relaxed in the future as implementations become available.

As mentioned earlier, FAM is referenced through a descriptor (much like a file descriptor), not a standard pointer that can be de-referenced to access the content directly. The allocation API returns opaque (to the application) read-only descriptors that are portable in the sense that they can be used by any PE to access the data. Descriptors enable applications to specify global/virtual references that will work for any PE regardless of where the region may be mapped into the PE's virtual address space.

Because both regions and data items can be long-lived, access permissions are associated with both allocated data items and regions in FAM. Currently, the API assumes that standard Unix® permissions can be used, although implementations may choose additional mechanisms for managing access. Permissions can be changed if desired (assuming that the requestor has the appropriate access rights). Since OpenFAM deals with FAM that is accessible directly from any PE, there is no direct association of allocated FAM with a specific PE. Assuming appropriate access rights, any PE has the ability to allocate data items in FAM and/or use data items allocated by other PEs, enabling much more efficient mechanisms for passing data between PEs and even within larger computational workflows consisting of multiple stages.

Figure 5 and Figure 6 provide an overview of these APIs. `fam_create_region()` creates a new region in FAM for use by the application, and returns a descriptor that can be used portably across PEs (or programs in a larger workflow) to reference that region. Regions are named and registered with the metadata service when they are created to enable PEs to retrieve the descriptor associated with the region using user-friendly names. Regions may be resized using `fam_resize_region()` operation. If the region size is reduced, data items that reside within the truncated part of the region will be lost. Region size can be safely increased without affecting existing data items within it. Finally regions can be destroyed using `fam_destroy_region()`, which deallocates the region (and all data items within it) and frees the corresponding FAM. Deallocation occurs asynchronously after the `fam_destroy_region()` call to allow other PEs using the region to finish. The behavior of the library if a PE accesses a region (or data item) after it has been destroyed (or deallocated) by a different PE is implementation dependent.

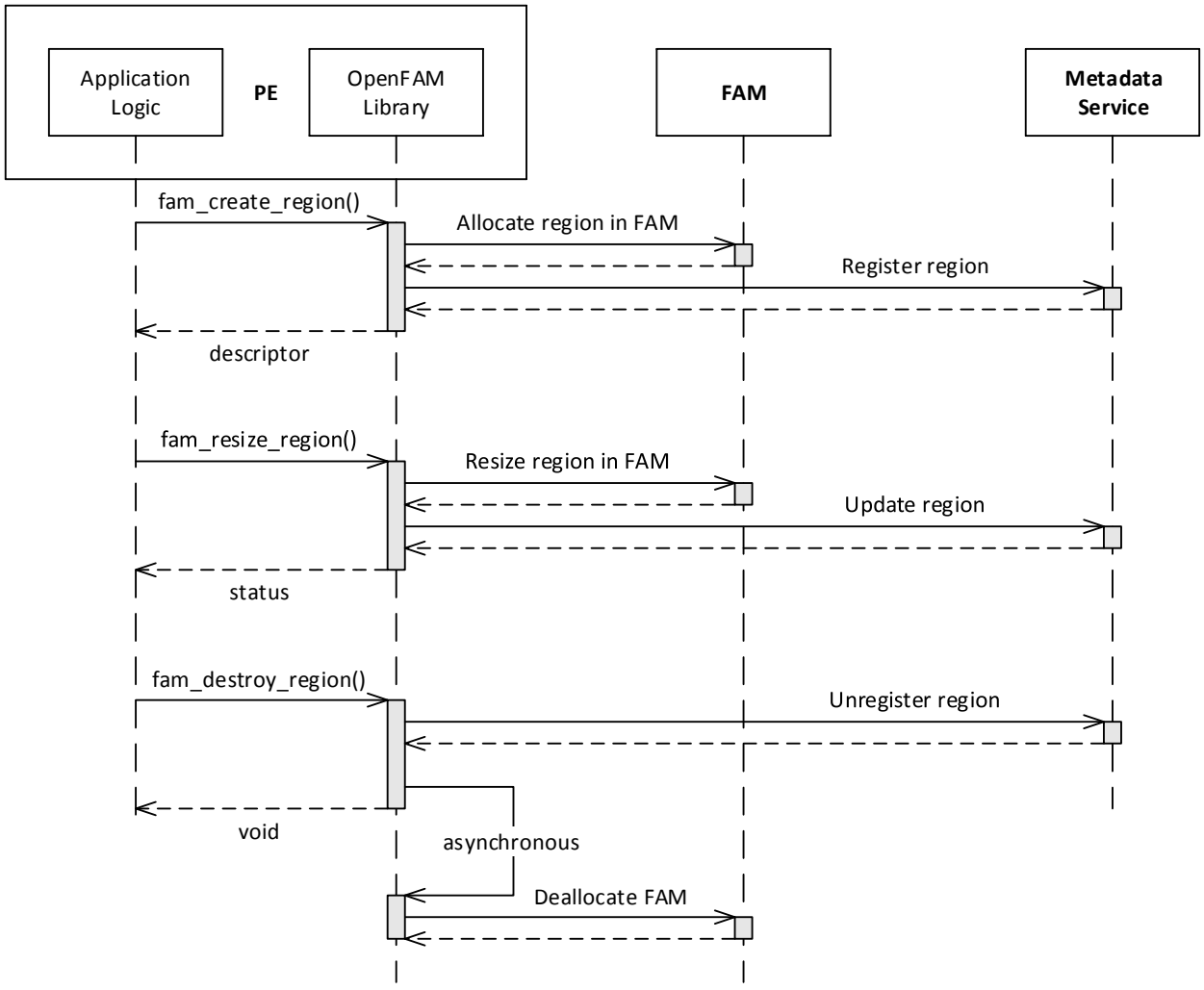

Figure 5: Creation and destruction of FAM regions

Space for data items within a region can be allocated using `fam_allocate()`, which returns a descriptor to that data item, and de-allocated by calling `fam_deallocate()`. Optionally, data items may be named to simplify access to them by other PEs or programs after allocation. The application is responsible for de-allocating unnamed data items or ensuring that the corresponding descriptors are persisted by the application.

Finally, assuming that the caller has the correct access rights, permissions on either a region or a data item can be changed using `fam_change_permissions()`. As mentioned earlier, the actual semantics of permissions and how they are implemented or enforced is system dependent, and outside the scope of this specification. The high-level view of this API is shown in Figure 7.

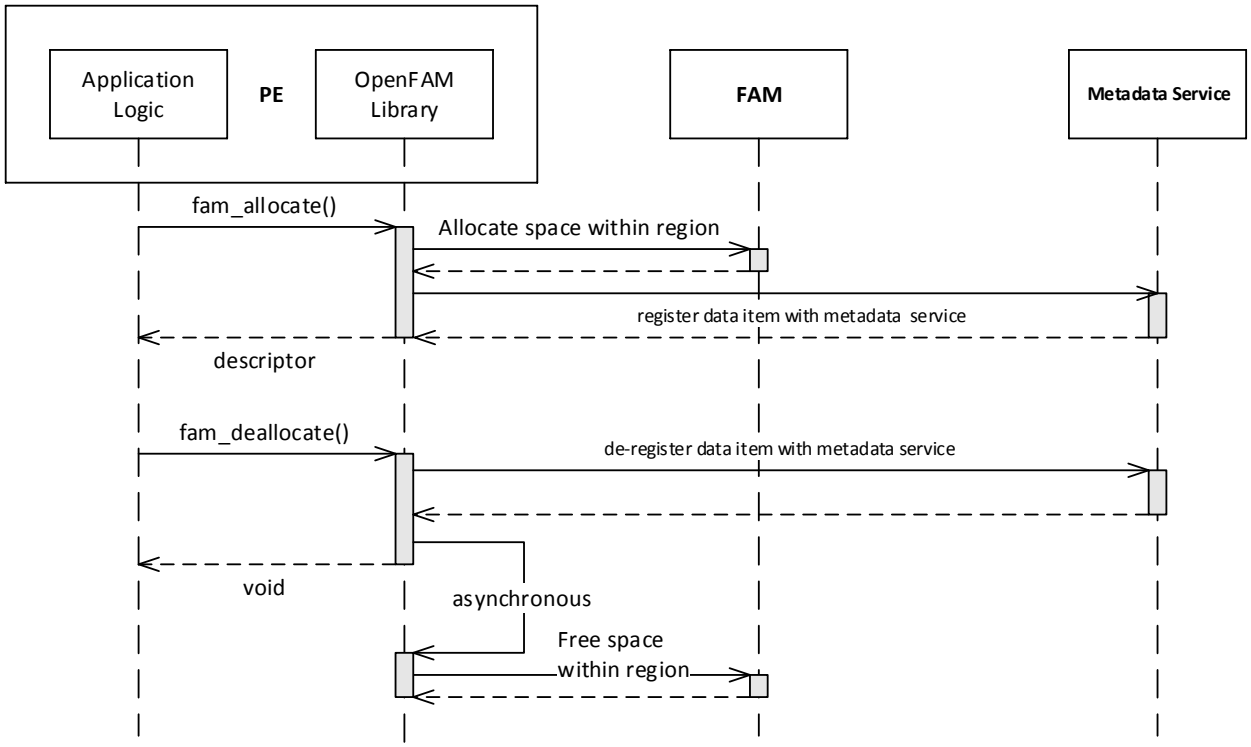

Figure 6: Allocation and deallocation of data items

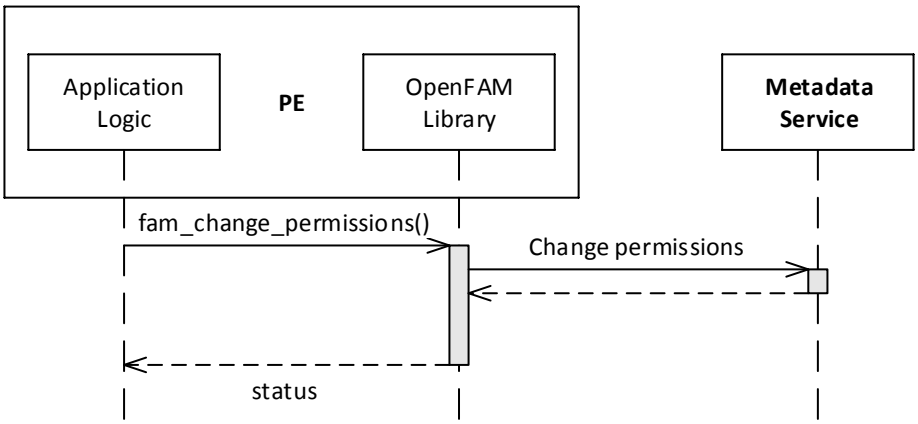

Figure 7: Change permissions

fam_create_region

Create a new region in FAM.

Synopsis

```
Fam_Region_Descriptor *fam_create_region(char *name, uint64_t size, mode_t
permissions, Fam_Redundancy_Level redundancyLevel, ...);
```

Description

This method creates a region in FAM for subsequent use

Input Arguments

Name	Description
name	Name of the region to be created.
size	Requested size of the region in bytes.
permissions	Permissions to be associated with this region.
redundancyLevel	Desired redundancy level.

Return Values

Descriptor to the created region. Null if region could not be created.

Notes

Note that the system may round up the actual size of the region to align memory boundaries. Currently the maximum size of a region is limited to 16 EiB by the API.

Example

```
#include <stdlib.h>
#include "fam.h"
int main(void){
	// ... Initialization code here
	// create a 10 GB region with 0777 permissions and RAID5 redundancy
	Fam_Region_Descriptor *rd =
		fam_create_region("myRegion", (uint64_t)10000000000, 0777, RAID5);
	if(rd != NULL){
		// use the created region
	} else {
		// error handling
	}
	// ... continuation code here
	// we are finished. Destroy the region and everything in it
	fam_destroy_region(rd);
}
```

fam_destroy_region

Destroy an existing region in FAM.

Synopsis

```
void fam_destroy_region(Fam_Region_Descriptor *descriptor);
```

Description

This destroys a region in FAM, and asynchronously frees space allocated to the region.

Input Arguments

Name	Description
descriptor	Descriptor of the region to be destroyed.

Return Values

None.

Notes

If a region is destroyed, all data items in it are also destroyed. If descriptors to the region or the contained data items are subsequently used, the behavior of the library is implementation dependent. Note that this call will trigger a delayed free operation, to permit other PEs currently using the region to finish.

Example

See `fam_create_region`

fam_resize_region

Resize an existing region in FAM

Synopsis

```
int fam_resize_region(Fam_Region_Descriptor *descriptor, uint64_t nbytes);
```

Description

This method resizes a region in FAM.

Input Arguments

Name	Description
descriptor	Descriptor of the region to be resized.
nbytes	New requested size of the region.

Return Values

Integer value following normal C convention of 0 for successful completion, 1 for unsuccessful completion, and a negative number in case of exception.

Notes

Regions may safely be increased in size (subject to system-wide limits). Reducing the region size (although allowed) implies that any data items in the truncated part of the region will be lost. The behavior of the library if those items are accessed later is implementation dependent.

Example

```
#include<stdlib.h>
#include "fam.h"
int main(void){
	// ... Initialization code here
	// create a 10 GB region with 777 permissions and RAID5 redundancy
	Fam_Region_Descriptor *rd =
		fam_create_region("myRegion", (uint64_t)10000000000, 0777, RAID5);
	if(rd != NULL){
		// resize the region to 20 GB
		fam_resize_region(rd, (uint64_t)20000000000);
		// use the re-sized region
	} else {
		// error handling
	}
	// ... continuation code here
}
```

fam_allocate

Allocate a data item in a region of FAM.

Synopsis

```
Fam_Descriptor *fam_allocate(char *name, uint64_t nbytes, mode_t accessPermissions,
Fam_Region_Descriptor *region);
Fam_Descriptor *fam_allocate(uint64_t nbytes, mode_t accessPermissions,
Fam_Region_Descriptor *region);
```

Description

These methods allocate space for a data item in some region of FAM.

Input Arguments

Name	Description
name	Name of the data item for subsequent references.
nbytes	Requested size of the data item in bytes.
accessPermissions	Access permissions for this data item.
region	Descriptor for the region where the data item is being created. If null, the default specified during `fam_initialize()` is used.

Return Values

A descriptor to the data item created. Null in case of errors.

Notes

Note that these methods may allocate space greater than the requested size to maintain an implementation dependent byte alignment. The allocated space is zeroed out during allocation. If a name is provided during allocation, it can be used within subsequent calls to `fam_lookup()` to retrieve the associated descriptor. If a name is not provided during allocation, then the program itself is responsible for saving and tracking the descriptor if required later, since the data in FAM will be unreachable without the descriptor.

Example

```
#include<stdlib.h>
#include "fam.h"
int main(void){
	// ... Initialization code here
	Fam_Region_Descriptor *region = fam_lookup_region("myRegion");
	// create 50 element unnamed integer array in FAM with 0600
	// (read/write by owner) permissions in myRegion
	Fam_Descriptor *descriptor =
			fam_allocate((uint64_t)(50 * sizeof(int)), 0600, region);
	if(descriptor == NULL){
		// error handling
	}
	// free allocated space in FAM
	fam_deallocate(descriptor);
}
```

fam_deallocate

Deallocate space being used by a data item in a region of FAM.

Synopsis

```
void fam_deallocate(Fam_Descriptor *descriptor);
```

Description

This method triggers a deallocation of previously allocated space for a data item in some region of FAM.

Input Arguments

Name	Description
descriptor	Descriptor associated with the space to deallocate.

Return Values

None.

Notes

Note that if a descriptor is used to access deallocated space, behavior of the library is implementation dependent. The implementation may reclaim space some indeterminate amount of time after the method is called, to permit other PEs currently using the data item to finish.

Example

See `fam_allocate`

fam_change_permissions

Change permissions associated with a data item or region in FAM.

Synopsis

```
int fam_change_permissions(Fam_Descriptor *descriptor, mode_t accessPermissions);
int fam_change_permissions(Fam_Region_Descriptor *descriptor, mode_t
accessPermissions);
```

Description

This method changes read/write permissions for a data item or region.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item or region.
accessPermissions	New access permissions to be associated with the item.

Return Values

Integer value following normal C convention of 0 for successful completion, 1 for unsuccessful completion, and a negative number in case of exception.

Notes

None.

Example

```
#include<stdlib.h>
#include "fam.h"
int main(void){
	// ... Initialization code here
	Fam_Region_Descriptor *region = fam_lookup_region("myRegion");
	// create 50 element named integer array in FAM with 0600
	// (read/write by owner) permissions in myRegion
	Fam_Descriptor *descriptor =
			fam_allocate("myItem", 50*sizeof(int), 0600, region);
	if(descriptor != NULL){
		// initialize the allocated space here with meaningful values
		// make the item read-only (0400) to prevent further change by
		// subsequent programs
		fam_change_permissions(descriptor, 0400);
	} else {
		// error handling
	}
}
```

Data Read and Write

This group of APIs provide routines to read/write FAM data. Since hardware support for cache coherence is currently not assumed across processing nodes, the programmer must provide appropriate synchronization mechanisms at the application level if two nodes access the same data items in FAM at the same time. The API is broken into four groups:

1. A **get/put** API supports movement of data between FAM and local memory.
2. A **map/unmap** API provides the ability to directly map data items in FAM into the local process virtual address space, enabling direct load-store access from the CPU to data in FAM.
3. The third group provides the ability to **gather (scatter)** data from (to) disjoint parts of FAM to (from) a contiguous array in local memory.
4. Finally, a **copy** API provides a hardware-assisted mechanism to replicate data from one part of FAM to another part of FAM.

These APIs are outlined in Figure 8 - Figure 10.

`fam_put` creates a copy of an object from local memory into FAM. Similarly `fam_get` creates a copy of an object from FAM into local memory. Note that subsequent modifications to the two copies are made independently, and no synchronization should be assumed between the two copies. Both APIs specify data movement in bytes, enabling movement of smaller chunks of data within larger data items (such as parts of a large array resident in FAM). Both `fam_put` and `fam_get` have blocking and non-blocking variants. The non-blocking calls return once the transfer has been initiated, and don't wait for completion. For example, `fam_put_nonblocking()` returns once data has been dispatched to FAM, and does not wait until the object has reached fabric-attached memory. Thus if multiple calls to `fam_get_nonblocking()` or `fam_put_nonblocking()` are made in succession, the order of data delivery is not guaranteed. To ensure ordering and completion, either blocking versions of the API or memory ordering operations (defined later) should be used. Non-blocking calls permit overlap of the round-trip latencies to FAM for multiple related operations, thereby improving efficiency over a serialized set of blocking operations. The blocking call variants wait until the data transfer has been completed. For example, `fam_get_blocking() waits until data` is delivered to local memory.

`fam_map()` and `fam_unmap()` directly map and unmap FAM addresses into the local process virtual address space—no copy exists in local memory in this case. These APIs provide a very programmer friendly mechanism for addressing data in FAM by enabling the processor to directly use load/store instructions to access FAM. While this makes programming much simpler (the application can treat FAM the same way it treats local memory), there may be performance differences associated with the use of the two APIs (get/put versus map/unmap) depending on the access patterns and characteristics of the underlying hardware, and the programmer should use the API that makes sense for the computation and memory accesses of the application.

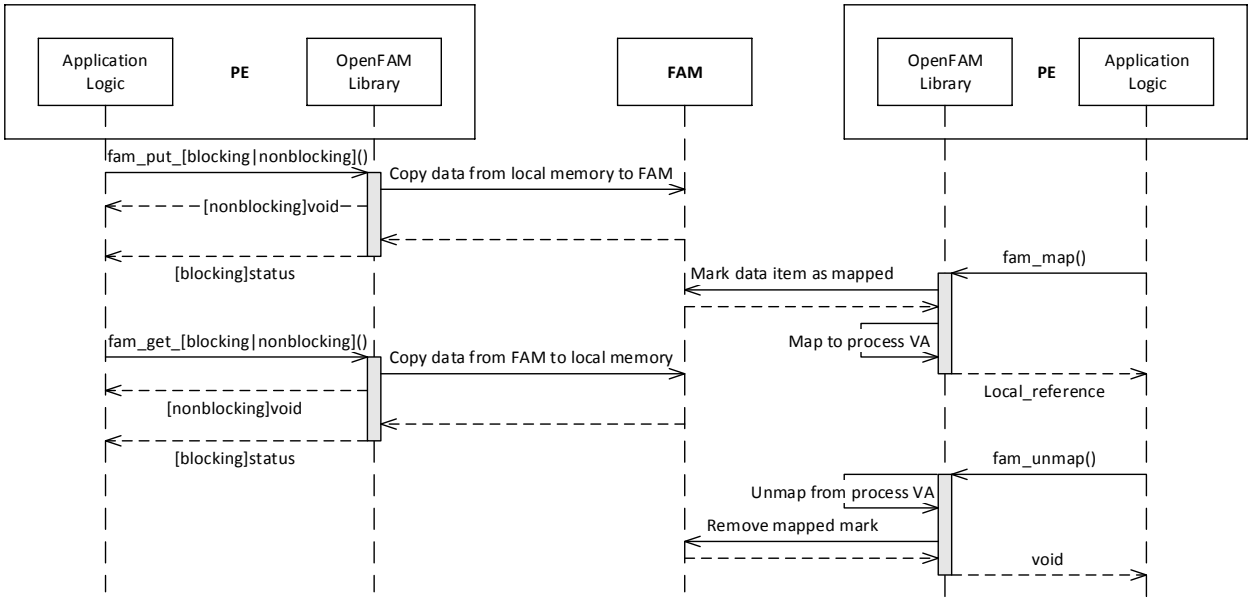

Figure 8: Data transfer APIs

Both get/put and map/unmap APIs deal with contiguous bytes in FAM. To support commonly used data access patterns, the API also provides the ability to gather elements from arrays being held in some data item in FAM into a contiguous part of local memory, and scatter them back from a contiguous part of local memory to disjoint parts of the array in FAM. These operations are outlined in Figure 9. A `fam_gather` operation allows the programmer to specify a starting element, a stride, the number of elements required, and the size of each element. The library then makes multiple parallel copies from FAM to adjacent regions of local memory. Conversely, a `fam_scatter` operation copies array elements placed sequentially in local memory, and scatters them in parallel to disjoint parts of the corresponding array in FAM. As with `fam_get` and `fam_put`, the OpenFAM API provides both blocking and non-blocking variants of `fam_gather` and `fam_scatter`.

The API also allows the programmer to specify an indexed gather/scatter operation. In an indexed operation, the stride between consecutive array elements is not assumed constant, but can be specified as an index array.

Initially, the API restricts these methods to gather and scatter elements from a single data item (e.g., parts of a large array). Subsequent versions of the API may also provide the ability to access elements from multiple data items if necessary.

Finally, as shown in Figure 10, `fam_copy()` supports direct movement of data between two parts of FAM, possibly from one region of FAM to another region of FAM. `fam_copy()`is non-blocking. No synchronization is assumed between the copies, and they can be modified independently after the copy is complete.

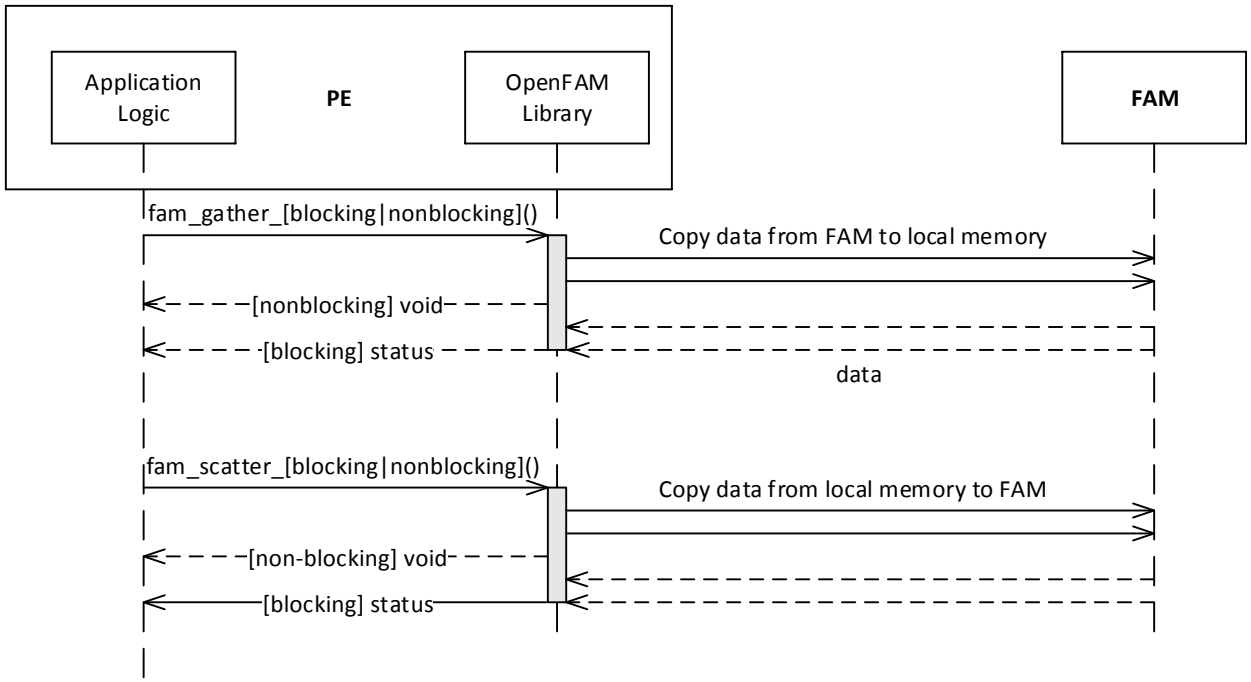

Figure 9: Gather/Scatter APIs

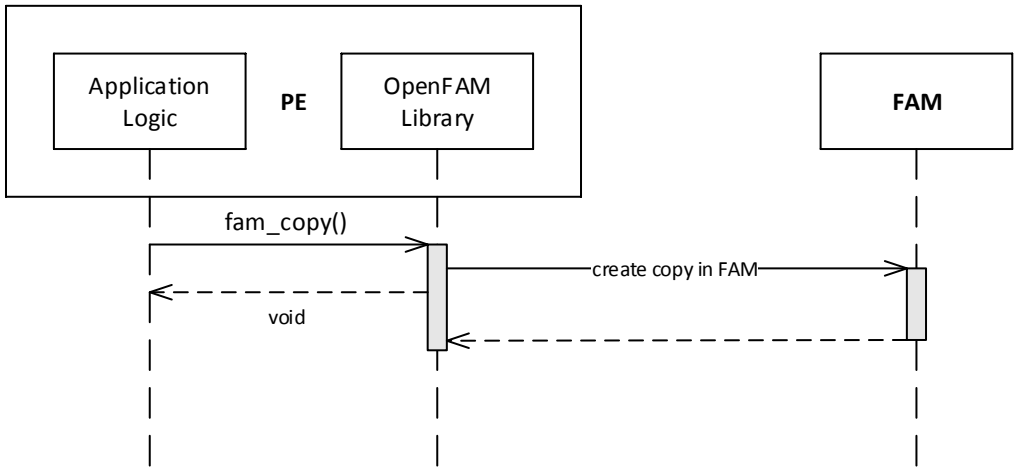

Figure 10: FAM Copy API

fam_get

Copy a segment of memory from FAM to local memory.

Synopsis

```
int fam_get_blocking(void *local, Fam_Descriptor *descriptor, uint64_t offset,
uint64_t nbytes);
void fam_get_nonblocking(void *local, Fam_Descriptor *descriptor, uint64_t offset,
uint64_t nbytes);
```

Description

Copy (part of) some data item from FAM to local memory.

Input Arguments

Name	Description
local	Appropriately sized area of local memory to receive data.
descriptor	Descriptor associated with the data item in FAM.
offset	Byte offset from the start of the data item in FAM.
nbytes	Number of bytes to copy from FAM to local memory.

Return Values

The non-blocking call does not return a value. The blocking call returns an integer value following normal C convention of 0 for successful completion, 1 for unsuccessful completion, and a negative number in case of exception.

Notes

The local memory pointer must point to an area of memory sufficient to accommodate the incoming data (nbytes). Insufficient allocation of local memory may cause segmentation violations or corrupted data in local memory.

At return, the memory pointed to by `local` contains a copy of the data in FAM for the blocking call. For the non-blocking call, the memory pointed to by the local pointer is not guaranteed to contain a copy of the data in FAM until after a successful completion of a `fam_quiet()` call.

Examples

```
// Example of blocking get

#include<stdlib.h>
#include "fam.h"

int main(void){
	// ... Initialization code here
	// look up the descriptor to a previously allocated data item
	// (a 50 element integer array)
	Fam_Descriptor *descriptor = fam_lookup("myItem", "myRegion");
	if(descriptor != NULL){
		// allocate local memory to receive 10 elements
		int *local = malloc(10 * sizeof(int));
		// copy elements 6-15 from FAM into elements 0-9 in local memory
```

```
                fam_get_blocking(local, descriptor, 6*sizeof(int), 10*sizeof(int));
                // ... we now have a copy in local memory to work with
        } else {
                // error handling
        }
}
// Example of non-blocking get
#include<stdlib.h>
#include "fam.h"

int main(void){
        // ... Initialization code here
        // look up the descriptors to two previously allocated data items
        // (50 element integer array)
        Fam_Descriptor *descriptor = fam_lookup("myItem", "myRegion");
        if(descriptor != NULL){
                // allocate local memory to receive 10 elements from different offsets
within the data item
                int *local1 = malloc(10 * sizeof(int));
                int *local2 = malloc(10 * sizeof(int));
                // copy elements 6-15 from FAM into local1 elements 0-9 in local memory
                fam_get_nonblocking(local1, descriptor1, 6*sizeof(int), 10*sizeof(int));
                // copy elements 26-35 from FAM into local2 elements 0-9 in local memory
                fam_get_nonblocking(local2, descriptor1, 26*sizeof(int),
        10*sizeof(int));
                // wait for previously issued non-blocking operations to complete
                fam_quiet();
                // ... we now have copies in local memory to work with
        } else {
                // error handling
        }
}
```

fam_put

Copy a segment of memory from local memory to FAM.

Synopsis

```
int fam_put_blocking(void *local, Fam_Descriptor *descriptor, uint64_t offset,
uint64_t nbytes);
void fam_put_nonblocking(void *local, Fam_Descriptor *descriptor, uint64_t offset,
uint64_t nbytes);
```

Description

Copy (part of) some data item from local memory to FAM.

Input Arguments

Name	Description
local	Appropriately sized area of local memory to use as source.
descriptor	Descriptor associated with the data item in FAM for the destination.
offset	Byte offset from the start of the data item in FAM.
nbytes	Number of bytes to copy.

Return Values

The non-blocking call does not return a value; the blocking call returns an integer value following normal C convention of 0 for successful completion, 1 for unsuccessful completion, and a negative number in case of exception.

Notes

See notes for `fam_get`.

Example

```
#include<stdlib.h>
#include "fam.h"
int main(void){
	// ... Initialization code here
	// look up the descriptor to a previously allocated data item
	// (a 50 element integer array)
	Fam_Descriptor *descriptor = fam_lookup("myItem", "myRegion");
	if(descriptor != NULL){
		// allocate an integer array and initialize it
		int local[] = {0,1,2,3,4,5,6,7,8,9};
		// replace elements 6-15 of the data item in FAM with values in local
memory
		fam_put_blocking(local, descriptor, 6*sizeof(int), 10*sizeof(int));
		// ... we now have completed a copy from local memory to FAM
	} else {
		// error handling
	}
}
```

fam_map

Map a data item in FAM to the process virtual address space.

Synopsis

```
void *fam_map(Fam_Descriptor *descriptor);
```

Description

This method maps a data item in FAM to the PE's virtual address space.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item or region in FAM.

Return Values

A pointer in the PE's virtual address space that can be used to directly manipulate contents of FAM.

Notes

Note that if multiple PEs simultaneously access a data item in FAM when at least one has mapped it, they are responsible for enforcing concurrency control and for ensuring that processor caches are managed appropriately. The implementation may limit the FAM size (hence the size of data items) that can be mapped into the processor address space at one time.

Example

```
#include<stdio.h>
#include "fam.h"
int main(void){
	// ... Initialization code here
	// look up the descriptor to a previously allocated data item
	// (an integer array with at least 10 elements)
	Fam_Descriptor *descriptor = fam_lookup("myItem", "myRegion");
	// map it into our local address space
	int *fam_array = (int *) fam_map(descriptor);
	// now we can directly manipulate the data in FAM
	for(int i = 0; i < 10; i++){
		printf("%d\n", fam_array[i]);    // print it
		fam_array[i]++;           // increment it
	}
	fam_unmap(fam_array, descriptor);      // we are done...
	// ... subsequent code here
}
```

fam_unmap

Unmap a data item in FAM from the process virtual address space.

Synopsis

```
void fam_unmap(void *local, Fam_Descriptor *descriptor);
```

Description

This method unmaps a data item in FAM from the process virtual address space.

Input Arguments

Name	Description
local	Local pointer to be unmapped.
descriptor	Descriptor associated with the data item in FAM to be unmapped.

Return Values

None.

Notes

None.

Example

See example under `fam_map`

fam_gather

Copy disjoint elements of a data item from FAM to local memory.

Synopsis

```
int fam_gather_blocking(void *local, Fam_Descriptor *descriptor, uint64_t nElements,
uint64_t firstElement, uint64_t stride, uint64_t elementSize);
int fam_gather_blocking(void *local, Fam_Descriptor *descriptor, uint64_t nElements,
uint64_t *elementIndex, uint64_t elementSize);
void fam_gather_nonblocking(void *local, Fam_Descriptor *descriptor, uint64_t
nElements, uint64_t firstElement, uint64_t stride, uint64_t elementSize);
void fam_gather_nonblocking(void *local, Fam_Descriptor *descriptor, uint64_t
nElements, uint64_t *elementIndex, uint64_t elementSize);
```

Description

Copy elements of a data item from FAM to local memory based on a constant or indexed stride.

Input Arguments

Name	Description
local	Pointer to appropriately sized area of local memory.
descriptor	Descriptor associated with the data item in FAM.
nElements	Number of elements to get.
firstElement	Index (in FAM) of the first element within the data item to get.
elementIndex	An array containing element indexes.
stride	Stride to use when getting elements.
elementSize	Size of each element to gather.

Return Values

The non-blocking calls do not return a value. The blocking calls return an integer value following normal C convention of 0 for successful completion, 1 for unsuccessful completion, and a negative number in case of exception. At return, the memory pointed to by `local` contains a copy of the data elements from FAM for the blocking calls. For non-blocking calls, local memory is not guaranteed to contain data until successful completion of a subsequent `fam_quiet()` call.

Notes

The local memory pointer must point to an area of memory sufficient to accommodate the incoming data. Insufficient allocation of local memory may cause segmentation violations or corrupted data in local memory. This API assumes that the data item contains uniformly sized elements (e.g, a large array). Currently, the API gathers elements from within a single data item.

Example

```
#include<stdlib.h>
#include "fam.h"
int main(void){
	Fam_Descriptor *descriptor = fam_lookup("myItem", "myRegion");
	if(descriptor != NULL){
		// allocate a 25-element integer array in local memory
		int *local = malloc(25 * sizeof(int));
		// gather all odd elements from myItem into local memory
		// first element is myItem[1]; collect 25 elements with stride 2
```

```
                fam_gather_blocking(local, descriptor, 25, (uint64_t) 1, 2,
sizeof(int));
                // gather myItem[1,7,13,15,16]
                int *indexedLocal = malloc(5 * sizeof(int));
                uint64_t indexes[] = {1, 7, 13, 15, 16};
                fam_gather_blocking(indexedLocal, descriptor, 5, indexes, sizeof(int));
                // ... we now have the correct elements in local memory
        } else {
                // error handling
        }
}
```

fam_scatter

Copy elements of a data item from local memory to disjoint parts of FAM.

Synopsis

```
int fam_scatter_blocking(void *local, Fam_Descriptor *descriptor, uint64_t nElements,
uint64_t firstElement, uint64_t stride, uint64_t elementSize);
int fam_scatter_blocking(void *local, Fam_Descriptor *descriptor, uint64_t nElements,
uint64_t *elementIndex, uint64_t elementSize);
void fam_scatter_nonblocking(void *local, Fam_Descriptor *descriptor, uint64_t
nElements, uint64_t firstElement, uint64_t stride, uint64_t elementSize);
void fam_scatter_nonblocking(void *local, Fam_Descriptor *descriptor, uint64_t
nElements, uint64_t *elementIndex, uint64_t elementSize);
```

Description

Copy elements of a data item from local memory to FAM based on a constant or indexed stride.

Input Arguments

Name	Description
local	Pointer to appropriately sized area of local memory.
descriptor	Descriptor associated with the data item in FAM.
nElements	Number of elements to put.
firstElement	Index (in FAM) of the first element within the data item to put.
elementIndex	A local array containing element indexes in FAM.
stride	Stride to use when putting elements.
elementSize	Size of each element to scatter.

Return Values

The non-blocking calls do not return a value. The blocking calls return an integer value following normal C convention of 0 for successful completion, 1 for unsuccessful completion, and a negative number in case of exception. At return, FAM contains a copy of the data elements for the blocking calls. Data in FAM is not guaranteed until successful completion of a subsequent `fam_quiet()` call for the non-blocking calls.

Notes

This API assumes that the data item contains uniformly sized elements (e.g, a large array). Currently, the API scatters elements within a single data item. Note that two versions of each API exist—a strided gather/scatter that assumes a constant stride and an indexed gather/scatter that uses an index array to gather/scatter the elements.

Example

```
#include "fam.h"
int main(void){
	// ... Initialization code here
	// look up the descriptor to a previously allocated data item
	// (a 50-element integer array)
	Fam_Descriptor *descriptor = fam_lookup("myItem", "myRegion");
	// local data
	int local[] = {0,1,2,3,4};
	// scatter the data into myItem[1,3,5,7,9]
	fam_scatter_blocking(local, descriptor,  5L, (uint64_t) 1, 2, sizeof(int));
	uint64_t indexes[] = {10, 17, 13, 15, 16};
	// in addition, place them in myItem[10,17,13,15,16] respectively
```

```
        fam_scatter_blocking(local, descriptor, 5L, indexes, sizeof(int));
        // ... we now have the correct elements in FAM
}
```

fam_copy

Create a second copy of a data item in FAM.

Synopsis

```
void fam_copy(Fam_Descriptor *src, uint64_t srcOffset, Fam_Descriptor *dest, uint64_t
destOffset, uint64_t nbytes);
```

Description

Creates a second copy of some data item in FAM and returns a descriptor to the new copy.

Input Arguments

Name	Description
src	Descriptor associated with the source data item in FAM.
srcOffset	Offset within the source descriptor where copy starts.
dest	Descriptor associated with the destination data item in FAM.
destOffset	Offset within the destination descriptor where the copy starts.
nbytes	Number of bytes to copy.

Return Values

None.

Notes

Note that the method is non-blocking: it returns after the copy has been initiated, and does not wait for completion of the transfer. Also note that `fam_copy()` performs an inconsistent copy, in that the copied data may reflect updates that are performed concurrently with the copy operation, rather than a point-in-time snapshot of the source data item that is consistent as of the beginning of copy operation. If the application desires a consistent copy, it is responsible for coordinating PE activity during the copy.

Example

TBD

Atomics

The atomics APIs provide mechanisms to ensure that operations in FAM are done in an atomic (all-or-nothing) fashion, i.e., do not result in torn reads or writes when the same location in FAM is accessed at the same time from different processing elements. Atomic operations are broken into two categories: non-fetching (which update data in FAM without returning a result) and fetching (which update data in FAM and return a result). Atomic operations that return values block until the operation completes, while those that do not return data values simply dispatch the operation to FAM, but do not wait for acknowledgements before returning (i.e., are non-blocking). These categories are shown in in Figure 11.

In the initial version of the API, non-fetching operations include `fam_set()`, `fam_add()`, `fam_subtract()`, `fam_min()`, `fam_max()`,`fam_and()`, `fam_or()`, and `fam_xor()`. Fetching operations include `fam_fetch()`, `fam_swap()`, `fam_compare_swap()`, `fam_fetch_add()`, `fam_fetch_subtract()`, `fam_fetch_min()`, `fam_fetch_max()`,`fam_fetch_and()`, `fam_fetch_or()`, and `fam_fetch_xor()`

`fam_set()`, `fam_fetch()`, `fam_swap()`, and arithmetic operators (fetching and non-fetching add, subtract, min, and max) are specified for 32-bit (int, float) and 64-bit (long, double) signed and unsigned operands. Logical operators (fetching and non-fetching and, or and xor) are specified for 32-bit (int) and 64-bit (long) unsigned operands.

Compare and swap (CAS) operations are specified for 32-bit (int) and 64-bit (long) signed and unsigned operands. In addition, the API defines a 128-bit double wide (long long) compare and swap.

Note that all atomics require hardware and fabric support, so their availability within an implementation is hardware dependent. Thus implementations may provide a subset (or superset) of the operations described in this section depending on the underlying hardware.

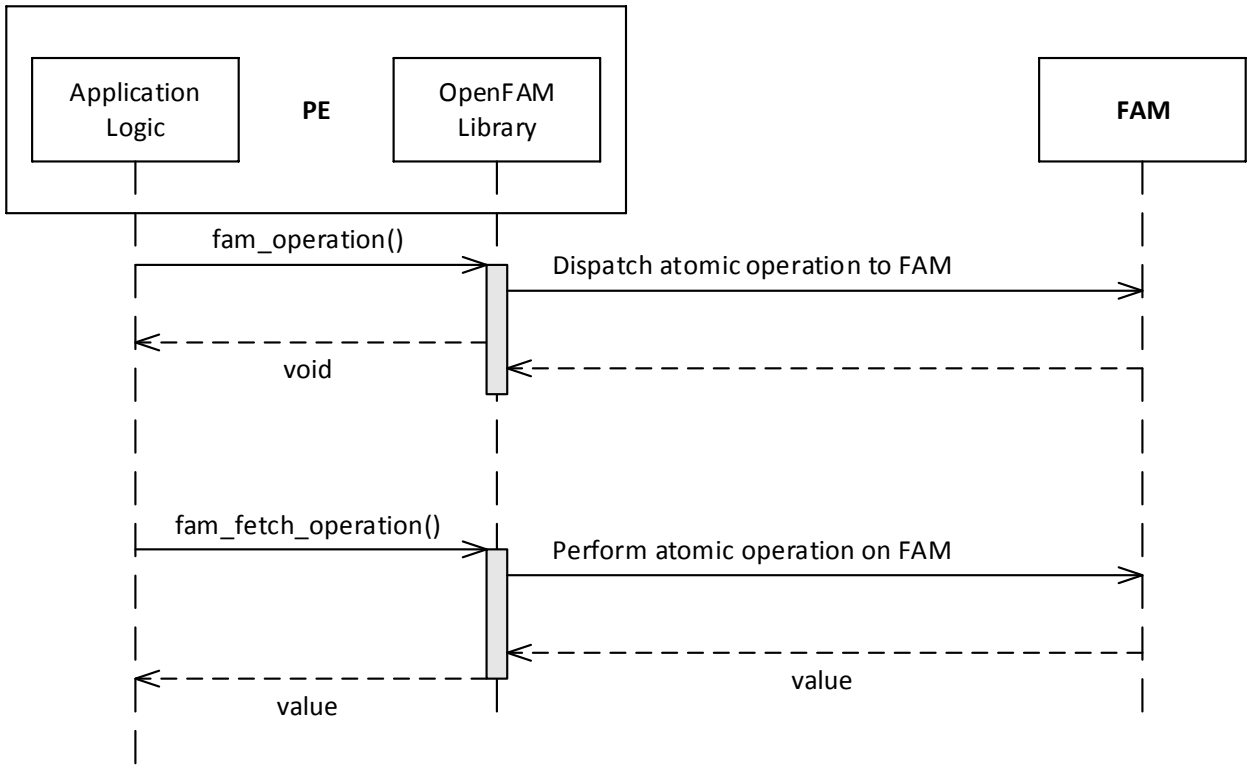

Figure 11: Atomic memory operations

fam_set

Atomically set a value in FAM.

Synopsis

```
void fam_set(Fam_Descriptor *descriptor, uint64_t offset, int32_t value);
void fam_set(Fam_Descriptor *descriptor, uint64_t offset, int64_t value);
void fam_set(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
void fam_set(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
void fam_set(Fam_Descriptor *descriptor, uint64_t offset, float value);
void fam_set(Fam_Descriptor *descriptor, uint64_t offset, double value);
```

Description

These methods atomically set a value in FAM.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be set at the given location.

Return Values

None.

Notes

These methods atomically set a value in FAM. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_add

Atomically add a value to a data item in FAM.

Synopsis

```
void fam_add(Fam_Descriptor *descriptor, uint64_t offset, int32_t value);
void fam_add(Fam_Descriptor *descriptor, uint64_t offset, int64_t value);
void fam_add(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
void fam_add(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
void fam_add(Fam_Descriptor *descriptor, uint64_t offset, float value);
void fam_add(Fam_Descriptor *descriptor, uint64_t offset, double value);
```

Description

These methods atomically add a value to an existing value in FAM.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be added to the existing value at the given location.

Return Values

None.

Notes

These methods atomically add a value to an existing value in FAM. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_subtract

Atomically subtract a value from some data item in FAM.

Synopsis

```
void fam_subtract(Fam_Descriptor *descriptor, uint64_t offset, int32_t value);
void fam_subtract(Fam_Descriptor *descriptor, uint64_t offset, int64_t value);
void fam_subtract(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
void fam_subtract(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
void fam_subtract(Fam_Descriptor *descriptor, uint64_t offset, float value);
void fam_subtract(Fam_Descriptor *descriptor, uint64_t offset, double value);
```

Description

These methods atomically subtract a value from an existing value in FAM.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be subtracted from the existing value at the given location.

Return Values

None.

Notes

These methods atomically subtract a value from an existing value in FAM. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_min

Atomically replace a value in FAM with the minimum of it and some given value.

Synopsis

```
void fam_min(Fam_Descriptor *descriptor, uint64_t offset, int32_t value);
void fam_min(Fam_Descriptor *descriptor, uint64_t offset, int64_t value);
void fam_min(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
void fam_min(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
void fam_min(Fam_Descriptor *descriptor, uint64_t offset, float value);
void fam_min(Fam_Descriptor *descriptor, uint64_t offset, double value);
```

Description

These methods atomically replace a value in FAM with the minimum of the value given and the existing value.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be compared to the existing value at the given location.

Return Values

None.

Notes

These methods atomically replace a value in FAM with the smaller of the existing value and the value given in the method. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_max

Atomically replace a value in FAM with the maximum of it and some given value.

Synopsis

```
void fam_max(Fam_Descriptor *descriptor, uint64_t offset, int32_t value);
void fam_max(Fam_Descriptor *descriptor, uint64_t offset, int64_t value);
void fam_max(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
void fam_max(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
void fam_max(Fam_Descriptor *descriptor, uint64_t offset, float value);
void fam_max(Fam_Descriptor *descriptor, uint64_t offset, double value);
```

Description

These methods atomically replace a value in FAM with the maximum of the value given and the existing value.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be compared to the existing value at the given location.

Return Values

None.

Notes

These methods atomically replace a value in FAM with the larger of the existing value and the value given in the method. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_and

Atomically replace a value in FAM with the logical AND of that value and some given value.

Synopsis

```
void fam_and(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
void fam_and(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
```

Description

These methods atomically replace a value in FAM with the logical AND of the value given and the existing value.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be combined with the existing value at the given location.

Return Values

None.

Notes

These methods atomically replace a value in FAM with the logical AND of the existing value and the value given in the method. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_or

Atomically replace a value in FAM with the logical OR of that value and some given value.

Synopsis

```
void fam_or(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
void fam_or(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
```

Description

These methods atomically replace a value in FAM with the logical OR of the value given and the existing value.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be combined with the existing value at the given location.

Return Values

None.

Notes

These methods atomically replace a value in FAM with the logical OR of the existing value and the value given in the method. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_xor

Atomically replace a value in FAM with the logical XOR of that value and some given value.

Synopsis

```
void fam_xor(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
void fam_xor(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
```

Description

These methods atomically replace a value in FAM with the logical XOR of the value given and the existing value.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be combined with the existing value at the given location .

Return Values

None.

Notes

These methods atomically replace a value in FAM with the logical XOR of the existing value and the value given in the method. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_fetch_TYPE

Atomically fetches a value from FAM.

Synopsis

```
int32_t fam_fetch_int32(Fam_Descriptor *descriptor, uint64_t offset);
int64_t fam_fetch_int64(Fam_Descriptor *descriptor, uint64_t offset);
uint32_t fam_fetch_uint32(Fam_Descriptor *descriptor, uint64_t offset);
uint64_t fam_fetch_uint64(Fam_Descriptor *descriptor, uint64_t offset);
float fam_fetch_float(Fam_Descriptor *descriptor, uint64_t offset);
double fam_fetch_double(Fam_Descriptor *descriptor, uint64_t offset);
```

Description

These methods a value from FAM.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.

Return Values

The value from FAM.

Notes

These methods atomically fetch a value from FAM. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_swap

Atomically replace a value in FAM with the given value and return the old value.

Synopsis

```
int32_t fam_swap(Fam_Descriptor *descriptor, uint64_t offset, int32_t value);
int64_t fam_swap(Fam_Descriptor *descriptor, uint64_t offset, int64_t value);
uint32_t fam_swap(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
uint64_t fam_swap(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
float fam_swap(Fam_Descriptor *descriptor, uint64_t offset, float value);
double fam_swap(Fam_Descriptor *descriptor, uint64_t offset, double value);
```

Description

These methods atomically replace a value in FAM with the value given and return the existing value.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be swapped with the existing value at the given location.

Return Values

The old value from FAM.

Notes

These methods atomically replace a value in FAM with the given value and return the original value. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_compare_swap

Atomically conditionally replace a value in FAM with the given value and return the old value.

Synopsis

```
int32_t fam_compare_swap(Fam_Descriptor *descriptor, uint64_t offset, int32_t
      oldValue, int32_t newValue);
int64_t fam_compare_swap(Fam_Descriptor *descriptor, uint64_t offset, int64_t
      oldValue, int64_t newValue);
uint32_t fam_compare_swap(Fam_Descriptor *descriptor, uint64_t offset,
      uint32_t oldValue, uint32_t newValue);
uint64_t fam_compare_swap(Fam_Descriptor *descriptor, uint64_t offset,
      uint64_t oldValue, uint64_t newValue);
int128_t fam_compare_swap(Fam_Descriptor *descriptor, uint64_t offset, int128_t
      oldValue, int128_t newValue);
```

Description

These methods atomically perform a compare and swap of a value in FAM, and return the old value from FAM. All of these methods atomically implement the following algorithm (where **TYPE** is one of the data types listed in the synopsis):

```
TYPE cas(TYPE *p, TYPE oldValue, TYPE newValue){
      if(*p != oldValue){
            return *p;
      } else {
            *p = newValue;
            return oldValue;
      }
}
```

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
oldValue	Expected old value at the FAM location.
newValue	New value to use if successful.

Return Values

The old value from FAM.

Notes

These methods atomically compare a value in FAM with an expected value and if the value in FAM is equal to the expected value, swap a new value. The old value is returned. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations. GCC currently does not define int128_t; it is defined in fam.h.

Example

TBD

fam_fetch_add

Atomically adds a value to a value in FAM, and returns the old value.

Synopsis

```
int32_t fam_fetch_add(Fam_Descriptor *descriptor, uint64_t offset, int32_t value);
int64_t fam_fetch_add(Fam_Descriptor *descriptor, uint64_t offset, int64_t value);
uint32_t fam_fetch_add(Fam_Descriptor *descriptor, uint64_t offset,
      uint32_t value);
uint64_t fam_fetch_add(Fam_Descriptor *descriptor, uint64_t offset,
      uint64_t value);
float fam_fetch_add(Fam_Descriptor *descriptor, uint64_t offset, float value);
double fam_fetch_add(Fam_Descriptor *descriptor, uint64_t offset, double value);
```

Description

These methods atomically add a value to the value in FAM, and return the old value.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	The value to be added to the existing value in FAM.

Return Values

The original value from FAM.

Notes

These methods atomically add a value to the existing value in FAM. The old value is returned. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_fetch_subtract

Atomically subtracts a value from a value in FAM, and returns the old value.

Synopsis

```
int32_t fam_fetch_subtract(Fam_Descriptor *descriptor, uint64_t offset, int32_t
value);
int64_t fam_fetch_subtract(Fam_Descriptor *descriptor, uint64_t offset, int64_t
value);
uint32_t fam_fetch_subtract(Fam_Descriptor *descriptor, uint64_t offset,
uint32_t value);
uint64_t fam_fetch_subtract(Fam_Descriptor *descriptor, uint64_t offset,
uint64_t value);
float fam_fetch_subtract(Fam_Descriptor *descriptor, uint64_t offset,
float value);
double fam_fetch_subtract(Fam_Descriptor *descriptor, uint64_t offset,
double value);
```

Description

These methods atomically subtract a value from the value in FAM, and return the old value.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	The value to be subtracted from the existing value in FAM.

Return Values

The original value from FAM.

Notes

These methods atomically subtract a value from the existing value in FAM. The old value is returned. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_fetch_min

Atomically replaces a value in FAM with the smaller of the value in FAM and a given value, and returns the old value.

Synopsis

```
int32_t fam_fetch_min(Fam_Descriptor *descriptor, uint64_t offset, int32_t value);
int64_t fam_fetch_min(Fam_Descriptor *descriptor, uint64_t offset, int64_t value);
uint32_t fam_fetch_min(Fam_Descriptor *descriptor, uint64_t offset,
uint32_t value);
uint64_t fam_fetch_min(Fam_Descriptor *descriptor, uint64_t offset,
uint64_t value);
float fam_fetch_min(Fam_Descriptor *descriptor, uint64_t offset, float value);
double fam_fetch_min(Fam_Descriptor *descriptor, uint64_t offset, double value);
```

Description

These methods atomically replace a value in FAM with the smaller of the value in FAM and the given value, and return the old value.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	The value to be compared with the existing value in FAM.

Return Values

The original value from FAM.

Notes

These methods atomically replace a value in FAM with the minimum of that value and a given value. The old value is returned. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_fetch_max

Atomically replaces a value in FAM with the larger of the value in FAM and a given value, and returns the old value.

Synopsis

```
int32_t fam_fetch_max(Fam_Descriptor *descriptor, uint64_t offset, int32_t value);
int64_t fam_fetch_max(Fam_Descriptor *descriptor, uint64_t offset, int64_t value);
uint32_t fam_fetch_max(Fam_Descriptor *descriptor, uint64_t offset,
uint32_t value);
uint64_t fam_fetch_max(Fam_Descriptor *descriptor, uint64_t offset,
uint64_t value);
float fam_fetch_max(Fam_Descriptor *descriptor, uint64_t offset, float value);
double fam_fetch_max(Fam_Descriptor *descriptor, uint64_t offset, double value);
```

Description

These methods atomically replace a value in FAM with the larger of the value in FAM and the given value, and return the old value.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	The value to be compared with the existing value in FAM.

Return Values

The original value from FAM.

Notes

These methods atomically replace a value in FAM with the maximum of that value and a given value. The old value is returned. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_fetch_and

Atomically replaces a value in FAM with the logical AND of that value and some given value, and returns the old value.

Synopsis

```
uint32_t fam_fetch_and(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
uint64_t fam_fetch_and(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
```

Description

These methods atomically replace a value in FAM with the logical AND of the value given and the existing value. The old value is returned.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be combined with the existing value at the given location.

Return Values

The old value from FAM.

Notes

These methods atomically replace a value in FAM with the logical AND of that value and the value given in the method. The old value is returned. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_fetch_or

Atomically replaces a value in FAM with the logical AND of that value and some given value, and returns the old value.

Synopsis

```
uint32_t fam_fetch_or(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
uint64_t fam_fetch_or(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
```

Description

These methods atomically replaces a value in FAM with the logical OR of the value given and the existing value. The old value is returned.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be combined with the existing value at the given location.

Return Values

The old value from FAM.

Notes

These methods atomically replace a value in FAM with the logical OR of that value and the value given in the method. The old value is returned. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

fam_fetch_xor

Atomically replaces a value in FAM with the logical AND of that value and some given value, and returns the old value.

Synopsis

```
uint32_t fam_fetch_xor(Fam_Descriptor *descriptor, uint64_t offset, uint32_t value);
uint64_t fam_fetch_xor(Fam_Descriptor *descriptor, uint64_t offset, uint64_t value);
```

Description

These methods atomically replaces a value in FAM with the logical XOR of the value given and the existing value. The old value is returned.

Input Arguments

Name	Description
descriptor	Descriptor associated with the data item.
offset	Offset within the data item in FAM where value is located.
value	Value to be combined with the existing value at the given location.

Return Values

The old value from FAM.

Notes

These methods atomically replace a value in FAM with the logical XOR of that value and the value given in the method. The old value is returned. Note that the offset argument must point to the correct value for the data type. Availability of these methods is dependent on hardware support for the operations.

Example

TBD

Memory Ordering

These operations provide ordering operations to FAM. Given that all PEs can access all data in FAM directly, we believe that many of the collective operations defined within other PGAS APIs are not necessary, and the initial version of the API only provides blocking and non-blocking ordering operations. Additional methods may be made available as we gain experience using the API.

`fam_fence()` is a non-blocking call that enables ordering of FAM operations from a PE: any FAM operations (put, scatter, atomics, copy) issued by the calling thread before the fence operation will be completed before any FAM operations by the calling thread after the fence are dispatched. `fam_quiet()` is a blocking version of `fam_fence()` that blocks the calling thread until pending FAM operations are complete.

The current API provides ordering of FAM operations from the calling PE thread to all of fabric-attached memory. If hardware support is provided, future versions of the API may also include methods to provide ordering of outstanding FAM operations on a per-region basis (i.e., operations to a particular region would be ordered independently of FAM operations to other regions).

Note that unlike other PGAS implementations, given that all PEs have equal access to all data items in FAM, the semantics of collectives, reductions, and all-to-all messaging are unclear when addressing FAM. Although messaging (both one-sided and two-sided) can be supported over the fabric, as can coordination primitives such as barriers, they are omitted in this API at this time. If necessary, such operations can be added in a later version.

fam_fence

Ensures that FAM operations issued by the calling PE thread before the fence are completed before FAM operations issued after the fence are dispatched.

Synopsis

```
void fam_fence(void);
```

Description

This method orders FAM operations (put, scatter, atomics, copy) issued by the calling PE thread before the fence to be performed before FAM operations issued by the calling PE thread after the fence. This method is non-blocking.

Input Arguments

None.

Return Values

None.

Notes

Note that this method does NOT order `fam_map()`-enabled load/store accesses by the processor to FAM.

Example

```
#include<stdio.h>
#include "fam.h"

int main(void){
	// ... Initialization code here

	// look up the descriptor to a previously allocated data item
	// (a 50-element integer array)
	Fam_Descriptor *descriptor = fam_lookup("myItem", "myRegion");
	// local data
	int local[] = {0,1,2,3,4};
	uint64_t indexes[] = {0, 7, 3, 5, 6};
	// scatter local data in myItem[0,7,3,5,6] respectively
	fam_scatter_nonblocking(local, descriptor, 5, indexes, sizeof(int));

	// ensure that the scatter is performed before subsequent FAM updates
	fam_fence();

	// now update later elements of the array in FAM
	int newLocal[] = {0,1,2,3,4,5,6,7,8,9};
	// update elements 11 - 20 in FAM using values in local memory
	fam_put_nonblocking(newLocal, descriptor, 10*sizeof(int), sizeof(local));
}
```

fam_quiet

Ensures that all pending operations to FAM issued by the calling PE thread have completed before proceeding

Synopsis

```
void fam_quiet(void);
```

Description

This method blocks the current PE thread until all its pending FAM operations are complete.

Input Arguments

None.

Return Values

None.

Notes

Note that this method does NOT order or wait for completion of `fam_map()`-enabled load/store accesses by the processor to FAM.

Example

```
#include<stdio.h>
#include "fam.h"

int main(void){
	// ... Initialization code here

	// look up the descriptor to a previously allocated data item
	// (a 50-element integer array)
	Fam_Descriptor *descriptor = fam_lookup("myItem", "myRegion");
	// local data
	int local[] = {0,1,2,3,4};
	uint64_t indexes[] = {10, 17, 13, 15, 16};
	// scatter local data in myItem[10,17,13,15,16] respectively
	fam_scatter(local, descriptor, 5, indexes, sizeof(int));

	// ensure that the scatter operation is complete
	fam_quiet();

	// look up the descriptor to a previously allocated completion flag
	Fam_Descriptor *flagDescriptor = fam_lookup("completionFlag", "myRegion");
	// atomically set the completion flag in FAM
	fam_set(flagDescriptor, 0, 1);

	// ensure that the set operation is complete
	fam_quiet();
	// we now have the correct items in FAM, and the completion flag has been set
}
```

Failure semantics

OpenFAM calls may be unsuccessful due to failures in fabric components, fabric-attached memory controllers or the fabric-attached memory media itself. We expect that lower-layer hardware and software will implement failure mitigation strategies (e.g., request retry, memory bad block remapping, inter-controller replication or erasure coding) that will mask (and in some cases, recover from) these failures. As a result, we interpret any failures reported to the OpenFAM layer to be unrecoverable.

The OpenFAM API provides two alternative approaches for dealing with failures: *failure-reporting* operation and *fail-fast* operation. With failure-reporting operation, failures are detected and associated with the call that triggered or experienced them. As a result, this mode can only be used with blocking OpenFAM calls. The type of error is reported to the OpenFAM layer, which can pass the error code to the application, allowing for application-specific error handling based on the severity of the error. For example, an application that implements its own application-level replication may choose to recover from a failed `fam_get_blocking()` request to an inoperable replica by retrying the operation with a different replica. A disadvantage of the failure-reporting approach is that performance is lower, due to the need to use only blocking calls for attributable error reporting.

FAM failures may affect the operation of a non-blocking call well after the call has returned, so it isn't always possible to attribute the failure to the call that first experienced it. Rather than support out-of-band error reporting (e.g., through exceptions), OpenFAM handles this situation by providing a fail-fast mode of operation. In this approach, which mimics OpenSHMEM's failure semantics, unrecoverable FAM errors cause the application (including all its PEs) to be terminated. Because fail-fast operation supports non-blocking calls, it provides the potential for higher performance.

Neither failure-reporting operation nor fail-fast operation provide guarantees of all-or-nothing execution of failed OpenFAM calls; partial completion is a possible outcome under failure scenarios. For fail-fast operation, any partially completed update calls to data persistently stored in FAM may result in corrupted application state that the application does not get the opportunity to recover before the termination of the current program invocation. (In failure-reporting operation, the application can choose whether and how to recover any potential corruptions at the time of the failed call.) As a result, applications using fail-fast operation (and potentially those using failure-reporting operation) should be structured to explicitly check at initialization whether persistent data needs to be recovered, and to perform the appropriate application-specific recovery. As an extreme approach, applications may even choose to treat FAM data as ephemeral, deleting any residual regions or data items and starting with a clean slate.

Discussion

This section covers additional details that may be relevant to reference implementations, as well as open issues for future consideration.

Implementation issues

Descriptors. Region and data item descriptors are intended to be data types that should be treated as opaque by the application, but which permit the OpenFAM library implementation to locate memory in FAM from any PE. For example, a data item descriptor may contain the offset of the data item within its region. This opacity assumption allows descriptors to be shared between PEs and programs in a workflow.

Due to the desire to easily share descriptors both across PEs and across more complex workflows, we want them to remain immutable once created. This desire has implications for the OpenFAM library implementation. For example, it isn't advisable to implement region resizing operations in such a way that relocates the region and changes the descriptor. Resizing regions will be most efficiently implemented through chaining, if it is not possible to extend the existing region in a contiguous fashion. By chaining, we mean using noncontiguous physical regions of FAM, which are logically treated as a single virtual region. Such chaining can be implemented by augmenting a virtual region's metadata to indicate the set of physical regions.[3] Since shrinking regions will cause errors to subsequent accesses to data items in the truncated sub-region, in future versions we may consider adding the restriction that resize operations only enlarge the region.

Groups. We define a group abstraction to permit specification of sets of PEs that are cooperating to accomplish a shared goal.[4] Although the group abstraction isn't used directly in the remaining OpenFAM calls described in this document beyond initialization/finalization, it enables coordination between PEs in other parts of the application runtime (e.g., to implement synchronization/barrier operations, detect PE failure). Toward that end, `fam_initialize()` may implement logic to incorporate the calling PE into whatever state is needed to track group membership, detect PE failure, etc. In the future, we envision that a PE could be a member of multiple groups, and so may call initialize/finalize multiple times, to delineate activity for a given group.

Future considerations

Options and configuration parameters. The options provided as an argument to `fam_initialize()` are intended to be a flexible way to capture options describing the internal library state, including state that might otherwise be captured as environment variables. Specifying options through environment variables can lead to non-deterministic execution, so we prefer to explicitly set such parameters as library options using this mechanism. The exact set of options is extensible, and hence can be expanded in the future to support necessary configuration parameters.

Scratch regions. Applications may wish to create regions that are automatically garbage collected at the end of the application's execution (either through successful completion or through fail-fast termination in the event of a FAM error). Given the OpenFAM semantics outlined in this document, this abstraction could be provided by the application deallocating any scratch regions during its finalization phase (for successful completion) or during its initialization phase (to handle any residual scratch regions from fail-fast termination). Future versions of the API

[3] Chaining is similar to how physical file systems use a collection of variable-length extents to comprise a file, or how OpenStack Swift manages large objects (https://docs.openstack.org/swift/latest/overview_large_objects.html).

[4] OpenFAM groups are similar in some respects to the *team* abstraction proposed for OpenSHMEM.

will consider system-level support for scratch regions (e.g., through region allocation-time scratch specification and automatic garbage collection at the end of the program).

Data item resizing. Currently the OpenFAM API does not support resizing (or automated re-packing) of data items within a region. This constraint may be relaxed in the future as implementations become available.

Multi-item gather/scatter. The current `fam_gather()` and `fam_scatter()` operate on the elements of a single data item in FAM. In the future, we will consider whether it is useful to gather (scatter) from (to) multiple FAM-resident data items.

Data item snapshots. The current `fam_copy()` performs an inconsistent copy, in that the copied data may reflect updates that are performed concurrently with the copy operation. To ensure a consistent copy without the need to quiesce application accesses to the copied region, future versions of the API will consider including functions to snapshot the source data item.

Per-region memory ordering. The current API provides ordering of FAM operations from the calling PE thread to all of fabric-attached memory. If hardware support is provided, future versions of the API may also include methods to provide ordering of outstanding FAM operations on a per-region basis (i.e., operations to a particular region would be ordered independently of FAM operations to other regions).

Data management services. In the future we may explore system services to move regions within FAM to recover from FAM failures, or alleviate contention for specific FAM resources. Such services should not disturb the layout of data items within a region during such moves: all data items within a given region should be moved *in toto* as a single unit. This will ensure that data item offsets within the region are preserved, meaning that data item descriptors need not change if the region is relocated.

Metadata service. The current version of the API primarily uses the metadata service to map user-friendly names to FAM descriptors. In the future, as we gain implementation experience, we will consider extending the API to include more options for querying (and if appropriate, changing) metadata attributes.

Based on past experience from adjacent fields like file systems, providing a metadata service that scales to a large number of items and a large number of users is challenging. Some of these challenges come from the semantics of the POSIX file API (e.g., strictly consistent reporting of metadata attributes; maintenance of attributes like last modification time, which change frequently upon data path accesses). By re-envisioning persistent data management in the context of FAM regions and data items accessed through the OpenFAM API, rather than files accessed via the POSIX file API, we believe that an OpenFAM metadata service can overcome the challenges of traditional file metadata services. One potential advantage for a FAM environment is that the shared state represented by region and data item metadata can be implemented as data structures stored in FAM, and directly accessible by any node. As a result, trusted code running on any (or even all) nodes can make use of one-sided accesses and atomics to access and update metadata in a scalable fashion.