Data Resilience in the dCache Storage System

A L Rossi¹, F Adeyemi², A Ashish², G Behrmann³, P Fuhrmann³, D Litvintsev¹, P Millar², T Mkrtchyan², A Mohiuddin², M Sahakyan², J Starek², S Yasar²

¹Fermi National Accelerator Laboratory, PO Box 500, 60510, Batavia, IL, USA
²IT Department, DESY, Notkrstrasse 85, Hamburg, Germany
³NORDUnet, Copenhagen, Denmark

E-mail: arossi@fnal.gov

Abstract. In this paper we discuss design, implementation considerations, and performance of a new Resilience Service in the dCache storage system responsible for file availability and durability functionality.

1. Introduction: file replication and dCache quality of service

From early in its development, the dCache storage system [1] has sought to provide file durability and availability on disk-only systems by generating multiple permanent copies (replicas) of files and distributing them across different locations (pools). This capability figures prominently within a broader set of quality of service features which the dCache team continues to expand and improve.

Given the crucial role played by file replication, it is therefore of utmost importance that the components responsible for it be aligned with the evolving dCache code base and that they function in the most efficient manner possible. The following paper first presents the rationale for the complete overhaul of replica handling, detailing the principal features of the new service. It then discusses a number of design and implementation decisions made during this process, and concludes with some remarks on measured performance.

2. From Replica Manager to Resilience

The original version of the replica service, also called Replica Manager, used pool-to-pool copying to guarantee that the number of replicas of newly written files met a pre-set minimum requirement. It maintained this minimum by creating additional replicas, if one or more pools containing that file go offline or otherwise become unreadable, and by eliminating excess replicas when the old locations once again become readable. The manager employed a special pool group in order to distinguish pools under its control from the rest of the dCache configuration; thanks to this partitioning, dCache could be run in what was referred to as "hybrid" mode, where access to one set of files stored only as multiple persistent disk copies coexisted alongside the usual caching and retrieval of others stored on tape.

Although minor adjustments and bug fixes have been applied to the Replica Manager, there has been no significant modification of its internals since 2007. Its obsolescence is particularly conspicuous in the utilization of its own database and schema, one which the subsequent adoption of the Chimera namespace [2] has rendered largely redundant. Moreover, the Replica Manager's tables are populated by querying the individual pool repositories, which is not as reliable or efficient a way of obtaining that information as querying the namespace database now is.
There is also a certain brittleness in the way the Replica Manager views and handles replication requirements. First, the number of replicas is determined by a global minimum and maximum for all files in the pool group. Second, the manager does not allow for more than one such group. To provide for multiple sets of requirements, it is necessary to run as many managers as there are different categories, a cumbersome and ultimately inefficient solution. But beyond those limitations, there is a more fundamental issue at stake: replication requirements should be tied directly to the system’s representation of the class of files being stored, and not be based on an organizing principle (pools and pool groups) of the storage infrastructure itself.

For these reasons, we decided to completely redesign this service. To distinguish the new version from its predecessor, we will henceforward refer to it as the "Resilience service", or simply as "Resilience". Resilience is intended to replace the Replica Manager, which has been deprecated and will be eliminated from dCache releases in the near future.

2.1. Redesign Goals

In designing a more up-to-date replication service, we were guided by the following objectives:

a) Elimination of dependency on a separate database without sacrificing scalability or reliability;

b) Horizontal scalability in both the number of pools and the number of files in the namespace (our benchmark figure was set at 100 pools, each with at least one million files);

c) Optimum use of existing dCache components and dCache communication;

d) Negligible impact of replication on other dCache core services;

e) Improved responsiveness to pool state changes and file availability;

f) A more flexible framework for defining replication requirements on sets of files;

g) No restrictions on pool sharing;

h) Live response to manual changes in pool configuration or storage requirements;

i) More tools for diagnosis, performance monitoring, and failure recovery or mitigation.

From the outset, the elimination of a resilience database was considered desirable not only because of the burden of synchronizing that data with the namespace and pool repositories, but because a good portion of it would still have to be cached in memory for improved performance. The horizontal scaling requirement further dictated that the service as a result would demand a larger memory allocation; hence the only clear advantage of continuing to back resilience with on-disk metadata persistence would be for crash recovery. To compensate for the lack of a database, periodic checkpointing of operation metadata to a file offered a possible solution. But making such checkpointing lossless also would mean introducing locking on the internal data structures, with an obvious reduction in concurrent processing and increased queuing. On the other hand, a combination of frequent but lossy checkpointing and less frequent periodic rescanning of all resilient pools promised an adequate guarantee that any copying or removal which may have been missed because of service failure and restart could take place within a reasonable amount of time, those operations not caught by the checkpointing eventually showing up during the full pool scans.

More difficult to evaluate was the best approach for integrating this service into the modern dCache architecture (c & d above). In the next section, we present some of our experiences in prototyping the component interaction.

From discussions with current and potential users of this service, it became clear to us that the ability to make adjustments without requiring a service restart when pools are added or eliminated, or resilience requirements are added or modified, was of prime importance. But it was also evident that their replication needs, based on classes of users or experiments, made mapping to pool groups awkward. Resilience constraints are now expressed via two parameters on the storage unit: required indicates the number of replicas each should receive; onlyOneCopyPer is used to partition replicas among locations on the basis of a tag system (a generalization of a global property of the Replica Manager which either allowed or disallowed copies of the same file in pools residing on the same
host). As part of the pool manager configuration, these resilience properties are controlled by the dCache administrator.

The Replica Manager handles only incoming (new) files marked precious, denoting a file written to a disk but not (yet) flushed to tape. To prevent such files from being flushed (thereafter being marked as cached, and thereby becoming subject to cleanup), one must apply the setting lfs=precious to the pools to which those files are directed. This setting was consequently required on all pools in the resilient pool group, meaning that these pools could not be used to host files having to go to tape; in other words, pools could not be shared between resilience and the rest of dCache. This incompatibility was further complicated by the introduction of the RetentionPolicy and AccessLatency attributes. For now only files with a CUSTODIAL RetentionPolicy are initially written to the pools as precious. Not only did this cause resilient and tape-backed files to share the same RetentionPolicy value, but it also left the property which should actually distinguish them, AccessLatency, largely irrelevant. To summarize: in abstract terms, coupling file replication to pool settings in this way was intrinsically incorrect; practically, it led to a confused configuration procedure.

All of this suggested the need for a fundamental change: Resilience should handle only those files a) with ONLINE AccessLatency, b) which are marked for replication (i.e., have their source on a pool which belongs to a resilient pool group), and c) have a storage classification with a replica count greater than one. These revised semantics permit a clean separation of resilient files from non-resilient, thus allowing them to cohabit the same pool, with the concomitant benefit of a potentially more economical use of disk. They also permit files to be stored both on tape and with permanent replicas on disk (CUSTODIAL + ONLINE), a combination which was not previously feasible using the Replica Manager.

While the Replica Manager did retry failed copy operations, the error logic built into Resilience is more nuanced (see section 5). In addition, Resilience raises an alarm if an operation fails because there are no accessible replicas of that file. This most typically occurs if all pools containing the replicas for that file go offline at once.

3. Prototyping component interaction

Interactions in dCache are built on messages between components (cells) serving to establish and execute transfers or removals of files. Aside from doors, which handle connections from clients according to a specific protocol, and pools, where the files are stored and served, there are several core services participating in these interactions. For new file writes, the most important of these are the PnfsManager, which controls the namespace, and the PoolManager, which, aside from selecting pools, also monitors their status and controls their configuration. These interactions can be rather complex, so finding the optimum (and conceptually clearest) way to situate resilience within this internal communication network was not immediately obvious. Consequently, we had to iterate over several prototypes before settling on the most favorable solution.

We experimented with three major design patterns which we will designate the hub, the distributed pipeline, and the mirrored subscriber, respectively. In all three of these patterns, it was understood that the actual work of copying would utilize the full-featured support of the pool MigrationModule; replica removal, on the other hand, would be managed by Resilience itself.

3.1. Hub interaction (prototype 1)

In this version, Resilience runs as a self-contained service, pulling information from the core services in response to arriving messages, and eventually running a task which communicates with the pool. File operations are serialized on their unique identifier (pnfsid) and treated recursively, each cache location update triggering another verification as to whether more work must be done. Operations on different files are handled concurrently. Testing of this “pull” model revealed that under high ingest rates, communication traffic could cause services to become intermittently unreachable, slowing down processing due to timeouts and increased retry rate. This model was soon discarded in favor of putting more of the work closer to the (meta)data.
3.2. Distributed pipeline interaction (prototype 2)

The intention of this design modification was to streamline communication between components by redistributing work. Resilience functions are split up and localized inside the core services and the pools. Cache location and pool status changes are handled locally, and the metadata necessary to verify and execute the operation is passed through what is mostly a one-way channel from one resilience handler to the next. Given the above-mentioned serialized recursive procedure for file operations, there is no real risk posed by stale information with regard to file attributes or number of locations; even the possibility that a source or target might become unavailable between initial selection of the pool and the actual execution of the task is not problematic, as the subsequent error would trigger a retry of the operation.

By itself, a distributed pipeline is nevertheless inadequate to handle the full semantics of replication, which also requires global information, such as whether a pool scan has completed or not, to perform correctly; neither does it easily allow for a view into the system as a whole, necessary for monitoring, task cancellation and recovery. It therefore becomes necessary to layer over the pipeline an operation or task registry; the registering, unregistering, and update notifications then introduce a new set of synchronizations. This distributed state machine results in an increase in component intercommunication and a complex failover when one or both of the core components (containing the resilience handlers) goes down, not to mention engendering code which is unnecessarily intricate and hard to maintain.

3.3. Mirrored subscriber interaction (final version)

Instead of moving resilience handling into the core modules, an equally effective way of reducing component interactions is to mirror the other components inside the Resilience service. With the exception of communication with the pool, all processing of resilience operations is again centralized and relatively self-contained.

Mirroring the PnfsManager means giving the Resilience service its own set of components for the namespace access API and its own connection pool to access the Chimera database. Mirroring the PoolManager means keeping a copy of the information held in the PoolMonitor. After some evaluation, it was decided the best way to achieve the latter was to have PoolManager publish on a special topic a message containing a refreshed copy of the PoolMonitor, and to have the interested parties (such as Resilience) subscribe to receive updates periodically. Direct interactions with the PoolManager are thereby eliminated. While Resilience still receives cache location update messages relayed by PnfsManager, it no longer requires any handlers to run in the same domain (JVM) as the latter.

One of the problems brought into relief by the previous two prototypes was that of memory footprint. Serialized recursive handling of operations on a given file, with its repeated queuing of verification and execution tasks, can potentially cause memory bloat. This is especially true when there are concurrent pool scan operations in progress. Throttling pool scans by using a blocking operations queue was not, however, a feasible solution, as it slowed down the processing of new files as well, and might eventually cause the queue responsible for the tasks handling those messages in turn to overflow. The data held in memory was thus restructured, as was the handling of multiple requests on a single pnfsid. Operation map entries were now given a counter, incremented upon receipt of another operation request for that pnfsid, decremented upon the completion of a particular pass of the operation. When the counter reaches zero, the entry is removed. Initial requests for new files immediately determine the number of missing replicas, and set the counter so that the necessary number of passes will be made. This is also true for the pool scanning done periodically. Scans from pool status updates (changes from UP to DOWN or vice versa), on the other hand, only increment the count by one, since only one location is directly implicated in the change. In essence, recursion was replaced by iteration.

Figure 1 illustrates the final version; it includes only the interactions involved in copying a new file; while it does indicate how pool status changes are conveyed to Resilience, it omits the processes...
connected with scanning the namespace for the resilient files on a specific pool. Figure 2 shows the full architecture.

![Figure 1 – Component interaction (new file writes), final version](image)

![Figure 2 – Resilience service architecture](image)

Some further discussion of optimizations applied to the main data structures follows.

4. Memory and concurrency optimizations

4.1. Message handler

Since at each pass of an operation, replica and pool status are re-verified, operations whose counts have been incremented beyond the required number do not risk the creation or deletion of too many replicas; operations are simply discarded if it is discovered that there is no work left to be done.
Nevertheless, it is desirable to avoid the extra time in the queue and the extra work this entails, whenever possible. Since operations are no longer handled recursively, only the first new cache location message originating outside of the Resilience "loop", so to speak, actually needs to be processed. To differentiate the messages pertaining to client-pool transfers from those resulting from Resilience operations, the message handler adds a session id which follows the operation through its entire lifecycle and across all domains; all messages received by Resilience which are marked by this id can be safely ignored.

4.2. Namespace access

Since Resilience now carries its own interface to the Chimera database, care must be taken to tune the use of database connection pools. Not only must the RDBMS (PostgreSQL) configuration be set to allow a sufficient number of connections for both PnfsManager and Resilience, but Resilience threads must be properly matched to connections so as to eliminate contention. The default configuration ensures that 200 concurrent file operations and 5 concurrent pool scans can be handled without blocking, which should be more than adequate in most circumstances. These settings can be adjusted.

Offloading onto the database the brunt of the verification logic, with the aim of returning only the locations of those files on a pool that lacked or had too many replicas, proved to be difficult to optimize. Doing this computation in memory using a cursor over the list of pnfs with an AccessLatency of ONLINE found on a specific pool outperformed the queries involving more complex joins.

4.3. File operation map

The heart of the Resilience service lies in the data structures dedicated to the single file operation. Because our goal is to be able to keep millions of such operations in memory, we followed a few strategies for reducing the memory footprint of each entry in the map used to track them. Primitive integers and static methods, rather than Java Enum objects, were used for constant values, and primitive integers representing indices rather than object references were used to point to entries in the pool information map.

The map entry needed only selective synchronization to protect just three of its internal fields, and only at three specific points in its lifecycle; otherwise, modification of its attributes belonged solely to a specific thread at distinct (i.e., implicitly serialized) instants. For the map itself, Java’s ConcurrentHashMap class, which permits concurrent access without locking, was the most efficient solution for allowing multiple inserts and reads to occur concurrently with any consumer thread removes. Only the initial queuing and cancellation required additional synchronization. The single adverse consequence of using the concurrent map is that of potentially dirty snapshots. This can occur while listing (through the admin interface), and also, as alluded to in section 2, in checkpointing.

For the actual writing of the checkpoint data, experimentation with Java NIO showed that the serialization involved was much too costly to handle millions of records within a minute-long cycle. We opted instead for a simple comma-delimited list written to a text file.

One final point to mention in this regard is loop performance. While the main consumer thread is doing work, it is outside any synchronization block, and thus cannot receive notifications. For this reason, a counter is used to track calls to signal the consumer made during this phase, and upon completion, the counter is checked to see if more work is immediately available before the consumer thread waits. This technique is also used on the pool operation queues.

4.4. Pool operation map

This data structure associates an operation entry with each known resilient pool in order to keep track of scanning done periodically and on pool status updates. In this case, Java’s LinkedHashMap is used to maintain implicit temporal ordering via queuing and dequeuing, so that pool operations with expired timestamps are near the head of the list.
4.5. Pool information map
This is a specialized set of indices and lists constructed from the PoolMonitor information, and refreshed only when there is a difference between the current state and that conveyed by a message received every 30 seconds from the PoolManager. This class benefits, therefore, from read-write synchronization, since there will be many more reads of for the most part stable information. This map also uses a special implementation of the Java List API, NonReindexableList, where new index numbers are always monotonically increasing, deletions do not cause index numbers to change, and element mutation is prohibited. This is crucial for the internal logic of Resilience, since live (running) references to pools or groups are by index number and must survive intervening map updates.

5. Error handling, fairness and starvation avoidance
The Resilience service makes a best effort to handle failed operations. Exceptions of specific types have been classified as either retriable or non-retriable. In the latter case, Resilience will seek to use a different source or target, if alternative locations exist. Failing that, the error is considered fatal, and an alarm is raised reporting that replication of the pnfsid was not currently possible, but that another attempt will be made during the next periodic pool check. A broken replica, when discovered, is removed and a new one made, if another source exists.

When electing waiting file operations to run, Resilience observes a policy of preferring the availability of at least two copies of the file. Hence, operations on files with currently only a single replica are promoted to the head of the waiting list. When the copy successfully terminates, but one or more additional replicas are needed, the operation is requeued. A non-fatal operation error will not cause it to lose its place in the queue.

A file operation can be generated either from an incoming cache location message, or from the scan of a location triggered either by a pool status change, a configuration change, or a periodic verification. Resilience treats the former as "foreground", the latter as "background" operations, and attempts to balance them fairly by promoting operations to the running state in proportion to the number of operations of that kind that are waiting. In order to avoid starvation, Resilience observes a maximum percentage for the running slots that either type can occupy when both types are queued.

6. Performance testing
During each iteration of the design-implement cycle, a set of stress tests was run on a full deployment of the system with approximately 100 pools. These tests maintained a constant load of at least 100 concurrent clients continuously writing 1 byte files without pause or timeout, for periods of up to 72 hours. The storage configuration for the test-bed was comprised of diverse units and replication requirements. Ingest of new files was also run simultaneous with pool scans triggered by taking pools offline and then returning them to service. Monitoring of heap and thread behavior in Resilience, the core services and the pools was conducted using JProfiler [3]. Resilience statistics were activated and the results plotted for the duration of the run. Over the course of several days, some slowdown in processing rates could be observed, but this was largely attributable to the increasing size of the namespace and burden on the pools; in general, Resilience was observed to run pari passu with PnfsManager in terms of the rate with which the latter would relay cache location messages.

Figure 3 shows the results of a 6-hour ingest-only test run on the final version of the Resilience service. In this case, there were 105 clients (using the anonymous DCAP/dccp protocol), with 75% of the files given 2 replicas and 25% given 3. Performance and stability are shown in terms of the rate of file operations sustained over time, and the ratio of copy operations to incoming new file locations over time. The histogram at the top left shows file operations per second obtained for the duration of the test. Average was found to be 745 +/- 2 Hz. The top right plot shows the stability of file operations per second over time. The bottom plot shows the ratio of file operations to new locations over time. The deviation from unity indicates a small amount of queuing/delay.
Conclusion

We have described the design and development of a new Resilience Service for the dCache storage system that meets the specified requirements. In the future this system will be extended to work with QoS and file lifecycle policies, as part of the development under the data management INDIGO DataCloud project [4].

Acknowledgments

Fermilab is operated by the Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy. We would like to thank FermiCloud Support for providing client nodes for performance testing.

References

[1] http://www.dcache.org
[2] Fuhrmann P, Gasthuber M, Mkrtchyan T, Proceedings of Computing in High Energy Physics Conference, C04-09-27, 2004, p 1180-1182.
[3] https://www.ej-technologies.com/products/jprofiler/overview.html
[4] https://www.indigo-datacloud.eu