**Design Document**

SanDisk FDF

Version 1.2

Sandisk Confidential

sd-logo

1 Introduction 4

2 Requirements 4

2.1 Large number of containers 4

2.2 Dynamic containers 4

2.3 Parallel enumeration 4

2.4 Mini Transaction 4

2.5 Slab garbage collection 4

2.6 Orderly shutdown 4

2.7 Expiry time 4

2.8 Usability improvements 4

3 High level design 4

3.1 Large number of containers 4

3.1.1 User APIs None 5

3.1.2 Internal APIs 5

3.1.3 Configuration 5

3.1.4 Known Limitations/Issues 5

3.1.5 Stats and Logs 5

3.1.6 Test cases 5

3.2 Dynamic containers 5

3.2.1 Design 5

3.2.2 User API 8

3.2.3 Internal API 10

3.2.4 Configuration 10

3.2.5 Known Limitations/Issues 11

3.2.6 Stats 11

3.2.7 Test Cases 11

3.3 Object Management 11

3.3.1 User API 12

3.3.2 Internal API 15

3.3.3 Configuration 15

3.3.4 Known Limitations/Issues 15

3.3.5 Stats 15

3.3.6 Test Cases 15

3.4 Parallel enumeration 18

3.4.1 Design 18

3.4.2 User APIs 18

3.4.3 Internal APIs 18

3.4.4 Configuration 19

3.4.5 Issues and Limitations 19

3.4.6 Statistics 19

3.4.7 Testcases 19

3.5 Mini Transaction 22

3.5.1 Design 22

3.5.2 User APIs 24

3.5.3 Internal APIs 24

3.5.4 Configuration 24

3.5.5 Issues and Limitations 24

3.5.6 Statistics and Logs 24

3.5.7 Test cases 25

3.6 Slab garbage collection 28

3.6.1 User APIs 29

3.6.2 Internal APIs 29

3.6.3 Configuration 29

3.6.4 Issues and Limitations 29

3.6.5 Statistics 29

3.6.6 Testcases 29

3.7 Usability improvements 32

3.7.1 Design 32

3.8 Expiry Time 32

3.8.1 Design 32

3.8.2 User APIs 32

3.8.3 Internal APIs 33

3.8.4 Configuration 33

3.8.5 Issues and Limitations 33

3.8.6 Stats 33

3.8.7 Testcases 33

3.9 Orderly shutdown 34

3.9.1 Design 34

3.9.2 User APIs 34

3.9.3 Internal APIs 34

3.9.4 Configuration 34

3.9.5 Issues and Limitations 34

3.9.6 Stats 34

3.9.7 Testcases 34

# Introduction

This document describes the features list for the FDF 1.1 release and high-level design.

# Requirements

## Large number of containers

The FDF1.1 release shall support up 65535 containers. Note: a few containers are reserved for FDF use. See Section 3.1.

## Dynamic containers

Applications shall be able to increase the size of the containers dynamically.

## Parallel enumeration

Application threads shall be able to enumerate containers in parallel.

## Mini Transaction

The FDF 1.1 release shall support simple transactions. Applications shall be able group one or more key value operations as a transaction. Applications need to deal with any failed operations in the transaction.

## Slab garbage collection

In the current release, flash space occupied by objects in slab mode containers are not reused after the objects are deleted. The FDF 1.1 release shall reuse freed flash space for storing new objects.

## Orderly shutdown

The FDF 1.1 shall bring down various services in FDF gracefully during shutdown.

## Expiry time

The FDF 1.1 shall support expiry times for objects. The FDF shall make the object invalid after specified expiry time is elapsed.

## Usability improvements

The FDF 1.1 shall report appropriate error messages in the logs when an API fails.

# High level design

## Large number of containers

The proposed method for supporting large number of containers is to implement "virtual containers" that have no physical storage on top of one physical container. Virtual containers have all of the metadata of physical containers but no pre-allocated storage. Each virtual container uses the storage of its associated physical container.

### User APIs None

### Internal APIs

None

### Configuration

None

### Known Limitations/Issues

* The global define MCD\_MAX\_NUM\_CNTRS limits the number of containers that can be created (not including the CMC). This limit is currently set to 128, mainly to simplify the UI. The target for MCD\_MAX\_NUM\_CNTRS is 65K. It appears that the current methods use global tables as well as per thread tables that are sized by this define. We may need to implement a different method of caching metadata in place of these static tables.
* The global define SDF\_MAX\_CONTAINERS in the protocol/action layers will also have to be increased.
* The size of the containers cannot be decreased.
* The null container id (FDF\_NULL\_CGUID) is defined as 0.
* Three containers are reserved for FDF use:
  + CMC – cguid 1. Physical container metadata.
  + VMC – cguid 2. Virtual container metadata.
  + VDC – cguid 3. Virtual container data objects.
* 65532 container ids are available for virtual containers.

### Stats and Logs

### Test cases

Refer section 3.3.6

## Dynamic containers

The design takes the following requirements in to account

* Physical and virtual containers must be restricted to the configured size.
* Size is specified in KB.
* Minimum container size is 1KB.
* Maximum container size is FDF\_FLASH\_SIZE - 3GB - 32MB (overhead for CMC, VMC & VDC).
* Max container size must not be exceeded on FDFWriteObject.
* Current container size must be maintained over all FDFWriteObject/FDFDeleteObject.

### Design

**Container Creation:**

In the current FDF API, containers are managed using the container API itself. That is, one container, the CMC, is used to maintain metadata objects for all other containers. The only difference between the CMC and other containers is that the CMC metadata is maintained in a global structure and not as a container object.

The proposed method for supporting dynamically sized containers is to implement "virtual containers" that have no preallocated physical storage. Virtual containers have all of the metadata of physical containers but no preallocated real storage. Each virtual container uses the storage of its associated physical container. In the proposed implementation, there will be three physical containers:

1. CMC - maintains physical container metadata
2. VMC – maintains virtual container metadata
3. VDC – maintains virtual data objects

The current container API and recovery mechanisms will be modified in the following way:

1. Creation of the CMC will use the existing method:
   1. Store metadata in the global CMC structure
   2. Build shard and blob persistent structures
2. The VMC and VDC will be created during FDF initialization
   1. The VMC will be sized to contain the maximum number of containers
   2. The VDC will be sized to allocated the remaining amount of storage
3. Creation of all virtual containers will use a modified method
   1. Store metadata as VMC objects
   2. Do not build shard and blob structures (no need since VDC owns all physical storage)
   3. In the metadata for virtual containers, the shard id of the VDC will be inserted. This allows access to virtual objects to map to the proper shard.
   4. A container list object will be maintained in the VMC for recovery. Alternatively, we could just enumerate the per-container metadata objects in the VMC for recovery.

Using the above method, containers can be located in the current way (through CMC or VMC metadata). The container recovery path will make use of the VMC container list object to properly fill out global data structures used to manage containers.

In order to manage sizing and resizing of containers, the following will be added to the container metadata:

* Object count
* Current size

The current size will be used to enforce container limits. The Configured max container size is maintained in Container metadata object and Flash storage structures. The current size of the container is maintained at Flash storage structures and the container size limit is enforced at the storage layer. Alternatively, these statistics could be maintained by the in-memory object hashtable module. This may make more sense because this layer will be responsible for scanning the hashtable at startup to get the initial values of object count and container size.

The FDF cache module will need to change the code that stores and looks up cached container metadata.  The current caching code maintains an array of size "SDF\_MAX\_CONTAINERS" and does a linear search to find a particular container or a free entry.  It needs to be replaced with a hash table for efficiently handling 10's of  thousands of containers.  The code changes are straightforward and isolated to action\_new.c, simple\_replication.c and recovery.c. The #define SDF\_MAX\_CONTAINERS will have to be set to an appropriate new limit.

**Container ID Allocation**

The NULL cguid is 0. It is used to define an invalid cguid.

Three physical containers are used to support FDF physical and virtual container operations:

* CMC
  + Physical container metadata.
  + Cguid 0
* VMC
  + Virtual container metadata.
  + Cguid 1
* VDC
  + Virtual container data objects.
  + Cguid 2

All cguid greater than the three physical cguid are assumed to be virtual cguid.

Container APIs are divided between physical and virtual container operations (open, close, delete, etc.).

**Container Open/Close**

The container open/close paths in the FDF API and object protocol API will be modified such that shards will not be accessed for virtual containers. Otherwise, these paths will execute as for physical containers.

In order to maintain compatibility with the FDF OSD and AIO layers, each virtual container will be assigned the VDC shard id. This allows a virtual container’s objects to be located in its home shard.

**Container Size**

At any given time, the current size of a container (keys + data) is limited to the configured size of the container. Once the limit is reached, further attempts to add data will be rejected.

The container may be resized.

The sum of all virtual container sizes will be limited to the total amount of storage less the size of the physical containers and other storage overhead:

Available virtual container storage = FDF\_FLASH\_SIZE – 3GB – 32MB.

**Container Resize**

The existing API FDFSetContainerProps() will be used for resizing the containers. Currently, a container may only be increased in size.

**Container Delete**

The container delete API will be required to enumerate all container objects and delete them before the metadata is released.

In order to make container delete crash safe, the container metadata will be marked with the "delete\_in\_progress" attribute set. During recovery, if the delete\_in\_progress flag is set, a delete container will be performed by the recovery function before FDF initialization completes.

**Flush Container**

The current flush container/flush cache/flush object implementations should work as-is.

**Enumeration**

The container API will call the enumeration interface to execute the FDF enumerate APIs. See Section 3.4.

**Compatibility with existing container API**

The existing physical container API will be maintained (though renamed) so that other container models may be supported. The virtual container API will use the current API definitions.

In this way, we could theoretically support multiple physical and virtual containers within a single FDF instance or perhaps in separate instances that use a configuration switch to determine the types of containers to allow.

Also, by supporting multiple physical containers, we can supports models in which virtual containers can be supported by multiple physical containers rather than just with the CMC.

The existing FDF API calls will support virtual container operations.

A new set of FDF physical API calls will support the existing container/object semantics but will be named:

* FDFOpenPhysicalContainer
* FDFClosePhysicalContainer
* FDFDeletePhysicalContainer
* etc.

**Container identifiers**

The existing container identifier, the cguid, is currently generated from a monotonically increasing counter, the container id. In order to simplify enumeration, FDF 1.1 will use a cguid counter that can reuse returned values. The FDF\_cguid\_t remains a uint64\_t.

**Per-Cache and Per-Container Stats**

The existing container stats mechanism should work as-is. Most of the container stats are collected in action\_new.c and fastcc\_new.c.  A large number of event counts are maintained in the global FDF state structure in the SDF\_action\_stats\_new\_t structure, which contains an array of "SDF\_MAX\_CONTAINERS" SDF\_cache\_ctnr\_stats\_t structures.  10's of thousands of containers, however, will require about 500MB of memory to hold these arrays.  It is assumed that this memory increase is tolerable.

Given the rules for passing container cguid's to the protocol layer, all stats will be recorded on the basis of:

* virtual cguid's for virtual containers physical cguid's for physical containers that hold virtual containers
* No statistics will be collected for the physical cguid of a physical container that holds virtual containers.

### User API

/\*\*

\* @brief Create and open a container.

\*

\* @param fdf\_thread\_state <IN> The FDF context for which this operation applies

\* @param cname <IN> container name

\* @param properties <IN> container properties

\* @param flags <IN> container open options

\* @param cguid <OUT> container GUID

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFOpenContainer(

struct FDF\_thread\_state \*fdf\_thread\_state,

char \*cname,

FDF\_container\_props\_t \*properties,

uint32\_t flags,

FDF\_cguid\_t \*cguid

);

/\*\*

\* @brief Close a container.

\*

\* @param fdf\_thread\_state <IN> The FDF context for which this operation applies

\* @param cguid <IN> container CGUID

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFCloseContainer(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t cguid

);

/\*\*

\* @brief Delete a container

\*

\* @param fdf\_thread\_state <IN> The FDF context for which this operation applies

\* @param cguid <IN> container CGUID

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFDeleteContainer(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t cguid

);

/\*\*

\* @brief Resize a container

\*

\* @param fdf\_thread\_state <IN> The FDF context for which this operation applies

\* @param cguid <IN> container CGUID

\* @param size <IN> container size in bytes (must be > than current size)

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFResizeContainer(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t cguid,

uint64\_t size

);

/\*\*

\* @brief Get container list

\*

\* @param fdf\_thread\_state <IN> The FDF context for which this operation applies.

\* @param cguids <OUT> pointer to container GUID array

\* @param n\_cguids <OUT> pointer to container GUID count

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFGetContainers(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t \*cguids,

uint32\_t \*n\_cguids

);

/\*\*

\* @brief Get container properties

\*

\* @param fdf\_thread\_state <IN> The FDF context for which this operation applies

\* @param cguid <IN> container global identifier

\* @param pprops <IN> pointer to structure into which to copy properties

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFGetContainerProps(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t cguid,

FDF\_container\_props\_t \*pprops

);

/\*\*

\* @brief Flush container

\*

\* @param fdf\_thread\_state <IN> The FDF context for which this operation applies.

\* @param cguid <IN> container global identifier

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFFlushContainer(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t cguid

);

/\*\*

\* @brief Flush the cache

\*

\* @param fdf\_thread\_state <IN> The SDF context for which this operation applies.

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFFlushCache(

struct FDF\_thread\_state \*fdf\_thread\_state

);

/\*\*

\* @brief Get FDF statistics

\*

\* @param fdf\_thread\_state <IN> The SDF context for which this operation applies

\* @param stats <OUT> pointer to statistics return structure

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFGetStats(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_stats\_t \*stats

);

/\*\*

\* @brief Get per container statistics

\*

\* @param fdf\_thread\_state <IN> The SDF context for which this operation applies

\* @param cguid <IN> container global identifier

\* @param stats <OUT> pointer to statistics return structure

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFGetContainerStats(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t cguid,

FDF\_stats\_t \*stats

);

### Internal API

TBD

### Configuration

Container configuration is specified in the FDF\_container\_props\_t variable that is passed into the open container API.

typedef struct {

uint64\_t size;

FDF\_boolean\_t persistent;

FDF\_boolean\_t evicting;

FDF\_boolean\_t writethru;

FDF\_durability\_level\_t durability\_level;

} FDF\_container\_props\_t;

Size is in bytes.

Persistence is currently ignored (it is always set true).

Evicting is currently ignored (it is always set false).

Writethru sets the mode of the cache (true = writethru, false = writeback).

Durability level may be set to partial or full.

Warnings will be issued if a user tries to set persistence to false or evicting to true.

### Known Limitations/Issues

Decreasing the size of a virtual container will not be supported in this release.

### Stats

### Test Cases

The existing FDF API test cases are valid for FDF 1.1 with the following exceptions:

* FIFO mode is not supported
* Eviction mode is not supported
* Non-persistent mode is not supported
* Container size is specified in bytes
* The minimum size for a container is 1 byte.
* The maximum size of a container is 4GB.
* Physical API tests will be required depending on when we intend on using them

See Section 3.3.6 for additional tests.

## Object Management

All objects will be stored in the VDC. In order to associate an object with its virtual container, the object key will be prefixed with the container id. This affects:

* FDFReadObject
* FDFWriteObject
* FDFDeleteObject
* FDFFlushObject
* FDFEnumerateContainerObjects
* FDFNextEnumeratedObject
* FDFFinishEnumeration

**Compatibility with existing object API**

The existing physical object API will be maintained (though renamed) so that other container models may be supported. The virtual object API will use the current API definitions.

In this way, we could theoretically support multiple physical and virtual containers within a single FDF instance or perhaps in separate instances that use a configuration switch to determine the types of containers to allow.

Also, by supporting multiple physical containers, we can supports models in which virtual containers can be supported by multiple physical containers rather than just with the CMC.

The existing FDF API calls will support virtual container operations.

A new set of FDF physical API calls will support the existing container/object semantics but will be named:

* FDFReadPhysicalObject
* FDFWritePhysicalObject
* etc.

### User API

/\*\*

\* @brief Get a copy of an object for read-only access. Return its current expiry time.

\*

\* Get an object and copy it into an FDF-allocated buffer. The application

\* only intends to read the object. The current expiry time is returned.

\*

\* @param fdf\_thread\_state <IN> The FDF context for which this operation applies.

\* @param cguid <IN> Identity of an open container with appropriate permissions.

\* @param key <IN> Key of object to be operated on.

\* @param keylen <IN> Length of key in bytes.

\* @param data <IN> Variable in which to return a pointer to the object (in a buffer

\* allocated by SDF; it must be freed by the application with a call

\* to SDFFreeObjectBuffer).

\* @param datalen <OUT> Pointer to the variable in which to return the actual size of

\* an object.

\*

\* @return FDF\_SUCCESS: operation completed successfully.

\* FDF\_BAD\_CONTEXT: the provided context is invalid.

\* FDF\_CONTAINER\_UNKNOWN: the container ID is invalid.

\* FDF\_OBJECT\_UNKNOWN: the object does not exist.

\* FDF\_IN\_TRANS: this operation cannot be done inside a transaction.

\* FDF\_FAILURE: operation failed.

\*/

FDF\_status\_t FDFReadObject(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t cguid,

char \*key,

uint32\_t keylen,

char \*\*data,

uint64\_t \*datalen

);

/\*\*

\* @brief Free an object buffer

\*

\* @param buf <IN> object buffer

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFFreeBuffer(

char \*buf

);

/\*\*

\* @brief Write entire object, creating it if necessary.

\*

\* Put an entire object, with contents copied from an application-provided

\* buffer. This may change the size of the object. The expiry time

\* is set. If the object does not exist, create it and assign its

\* value.

\*

\* @param fdf\_state <IN> The SDF context for which this operation applies.

\* @param cguid <IN> Identity of an open container with appropriate permissions.

\* @param key <IN> Key of object to be operated on.

\* @param keylen <IN> Length of key in bytes.

\* @param datalen <IN> Size of object.

\* @param data <IN> Pointer to application buffer from which to copy data.

\* @param flags <IN> create/update flags

\*

\* @return FDF\_SUCCESS: operation completed successfully.

\* FDF\_BAD\_CONTEXT: the provided context is invalid.

\* FDF\_CONTAINER\_UNKNOWN: the container ID is invalid.

\* FDF\_OUT\_OF\_MEM: there is insufficient memory/flash.

\* FDF\_IN\_TRANS: this operation cannot be done inside a transaction.

\* FDF\_FAILURE: operation failed.

\*/

FDF\_status\_t FDFWriteObject(

struct FDF\_thread\_state \*sdf\_thread\_state,

FDF\_cguid\_t cguid,

char \*key,

uint32\_t keylen,

char \*data,

uint64\_t datalen,

uint32\_t flags

);

/\*\*

\* @brief Delete an object, but check for expiry first.

\*

\* Delete an object, but check for expiry first.

\*

\* @param fdf\_thread\_state <IN> The SDF context for which this operation applies.

\* @param cguid <IN> Identity of an open container with appropriate permissions.

\* @param key <IN> Key of object to be operated on.

\* @param keylen <IN> Length of key in bytes.

\*

\* @return FDF\_SUCCESS: operation completed successfully.

\* FDF\_BAD\_CONTEXT: the provided context is invalid.

\* FDF\_CONTAINER\_UNKNOWN: the container ID is invalid.

\* FDF\_OBJECT\_UNKNOWN: the object does not exist.

\* FDF\_FAILURE: operation failed.

\*/

FDF\_status\_t FDFDeleteObject(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t cguid,

char \*key,

uint32\_t keylen

);

/\*\*

\* @brief Enumerate container objects

\*

\* @param fdf\_thread\_state <IN> The SDF context for which this operation applies

\* @param cguid <IN> container global identifier

\* @param iterator <IN> enumeration iterator

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFEnumerateContainerObjects(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t cguid,

struct FDF\_iterator \*\*iterator

);

/\*\*

\* @brief Container object enumration iterator

\*

\* @param fdf\_thread\_state <IN> The SDF context for which this operation applies

\* @param iterator <IN> enumeration iterator

\* @param cguid <IN> container global identifier

\* @param key <OUT> pointer to key variable

\* @param keylen <OUT> pointer to key length variable

\* @param data <OUT> pointer to data variable

\* @param datalen <OUT> pointer to data length variable

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFNextEnumeratedObject(

struct FDF\_thread\_state \*fdf\_thread\_state,

struct FDF\_iterator \*iterator,

char \*\*key,

uint32\_t \*keylen,

char \*\*data,

uint64\_t \*datalen

);

/\*\*

\* @brief Terminate enumeration

\*

\* @param fdf\_thread\_state <IN> The SDF context for which this operation applies

\* @param iterator <IN> enumeration iterator

\* @return FDF\_SUCCESS on success

\*/

FDF\_status\_t FDFFinishEnumeration(

struct FDF\_thread\_state \*fdf\_thread\_state,

struct FDF\_iterator \*iterator

);

/\*\*

\* @brief Force modifications of an object to primary storage.

\*

\* Flush any modified contents of an object to its backing store

\* (as determined by its container type). For coherent containers,

\* this is a global operation that applies to any cache or buffer

\* in the SDF cluster. For non-coherent containers, this only applies

\* to the local cache.

\*

\* @param fdf\_thread\_state <IN> The SDF context for which this operation applies.

\* @param cguid <IN> Identity of an open container with appropriate permissions.

\* @param key <IN> Key of object to be operated on.

\* @param keylen <IN> Length of key in bytes.

\*

\* @return FDF\_SUCCESS: operation completed successfully.

\* FDF\_BAD\_CONTEXT: the provided context is invalid.

\* FDF\_CONTAINER\_UNKNOWN: the container ID is invalid.

\* FDF\_OBJECT\_UNKNOWN: the object does not exist.

\* FDF\_IN\_TRANS: this operation cannot be done inside a transaction.

\* FDF\_FAILURE: operation failed.

\*/

FDF\_status\_t FDFFlushObject(

struct FDF\_thread\_state \*fdf\_thread\_state,

FDF\_cguid\_t cguid,

char \*key,

uint32\_t keylen

);

### Internal API

None

### Configuration

None

### Known Limitations/Issues

None

### Stats

### Test Cases

|  |  |  |
| --- | --- | --- |
| # | Test case | Description |
| 1 | Create 65K containers and check all can be recovered | 1. Configure FDF with 16G storage 2. Create 65K equal sized containers with FULL durability 3. Fill all the containers with objects 4. Restart the FDF   Expected Result:   1. All the containers must exist 2. The data of all the containers must be consistent |
| 2. | Enumerate container list internally | 1. Configure FDF with 16G storage 2. Create 65K equal sized containers with FULL durability 3. Fill all the containers with some objects 4. Enumerate the list of containers   Expected Result:   1. The enumeration should list all the containers |
| 3. | Check if FDF can recover from a crash while base physical containers are created | 1. Configure FDF with 16G storage 2. Start the FDF 3. Inject fault or kill while CMS is being created 4. Restart the FDF   Expected Result:  The FDF should start successfully |
| 5 | Check if all objects of a container are cleaned from CMC after the container is deleted | 1. Configure FDF with 16G storage 2. Create some number of containers 3. Start the workload to the containers 4. Delete one or more containers   Expected Result:   1. All objects of the deleted containers must be removed from CMC containers.(We can enumerate the base container to verify this) |
| 6 | Check if simultaneous deletes of a container are handler correctly | 1. Configure FDF with 16G storage 2. Create some number of containers 3. Start the workload and fill the container with some amount of data 4. Delete a container from multiple application threads at same time   Expected Result:  1. one of the application thread should succeed deleting containers and other threads should get error code |
| 7 | Check if container size can be increased | 1. Configure FDF with 16G storage 2. Create a container with size 1MB 3. Start the workload and fill the container with 1MB data. 4. Increase the size of the container to 2MB 5. Continue the workload   Expected Result:   1. Application shall be able to load up to 2MB amount of data |
| 8 | Check if container ID of a deleted container can be reused | 1. Configure FDF with 16G storage 2. Create 10 containers 3. Delete all containers 4. Create 10 containers again   Expected Result:   1. The new containers should get the Ids of the deleted containers |
| 9 | Check if container creation fails after the storage is fully utilized | 1. Configure FDF with 16G storage 2. Create 16 containers with size 1G each 3. Create 17th container   Expected Result:   1. The container creation should fail |
| 10 | Check if there is any memory leak after the containers are deleted | 1. Configure FDF with 16G storage 2. Create 65K equal sized containers with FULL durability 3. Fill all the containers with objects 4. Delete all containers   Expected Result:  1. No memory leak |
| 11 | Check if periodical durability works | 1. Configure FDF with 16G storage   1. Create a container with periodic durability 2. Start the workload 3. Kill FDF 4. Restart FDF   Expected Result:   1. Up to 512 objects/updates can be lost. Remaining objects must exist |
| 12 | Check if the durability level Software Crash | 1. Configure FDF with 16G storage 2. Create a container with the durability level “Software Crash” 3. Start the workload 4. Kill FDF 5. Restart FDF   Expected Result:  No objects/updates must be lost. |
| 13 | Check if the durability level Hardware Crash | 1. Configure FDF with 16G storage 2. Create a container with periodic durability 3. Start the workload 4. Restart system   Expected Result:  No objects/updates must be lost |
| 14 | Check if multiple containers with different durability level works | 1. Configure FDF with 16G storage 2. Create 5 containers with periodic durability, 5 containers with “Software Crash” and 5 containers with “hardware crash” 3. Start the workload to all the containers 4. Restart system   Expected Result:   1. Up to 512 objects can be lost in the containers with the periodic durability and “software crash” durability 2. No objects can be lost in the containers with the “HW crash” durability |
| 15 | Check if persistent and non persistent stats variables work | 1. Configure FDF with 16G storage 2. Create some number of containers 3. Start the workload to all the containers   Expected Result:  The stats variables like current items, total items, deleted items and so on should reflect correct information |
| 16 | Check if persistent stats variables work after FDF restart | 1. Configure FDF with 16G storage 2. Create some number of containers 3. Start the workload to all the containers 4. Restart the FDF   Expected Result:  The persistent stats variables like current items, total items, slab use, … should reflect correct information |
| 17 | FDF 1.0 Tests |  |

## Parallel enumeration

FDF applications might need to scan all the objects in a container for various application use cases. The enumeration functionality in the current FDF version very limited, mainly it does not support parallel enumeration.

The FDF 1.1 shall support parallel enumeration of containers in order to support such use cases. One or more application threads shall be able to enumerate one or more

containers in parallel. In FDF, we have one base physical container and large set of logical containers. FDF architecture will be enhanced for enumerating logical containers. Applications shall be able to end enumerating at any point during the enumeration. More than one application threads shall be able enumerate same container. One application thread shall be able to enumerate more than one containers at same time.

This feature enables applications to enumerate all objects in a container efficiently and allows more than one threads to enumerate one or more containers in parallel.

### Design

Containers will now be implemented virtually and will live in a large physical container that expands to use all of flash. Logical containers are differentiated by a 16 bit container id which resides in the hash table entry for each object.

A container is enumerated by running through the hash table and returning all entries that match the container id. The container id will now be stored as part of the hash tables and log records. Logging needs to ensure that the container id is preserved.

### User APIs

No change to current enumeration APIs.

### Internal APIs

None.

### Configuration

Enumeration does not require any configuration

### Issues and Limitations

* Maximum of roughly 65000 containers will be supported.
* Internal hash table now uses 25% more space.
* Some performance impact; positive in many cases.
* If we run at a million operations per second continuously for more than eight years and 11 months, counters will overflow and FDF will crash

### Statistics

Enumeration does not have any statistics.

### Testcases

|  |  |  |
| --- | --- | --- |
| # | Test case | Description |
| 1 | Enumerate single container from multiple application threads while there is no write load | 1. Create a 1G container C1 2. Fill container with data(example: memslap) 3. Start N(where N=10) application threads 4. Enumerate the container C1 from all threads   Expected Result:   1. All enumerations should succeed 2. The enumerated data from all threads must be consistent with container data 3. No memory leak |
| 2 | Enumerate single container from multiple application threads while there is write load under progress | 1. Create a 1G container C1 2. Fill container with some data(example: memslap) 3. Start dynamic work load(Memslap) which does update, insert and deletes 4. Start N(where N=10) application threads 5. Enumerate the container C1 from all threads   Expected Result:   1. All enumerations should succeed and return container objects 2. No memory leak |
| 3 | Enumerate multiple containers from multiple application threads while there is no write load | 1. Create five 1G containers C1 to c5 2. Fill container with data(example: memslap) 3. Start ten(where N=10) application threads 4. Enumerate the five containers with two threads per container   Expected Result:   1. All enumerations should succeed 2. The enumerated data from all threads must be consistent with container data 3. No memory leak |
| 4 | Enumerate multiple containers from multiple application threads while there is write load under progress | 1. Create five 1G containers C1 to c5 2. Fill the containers with some data(example: memslap) 3. Start dynamic work load(Memslap) which does update, insert and deletes 4. Enumerate the five containers with two threads per container   Expected Result:   1. All enumerations should succeed and return container objects 2. No memory leak |
| 5 | Enumerate multiple containers from one thread | 1. Create five 1G containers 2. Fill container with data(example: memslap) 3. Start 1 application thread 4. Enumerate the all five containers from one thread   Expected Result:   1. All enumerations should succeed 2. The enumerated data from all threads must be consistent with container data 3. No memory leak |
| 6 | Check if the application thread can terminate the enumeration before enumeration completes | 1. Create a 1G container C1 2. Fill container with data(example: memslap) 3. Start 1 application thread1 4. Enumerate the container C1 and terminate the enumeration after partial data is retrieved from application thread   Expected Result:   1. The enumeration should succeed 2. No memory leak |
| 7 | Check if enumeration on closed container fail | 1. Create a 1M container C1 2. Fill container with data(example: memslap) 3. Close the container 4. Start 1 application thread1 5. Enumerate the container C1   Expected Result:   1. The enumeration initializer call should fail with appropriate error code |
| 8 | Check container deletes are handled appropriately while enumeration is under progress | 1. Create a 1G container C1 2. Fill container with data(example: memslap) 3. Start N(where N=10) application threads 4. Enumerate the container C1 from N-1 threads 5. Delete the container from Nth thread   Expected Result:   1. All enumerations should be gracefully terminated before the delete 2. No memory leak |
| 9 | Check close container operations are handled appropriately while enumeration is under progress | 1. Create a 1G container C1 2. Fill container with data(example: memslap) 3. Start N(where N=10) application threads 4. Enumerate the container C1 from N-1 threads 5. Close the container from Nth thread   Expected Result:   1. All enumerations should be gracefully terminated before close operation 2. No memory leak |
| 10 | Check if enumeration succeeds after FDF restart | 1. Create five 1G containers C1 to c5 2. Fill container with data(example: memslap) 3. Start dynamic work load(Memslap) which does update, insert and deletes 4. Restart FDF 5. Start ten(where N=10) application threads 6. Enumerate the five containers with two threads per container   Expected Result:   1. All enumerations should succeed 2. The enumerated data from all threads must be consistent with container data 3. No memory leak |
| 11 | FDF performance while enumeration | 1. Create five 5G containers C1 to c5 2. Fill the containers with some data(example: memslap) 3. Start dynamic work load(Memslap) which does update, insert and deletes 4. Enumerate the five containers with two threads per container 5. Measure standard FDF performance(read, update, inserts)   Expected Result:   1. All enumerations should succeed and return container objects 2. No memory leak 3. Performance is not affected |
| 12 | Enumeration while garbage collection under progress | 1. Create five 1G containers C1 to c5 2. Fill the containers with some data(example: memslap) 3. Start dynamic work load(Memslap) which does update, insert and deletes 4. Enumerate the five containers with two threads per container 5. Kickstart the SLAB garbage collection   Expected Result:   1. All enumerations should succeed and return container objects without any duplicate 2. No memory leak |

## Mini Transaction

This feature enables applications to group one or more key value operations as a transaction and is intended for specific use by Mongo. Proper operation is defined for a single transaction at a time (ie: no nested transactions). Transaction is conducted by specific app thread. Concurrent operations by other threads are permitted. Durability of the transaction will be that of the most stringent container.

The mini transaction support does not implement full transaction support, e.g. rollback of transaction will not be implemented. It is assumed that transactions are short in time and size. Every thread can only have one active transaction at a time.

Concurrent transactions in different threads, however, permitted.

### Design

The following APIs will be exposed to application for mini transaction support:

\* FDFTransactionStart

\* FDFTransactionCommit

Three internal call introduced:

\* fdf\_trx\_start

\* fdf\_trx\_commit

\* fdf\_trx\_id\_get

Start transaction

FDF\_status\_t

FDFTransactionStart(struct FDF\_thread\_state \*fdf\_thread\_state)

Initialize a transaction by assigning a transaction id to the thread local

variable. Non zero value of above mentioned variable signifies that all

subsequent object sets will be part of the transaction. The function signals

lower level tier that transaction is started, by calling

fdf\_trx\_start(uint64\_t trx\_id).

Function returns an error status code if current thread already has a

transaction started to handle the error case of double FDFTransactionStart

calls.

Commit transaction

FDF\_status\_t

FDFTransactionCommit(struct FDF\_thread\_state \*fdf\_thread\_state)

This function signals lower level tier that transaction is going to commit by

calling fdf\_trx\_commit(uint64\_t trx\_id) and zeroes thread local transaction id

at the end.

Function returns an error status code if there is no active transaction

currently running to handle the error case of double FDFTransactionCommit.

Write Object

Nothing specific will be done on FDFWriteObject at a higer level, lower level

logic will be provider with a way of determining current thread's transaction

id instead, e.g. fdf\_trx\_id\_get(). That function should be used to keep track

of transaction log records and other required processing at a lower level.

**Recovery of container deletion process**

Conclude recovery of CMC, VMC and VDC, and enumeration metadata, and then call up to object deletion layer.

**Log layer shutdown**

In the spirit of data preservation, the log layer will support FDF shutdown.

The logging API will be expanded to include

bool log\_start\_trx( trx\_id);

bool log\_commit\_trx( trx\_id);

trx\_id transaction ID) is conveyed from the public FDF layer through the per-thread context. TRX log entries are placed in a side buffer of finite length. When the transaction is committed, the TRX log entries are appended to the current log, and processed in standard fashion. Isolation is of the dirty-read variety. If the transaction exceeds the op limit, it is aborted. log\_start\_trx() returns FALSE if a transaction is already underway, else TRUE. log\_commit\_trx() returns FALSE if the transaction was aborted, otherwise TRUE.

**Recovery of container deletion process**

Call will be provided by Container API and look like so:

void resume\_containers\_deletion( );

**Log layer shutdown**

The log layer will offer an entry point to be called by FDFShutdown.

void log\_layer\_shutdown( );

Unwritten entries (from soft durability containers) will be flushed

to the log structure on flash.

### User APIs

The following new APIs are added to FDF API.

FDFTransactionStart

FDFTransactionCommit

The following error codes are added

FDF\_FAILURE\_NO\_TRANS

FDF\_FAILURE\_ALREADY\_IN\_TRANS

### Internal APIs

The following internal calls are introduced:

fdf\_trx\_start

fdf\_trx\_commit

fdf\_trx\_id\_get

### Configuration

No configuration required for the feature

### Issues and Limitations

* One transaction per thread
* No rollback of transactions
* User responsible for transaction rollback in the case of puts failure.
* User can read dirty data, e.g. updated object of not yet committed transaction
* Transaction should be small in size and time

### Statistics and Logs

The following stats will be added

* Current number of transactions
* Total number of transactions
* Maximum transaction size
* Minimum transaction size
* Average transaction size

### Test cases

|  |  |  |
| --- | --- | --- |
| # | Test case | Description |
| 1 | Transaction to a container from single app thread | 1. Create a 1M container C1 2. Start the work load to C1(example: memslap) 3. Start N(where N=1) application threads 4. Send a complete transaction with M(where M=10) updates from the application thread   Expected Result:   1. All updates in the transaction should succeed 2. No Memory leak |
| 2 | Transaction to a container from multiple app threads | 1. Create a 1M container C1 2. Start the work load to C1(example: memslap) 3. Start N(where N=10) application threads 4. Send a complete transaction with M(where M=10) updates from N app threads   Expected Result:   1. All updates in the transaction should succeed 2. No Memory leak |
| 3 | Transactions to multiple containers from single thread | 1. Create 5 1M containers 2. Start the work load to all containers (example: memslap) 3. Start N(where N=1) application thread 4. Send a complete transaction with M(where M=10) updates from the application thread to all containers   Expected Result:   1. All updates in the transaction should succeed 2. No Memory leak |
| 4 | Transactions to multiple containers from multiple threads | 1. Create 5 1M containers 2. Start the work load to all containers (example: memslap) 3. Start N(where N=10) application thread 4. Send a complete transaction with M(where M=10) updates from the application thread to all containers. Two threads per container   Expected Result:   1. All updates in the transaction should succeed 2. No Memory leak |
| 5 | Invalid transaction – Transaction commit without transaction start | 1. Create five 1M containers 2. Start N(where N=5) application thread 3. Send “transaction commit” without “transaction start”   Expected Result:   1. Appropriate error should be returned 2. No Memory leak |
| 6 | Invalid transaction – Transaction start without ending previous transaction | 1. Create five 1M containers 2. Start N(where N=5) application thread 3. Send “transaction start”, some updates and then “transaction start” again   Expected Result:   1. Appropriate error should be returned 2. No Memory leak |
| 7 | Check if partial transaction from single thread is rolled back during recovery after crash | 1. Create a 1M container C1 2. Start the work load to C1(example: memslap) 3. Start N(where N=10) application threads 4. Start transaction 5. Send M updates(where M=100) updates from N app threads 6. Crash FDF 7. Restart FDF   Expected Result:   * The M updates should not exist * No Memory leak |
| 8 | Check if partial transactions to multiple containers from multiple threads are rolled back during recovery after crash | 1. Create a five1M containers 2. Start the work load to containers(example: memslap) 3. Start N(where N=10) application threads 4. Start transaction from each app thread to containers(2 threads per container) 5. Send M updates(where M=1000) updates from N app threads 6. Crash FDF 7. Restart FDF   Expected Result:   1. The M updates should not exists 2. No Memory leak |
| 9 | Check if partial transactions to multiple containers from multiple threads are rolled back during recovery after crash when log is rotated before crash | 1. Create a five1M containers 2. Start the work load to containers(example: memslap) 3. Start N(where N=10) application threads 4. Start transaction from each app thread to containers(2 threads per container) 5. Send M updates(where M=10000000) updates from N app threads such that log rotation occurs few times 6. Crash FDF 7. Restart FDF   Expected Result:   1. The M updates should not exists   No Memory leak |
| 10 | Check if enumeration work while transactions are under progress | 1. Create a five1M containers 2. Start the work load to containers(example: memslap) 3. Start N(where N=10) application threads 4. Start transaction from each app thread to containers(2 threads per container) 5. Send M updates(where M=10000000) updates from N app threads 6. Enumerate the containers from different app threads 7. Complete the transactions   Expected Result:   1. Enumeration should succeed and return partially committed dta 2. No Memory leak |
| 11 | Check if long running transactions are handled appropriately | 1. Create a five1M containers 2. Start the work load to containers(example: memslap) 3. Start N(where N=10) application threads 4. Start transaction from each app thread to containers(2 threads per container) 5. Send M updates(where M= infinity) updates from N app threads   Expected Result:  Appropriate warning should be displayed in the logs  Should the transaction terminate? |
| 12 | Check if a failure of one or more updates in transactions are handled properly | 1. Create a five1M containers 2. Start the work load to containers(example: memslap) 3. Start N(where N=10) application threads 4. Start transaction from each app thread to containers(2 threads per container) 5. Send M updates(where M= 100) updates from N app threads 6. Send some invalid operation such as deleting an object the does not exist 7. Complete the transactions   Expected Result:   1. Appropriate errors should be returned 2. No memory leak |

## Slab garbage collection

In the current version of FDF, the 32MB flash segments occupied by objects of a particular size in a SLAB container cannot be reclaimed and reused for storing new objects of different size after all objects in the segments are removed.

**Design:**SLAB reuse effort implements segments reclamation when they become completely empty. That functionality can be change to trigger segment reclamation not at zero objects in the segment, but when number of objects falls below some threshold, which can be compile time or exposed as a user variable. Other condition for triggering that logic is enough free slot in the remaining segments of the same class, enough to copy object of to be reclaimed segment, plus some threshold in order to avoid flip flop segment allocation/reclamation.

Amount of used SLABs is checked at every object deletion and if above mentioned conditions satisfied the remaining objects of the segment asynchronously copied to other segments of the same class one by one. Corresponding hash bucket lock will be held during object copy to prevent corruption.

Asynchronous behavior will be implemented by background defragmentation thread and task queue. User thread will signal background thread about available work by putting the number of compaction candidate segment to task queue.

The slab reallocation shall have a mode where no threshold configuration is required. The garbage collection mechanism automatically finds two segments that can be merged to one and does the merging transparently.

### User APIs

None

### Internal APIs

bool slab\_gc\_init(mcd\_osd\_shard\_t\* shard, int threshold);

void slab\_gc\_signal(mcd\_osd\_shard\_t\* shard, mcd\_osd\_segment\_t\* segment);

void slab\_gc\_end(mcd\_osd\_shard\_t\* shard);

void slab\_gc\_update\_threshold(mcd\_osd\_shard\_t \*shard, int threshold);

void slab\_gc\_print\_stats(FILE\* log);

### Configuration

Two configuration properties will handle compaction behavior:

FDF\_SLAB\_GC\_THRESHOLD

Sets the percent of minimum number of objects in the class. Falling beyond this threshold trigger segment compaction. Default value is 100 percent, what means garbage collect always, e.g. automatic garbage collection.

FDF\_SLAB\_GC

Enables(On) or disables(Off) garbage collection entirely. Default is On.

### Issues and Limitations

None

### Statistics

segments\_compacted – segments prepared for deallocation

segments\_freed – segments really deallocated

slabs\_relocated – number of slabs copied

blocks\_relocated – number of block copied

relocate\_errors – number of errors during slab relocation

gc\_signalled – count the times gc thread is signaled to start compaction

segments\_cancelled –count errors happened during compaction and segments put back to the segments list

### Testcases

|  |  |  |
| --- | --- | --- |
| # | Test case | Description |
| 1 | Check if garbage collection compacts 2 eligible segments in a single slab class adds the freed segments to freelist | 1. configure FDF with 10G storage 2. Create a container 3. Setup an workload that creates holes in two segments in a slab class such that it triggers garbage collection 4. Continue the workload   Expected result   1. The two segments should be merged as one 2. Free segment list count should increase by one 3. No memory leak 4. Logs appropriate message in the log |
| 2 | Check if garbage collection compacts more than 2 eligible segments in a single slab class and adds the freed segments to free list | 1. configure FDF with 10G storage 2. Create five containers 3. Setup an workload that creates holes in more than two segments such that it triggers garbage collection 4. Continue the workload   Expected result   1. The available segments should be merged and freed segments should be added to free list 2. No memory leak |
| 3 | Check if garbage collection compacts more than 2 eligible segments in a multiple slab class and adds the freed segments to free list | 1 configure FDF with 10G storage   1. Create five containers 2. Setup an workload that creates holes in more than two segments in all slab class such that it triggers garbage collection 3. Continue the workload   Expected result   1. The available segments should be merged and freed segments should be added to free list 2. No memory leak 3. Logs appropriate message in the log |
| 4. | Check if containers are recovered properly when FDF crashes during compaction | 1. configure FDF with 10G storage 2. Create five containers 3. Setup an workload that creates holes in more than two segments in all slab class such that it triggers garbage collection 4. Abort the FDF while pointers are manipulated   Expected result  1. The container data must be consistent |
| 5. | Check if garbage collection does appropriate actions when the segments being compacted fills up with new data | 1. configure FDF with 10G storage 2. Create five containers 3. Setup an workload that creates holes in more than two segments in all slab class such that it triggers garbage collection 4. Fill up the segments which has holes while compaction is under progress   Expected result   1. Garbage collector should skip compaction 2. Logs appropriate message in the log |
| 6 | Check if enumeration works ok during compaction | 1. Configure FDF with 10G storage 2. Create five containers 3. Setup an workload that creates holes in more than two segments in all slab class such that it triggers garbage collection 4. Enumerate the containers while compaction is under progress   Expected result   1. Enumeration should not return duplicate data |
| 7 | Measure the FDF performance while garbage collection is under progress | 1. configure FDF with 10G storage 2. Create five containers 3. Setup an workload that creates holes in more than two segments in all slab class such that it triggers garbage collection 4. Continue the workload and measure the performance   Expected result   1. The available segments should be merged and freed segments should be added to free list 2. No memory leak 3. Logs appropriate message in the log 4. Performance is not affected |
| 8 | Check if the failed writes due to “No space” automatically succeeds after a segment becomes free | 1. configure FDF with 10G storage 2. Create five containers 3. Setup an workload that creates holes in more than two segments in all slab class such that it triggers garbage collection 4. Continue the workload such that it reaches “No space” error 5. Continue write ignoring this error   Expected result   1. Write should succeed immediately after a free segment is available |

## Usability improvements

### Design

Ensure that if a FDF API call fails, a descriptive message is logged. Clean up log messages where appropriate.

**Tests**

Program which triggers error conditions; will need to manually look at the log to ensure that appropriate message is generated.

## Expiry Time

New read and write APIs will be created for setting and reading expiry times for objects.

### Design

Expiry time options will be added to existing FDFRead and FDFWrite APIs

### User APIs

The following new APIs will be added

FDFReadObjectExpiry(struct FDF\_thread\_state  \*fdf\_thread\_state, FDF\_cguid\_t  cguid,  FDF\_readobject\_t \*robj)

FDFWriteObjectExpiry(struct FDF\_thread\_state  \*fdf\_thread\_state, FDF\_cguid\_t  cguid, FDF\_writeobject\_t \*wobj)

The keys and objects are passed through the following structures.

typedef struct {

       char            \*key;

       uint32\_t        key\_len;

       char            \*data;

       uint64\_t        data\_len;

       SDF\_time\_t      current;

       SDF\_time\_t      expiry;

} FDF\_readobject\_t;

typedef struct {

       char            \*key;

       uint32\_t        key\_len;

       char            \*data;

       uint64\_t        data\_len;

       SDF\_time\_t      current;

       SDF\_time\_t      expiry;

} FDF\_writeobject\_t;

In this release, applications need to read the object and set again for changing and finding out expiry time of an object.

### Internal APIs

None

### Configuration

None

### Issues and Limitations

None

### Stats

None

### Testcases

|  |  |  |
| --- | --- | --- |
| # | Test case | Description |
| 1 | Set an object with expiry time and verify if the object is invalidated after expiry | 1. Create a 1G container 2. Set all objects in the container with expiry time   Expected Result   1. All objects should be invalidated after expiry time and current items count should become 0 |
| 2 | Check if the object does not expire if no expiry time is set | 1. Create a 1G container 2. Set some objects in the container with expiry time and others without   Expected Result   1. All objects set with expiry time should be invalidated after expiry time and other objects should exist 2. The current items count should reflect this |

## Orderly shutdown

### Design

The shutdown implementation for this release will, at minimum, flush and sync any dirty data in cache or in flight.  It is assumed that the user will not issue any other FDF operations after shutdown has been initiated.

On a best efforts basis, the shutdown code will also:

* Block all future FDF requests before starting the flush and sync.
* Set a persisted "clean shutdown" flag so that restart can be accelerated after clean shutdowns.  This must done after the flush and sync.  This will be done by calling a function provided by the logging subsystem.
* Call any existing shutdown functions that already exist for various FDF modules.  These include:
  + protocol/action (will release FDF cache memory [large!])
  + protocol/home (?)
  + messaging
* Add the most critical shutdown functions that do not already exist (eg: release mcd\_osd hashtable memory)

### User APIs

### Internal APIs

### Configuration

### Issues and Limitations

### Stats

### Testcases

|  |  |  |
| --- | --- | --- |
| # | Test case | Description |
| 1 | Check if FDF does not loose objects during graceful shutdown for each supported durability setting | 1. Create a couple of 1G container 2. Start the workload which does create, delete containers and object operations at very high rate 3. Stop workload at random point 4. Immediately initiate FDF graceful shutdown by calling FDF shutdown 5. Kill the FDF application after shutdown complted 6. Restart the FDF application   Expected Result:   1. The objects and container state must be consistent as of the shutdown point |
| 2 | Check if FDF does not loose objects during graceful shutdown for each supported durability setting while application is running | 1. Create a couple of 1G container 2. Start the workload which does create, delete containers and object operations at very high rate 3. Stop workload at random point 4. Immediately initiate FDF graceful shutdown by calling FDF shutdown 5. Keep the FDF application running 6. Restart the FDF after shutdown completes   Expected Result:   1. The objects and container state must be consistent as of the shutdown point |
| 3 | Check if FDF rejects container and object operations after shutdown is initiated | 1. Create a couple of 1G container 2. Start the workload which does create, delete containers and object operations at very high rate 3. Initiate FDF graceful shutdown by calling FDF shutdown   Expected Result:   1. All the container and object operations should fail after shutdown is initiated |
| 4 | Check if FDF takes optimized recovery path after graceful shutdown | 1. Create a couple of 1G container 2. Start the workload which does create, delete containers and object operations at very high rate 3. Stop workload at random point 4. Immediately initiate FDF graceful shutdown by calling FDF shutdown 5. Kill the FDF application after shutdown completes 6. Restart the FDF application   Expected Result:   1. The FDF should not do full recovery and should take optimized recovery path 2. The objects and container state must be consistent as of the shutdown point |
| 5 | Check if FDF does not take optimized recovery path after a crash | 1. Create a couple of 1G container 2. Start the workload which does create, delete containers and object operations at very high rate 3. Kill the FDF application 4. Restart the FDF application   Expected Result:   1. The FDF should not do optimized recovery and do full recovery 2. The objects and container state must be consistent as of the shutdown point |
| 6 | Check if graceful FDF shutdown releases all the memory | 1. Create a couple of 1G container 2. Start the workload which does create, delete containers and object operations at very high rate 3. Stop workload at random point 4. Immediately initiate FDF graceful shutdown by calling FDF shutdown 5. Keep the FDF application running 6. Restart the FDF after shutdown completes   Expected Result:   1. The objects and container state must be consistent as of the shutdown point 2. No memory leak |
| 7 | Check if per thread states are freed after shutdown is initiated | 1. Create a couple of 1G container 2. Start the workload which does create, delete containers and object operations at very high rate 3. Stop workload at random point 4. Immediately initiate FDF graceful shutdown by calling FDF shutdown   Expected Result:   1. Verify if per thread state is freed |

## Statistics Enhancement

The FDF 1.2 release contains the following enhancements to the FDF stats.

1. Admin TCP Port which allows user to connect to FDF through a tcp client(e.g telnet and query, configure stats.
2. User friendly stats output

The following commands are supported through the admin port.

1. container list

Lists the names of all the open containers

1. container stats <container name> [v]

Prints container stats on the client connection. The option v when specified prints additional Flash layer statistics

1. container stats\_dump <container name|all> [v]

Dumps the container stats of given container or all the containers to a file configured through the property FDF\_STATS\_FILE. If periodic dump is enabled, this command does not work. User needs to disable the periodic dumps to use this command

1. container autodump <enable|disable>

Enables/disables automatic periodic dumps

1. quit

exits the client connection

1. help

prints all supported commands

Example usage:

* telnet localhost 515350 and then type "container list" on the prompt
* echo “container list” | nc localhost 515350

The following configuration variables are introduced to configure :

FDF\_ADMIN\_PORT : TCP admin port. Default 51350

FDF\_STATS\_DUMP : enable/disble(1 or 0) periodic dumps. Dafault value 0

FDF\_STATS\_NEW : This variable enables the new stats output layout. By default it is enabled. If disabled, the stats are dumped in the old format.

FDF\_ADMIN\_ENABLED: Enable/disable admin port

FDF\_STATS\_FILE : file for dumping stats

**Sample Stats output:**

Timestamp:Fri Feb 15 23:53:54 2013

[Per Container Statistics]

Container Properties:

Name :container-9aa65940

Cguid :23

Size :1048576kb

Persistence :1

Eviction :0

Writethrough:1

FIFO :0

Asyncwrites :0

Durability :0

Num\_objs :1

Used\_space :512

Application requests:

num\_set\_objs\_with\_expiry = 1

num\_get\_objs\_and\_check\_expiry = 1

Overwrite and write-through statistics:

num\_new\_entries = 1

num\_writethrus\_to\_flash = 1

Cache to Flash Manager requests:

num\_set\_objs\_and\_put = 1

Flash Manager responses to cache:

num\_set\_objs\_completed = 1

Flash Manager requests/responses:

num\_set\_objs = 1

Flash layer return codes:

num\_success = 1

[Overall FDF Statistics]

Flash statistics:

num\_items\_flash = 20

num\_items\_total = 20

nun\_inval\_evictions = 4

num\_put\_ops = 20

num\_full\_hash\_buckets = 1

flash\_space\_allocated = 33554432

flash\_space\_consumed = 10240

Flash layout:

flash\_class\_map 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

flash\_slab\_map 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Application requests:

num\_set\_objs\_with\_expiry = 1

num\_get\_objs\_and\_check\_expiry = 1

Overwrite and write-through statistics:

Cache to Flash Manager requests:

num\_set\_objs\_and\_put = 1

Flash Manager responses to cache:

num\_set\_objs\_completed = 1

Flash Manager requests/responses:

num\_set\_objs = 1

Flash layer return codes:

num\_success = 1

Cache statistics:

num\_hash\_buckets\_in\_cache = 100000

num\_cache\_partitions = 10000

num\_objects\_in\_cache = 65

max\_cache\_capacity = 100000000

current\_data\_size\_in\_cache = 8552

current\_key\_and\_data\_size\_in\_cache = 9384

background\_flush\_progress = 100

max\_parallel\_flushes = 8

max\_parallel\_bg\_flushes = 8

time\_to\_wait\_after\_bgflush\_for\_nodirty\_data = 1000

max\_percent\_limit\_on\_modifiable\_cache = 100

num\_app\_buffers\_inuse = 20

num\_cache\_ops\_in\_progress = 23