DB Back-ended Filesystem for Science

Tigran Mkrtchyan[1,4,*,†], Krishnaveni Chitrapu[2,†], Dmitry Litvintsev[3], Svenja Meyer[1], Paul Millar[1], Lea Morschel[1], Albert Rossi[3] and Marina Sahakyan[1,†]

[1]*Deutsches Elektronen-Synchrotron (DESY), Notkestrasse 85, 22607 Hamburg, Germany*
[2]*National Supercomputer Center, SE-581 83 Linköping, Sweden*
[3]*Fermi National Accelerator Laboratory (FNAL), Batavia, 60510-5011 IL USA*
[4]*FernUniversität in Hagen, Universitätsstraße 47, 58097 Hagen, Germany*

Abstract
dCache is a distributed storage system developed at Deutsches Elektronen-Synchrotron (DESY) in collaboration with Fermi National Accelerator Laboratory and the Nordic eInfrastructure Collaboration (NeIC) to manage large numbers of disk servers and ensure transparent data migration to and from archival storage. Its multifaceted approach provides an integrated solution to support various scientific use cases using the same storage infrastructure, including high throughput data ingest, data sharing over wide area networks, efficient access from High-Performance Computing (HPC) clusters, and long-term data persistence on tertiary storage. The namespace/metadata component of dCache, known as Chimera, is built on top of a relational database and heavily relies on ACID (atomicity, consistency, isolation, durability) semantics to maintain filesystem consistency and integrity.

This paper offers an overview of Chimera's current capabilities and design. Additionally, it emphasizes the necessity for future enhancements to ensure scalability and meet the evolving demands of scientific research.

Keywords
Distributed Storage System, Filesystem Metadata, High-Performance Data Access, Scalability

1. Introduction

The ever-increasing amount of data that is produced by modern scientific facilities like EuXFEL, PETRA III or LHC puts high pressure on the data management infrastructure at the laboratories. The challenges that have to be addressed span over the full data life-cycle: from ingest and efficient data analysis, up to long-term preservation. Even though object stores, like Amazon S3[1], become more popular, typically data is stored in POSIX-compliant[2] filesystems, e.g. stored as a large number of files organized in a hierarchical directory structure. To achieve the desired performance, durability, and parallelism, such filesystems are implemented as distributed storage clusters, that almost linearly scale the capacity and the aggregated bandwidth with the number of installed data servers in the systems. One such storage system is dCache[3], which is developed by Deutsches Elektronen-Synchrotron (DESY) in collaboration with Fermi National Accelerator Laboratory and Nordic eInfrastructure Collaboration (NeIC).

dCache provides a system for storing and retrieving huge amounts of data, distributed among a large number of heterogeneous server nodes, under a single virtual filesystem tree with a variety of standard access methods. It strictly separates the file namespace of its data repository from the actual physical location of the datasets. The file names, attributes, and filesystem tree are managed in an internal database and exposed through a namespace component. By splitting a file's metadata and data, dCache uses a unique identifier for each file which is independent of the file's name and location. By using this level of indirection, dCache can store multiple copies of a file, dynamically add or remove new locations and make use of external storage like S3 or tape. The system horizontally scales with the number of data servers. By adding new nodes into the system both storage capacity and aggregated data bandwidth grow.

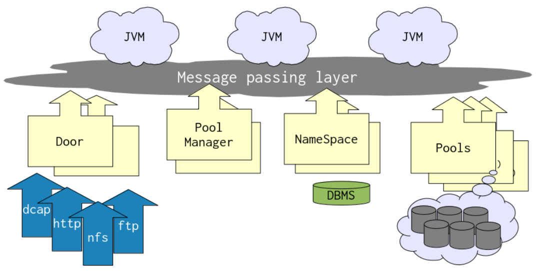


Figure 1: dCache overview

35th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), May 22-24, 2024, Herdecke, Germany.
*Corresponding author.
†These authors contributed equally.
tigran.mkrtchyan@desy.de (T. Mkrtchyan); krishnaveni@nsc.liu.se (K. Chitrapu); litvinse@fnal.gov (D. Litvintsev); svenja.meyer@desy.de (S. Meyer); paul.millar@desy.de (P. Millar); lea.morschel@desy.de (L. Morschel); arossi@fnal.gov (A. Rossi); marina.sahakyan@desy.de (M. Sahakyan)

A simplified dCache architecture is demonstrated in Figure 1. There are four main components: *doors*, the user entry points, that speak one of the supported access protocols; *pools*, the data servers that store the data and talk all supported protocols; *pool manager*, the component which is responsible for the data placement, e.g, selects which pool should be used the a given transfer; and the *name space*, a component that stores files metadata, hierarchical file system view and enforces POSIX semantics on file system operations. All components talk to each other by sending or receiving messages over a TCP/IP network. For high availability and load balancing, multiple copies of dCache components can be started in a single deployment. The topology auto-discovery and coordination between multiple instances of dCache services are based on Apache Zookeeper[4] - an open-source server for highly reliable coordination of distributed applications. The resilience of data is provided by redundant file replicas or a tape copy.

The namespace/metadata component of dCache[3] is built out of several layers (Figure 2). The top one called *PnfsManager*[1], is responsible for interactions between the rest of the dCache and under laying filesystem backend. The second one is a filesystem abstraction layer that might have multiple implementations. Nowadays only one implementation is used, called *Chimera*[6][2], which is built on top of a relational database. On a start, chimera detects the database flavor and enables database-specific optimizations. A generic SQL driver is used if such a database flavor-specific driver is missing. In production deployments, the PostgreSQL[7] database is used. The development and unit testing relies on embedded HSQLDB[8]. Such architecture allows small evolutionary changes, known as *strangler tree pattern*[9], to evolve the namespace component to respond to new requirements or technology changes.

[1]The name comes from Pretty Normal File System[5], the original backend for file metadata in dCache

[2]The name comes from an animal from ancient Greek mythology with a lion's head and fore parts, a goat's body, a dragon's rear, and a tail in the form of a snake.

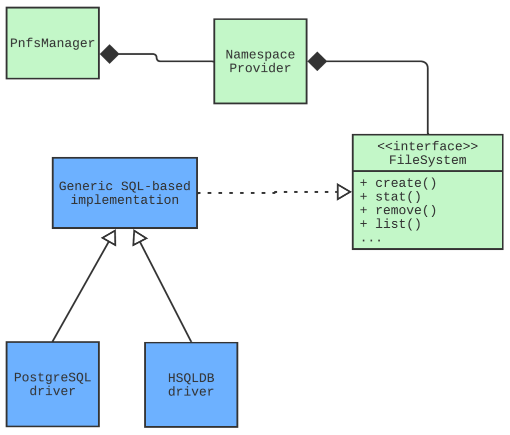


Figure 2: dCache's namespace implementation diagram.

The filesystem operations are available as a Java-API layer that utilizes database transactions to guarantee the filesystem consistency. dCache's namespace highly relies on database's ACID (atomicity, consistency, isolation, durability)[10] semantics for filesystem consistency and integrity: any file system change is performed atomically in a single transaction; transaction isolation guarantees, that end users and applications always see a consistent file system state; concurrent updates to the same filesystem object are synchronized. For example, by concurrent creation of a file 'foo' in a single directory, only one client will succeed, all others will fail with 'file exists' error; and finally, all committed file system modifications will remain committed even in the case of a system failure.

The operational experience over the latest decade has proven that the existing design easily handles the current metadata access requirements (10kHz), including high availability (HA) provided by underlying database technology.

2. Chimera Design

The namespace component of dCache is designed to address the following requirements:

Unique filesystem object IDs independent from name File names are not persistent, while data is. Users can rename files, but still be able to access the original data;

Metadata associated with files and directories An arbitrary metadata can be associated with files, in particular, storage system-specific information like tape name, offset and so on;

Name-to-ID and vice versa mapping By referencing files in the storage system by ID we need a possibility to find the file ID while users will operate by file names;

Filesystem consistency In a highly concurrent distributed system all clients should see a consistent view of the namespace;

Ability to bypass POSIX interface Though end-users typically use the POSIX interface to access the data, the system itself, as well as system administrators need an alternative path for maintenance operations or functionality beyond the POSIX standard.

The database schema contains several tables that have *foreign key* constrain to a main table with a list of all filesystem objects, known as *inodes* (Listing 1). Note, that filesystem objects are represented by two unique IDs. One is the database auto-generated ID, which is used for reference from other tables, and one externally provided UUID-based identity, which is used by the rest of the systems to identify files on data servers. Such internal+external combination reduces the storage requirements of the database engine (int vs. char(36)) and allows the merging of multiple instances into a single one, if needed. The filesystem hierarchical view, the file system tree, is implemented as *adjacency list*, see Listing 2, which is well suited for single path element lookups, havely used by UNIX virtual filesystem layer[11]. An example of file the creation SQL procedure is demonstrated in Listing 3.

```
CREATE TABLE t_inodes (
    ino serial PRIMARY KEY, -- obj id
    id char(36) UNIQUE, -- UUID used
        by storage systems
    type integer NOT NULL, -- object
        type
    mode integer NOT NULL, -- POSIX
        permissions
    nlink integer NOT NULL, --
        number of references
    uid integer NOT NULL, -- owner
        numeric id
    gid integer NOT NULL, -- owner
        group numeric id
    size bigint NOT NULL, -- object
        size in bytes
    crtime timestamp NOT NULL, --
        object creation time
    ctime timestamp  NOT NULL, --
        object's last attribute
        change time
    atime timestamp  NOT NULL, --
        object last access time
    mtime timestamp  NOT NULL, --
        object last modification time
);
```

Listing 1: Inodes table

```
CREATE TABLE t_dirs (
    parent bigint NOT NULL, --
        parent object id
    child bigint NOT NULL,  -- child
        object id
    name varchar(255) NOT NULL, --
        child's name in parent
        directory
        FOREIGN KEY (parent)
            REFERENCES t_inodes(ino),
        FOREIGN KEY (child)
            REFERENCES t_inodes(ino),
        PRIMARY KEY (parent, name)
);
```

Listing 2: Directory hierarchy table

Other tables are used to represent extended file attributes, checksums, user-defined tags, pointers to the physical location of the file, data placement policies and other information used by dCache.

```
    BEGIN
        -- create a new inode record
        INSERT INTO t_inodes (ino,
            ...) VALUES (ino, ...)

        -- create an entry with a
            given name in the parent
            directory
        INSERT INTO t_dirs (parent,
            child, name) VALUES
            (parent_ino, ino,
            'obj_name')
    COMMIT
```

Listing 3: Create entry example

If the parent directory does not exist or already contains an object with the given name, then the database transaction will fail to guarantee the filesystem consistency. With such a directory structure renaming a file or directory performed by an update of a single record, and moving the whole directory subtree into a new location is independent of the number of filesystem objects and the depth of that subtree.

The database schema of Chimera allows very efficient so-called singleton queries, which are queries that return a single row, for example, when querying file attributes or checking an object's existence in a directory, as well as a directory listing. Moreover, by exposing filesystem internals via DB query interface, some filesystem maintenance operations can be performed bypassing the POSIX interface, thereby not being limited by it. An example of such operations is the calculation of storage usage per user, finding directories with an abnormally large number of child objects (files or directories), or data deduplication based on checksums stored in one of the auxiliary tables. Nevertheless, there are downsides to such a design as well. Though Chimera is very effective for lookups in a single directory, accessing the files by full path requires multiple sequential lookups per file path element. Such accesses are typical for URL-based protocols, like HTTP. Though *nested sets* based representation of the filesystem hierarchy is more efficient for fullpath-based access, it comes with a high performance penalty when a new tree nodes, e.g. new directories, are created. For that reason, even though the majority of remote data transfers are performed by the HTTP protocol, the constantly changing large directory trees and the dominance of POSIX-like access from the local compute clusters (the direct file system mounts through NFSv4.1[12]) define the optimization priorities.

3. Transactional requirements

As mentioned above, Chimera highly relies on ACID capability of the underlying database. Moreover, some filesystem operations operate on multiple objects (records) at the same time and therefore require atomic operations on multiple records in a single transaction. For example, moving a file from one directory to another should delete the entry in the source directory, if exists, delete the file with the same name in the destination directory (this already includes updates on a directory structure and list of all inodes), and, finally, create a new record in the destination directory. To ensure consistency, all four updates must be performed as an atomic operation and, thus should be executed as a single transaction. However, strong consistency guarantees provided by relational databases introduce scalability limitations:

- Database transaction serialization in workloads, where a large number of concurrent clients update a single directory.
- Huge tables require large disk storage, memory and additional CPU resources on database servers. In HA clusters, all nodes must fulfil such requirements.
- Clients that are querying different sets of metadata can't be dynamically served by multiple servers after the connection to a database has been established, thus dynamic workload distribution is not possible, which results in overloaded servers on one side, and underutilized servers on the other.

The popular *NoSQL* solution doesn't have the limitations of traditional *RDMS* but has weaker consistency guarantees.

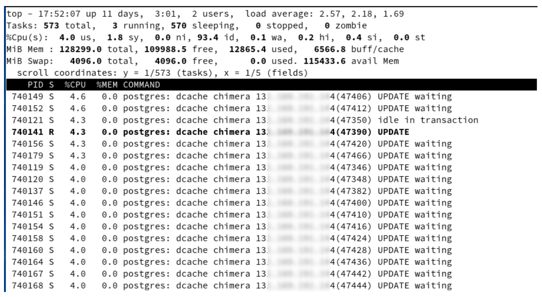
There are several attempts to make scalable and consistent ACID-compliant databases, so-called NewSQL databases, for example CockroachDB[13]. By supporting the PostgreSQL wire protocol CockroachDB is almost a drop-in replacement for dCache's namespace database. Unfortunately, basic performance tests have shown the poor performance of CockroachDB in comparison to a stand-alone PostgreSQL server.

4. Future work

Though the system's scalability satisfies today's requirements, it might become a bottleneck for upcoming detector and accelerator upgrades, which are expected by PETRA-IV in 2028, or the High Luminosity phase of the LHC starting in 2029. According to the *ATLAS Software and Computing HL-LHC Roadmap*[14], in the High-Lumi LHC mode, the ATLAS experiment will record 7-10 times more data than today, thus putting higher demand on the storage system, including the metadata component. The same data rate growth is expected by other experiments at LHC as well. To handle the amount of generated events, it is common practice for experiments to write data in multiple parallel streams, often into a single directory.

For the metadata service (as well as for storage and database systems), there are two important metrics: latency and throughput, which measure different aspects of system performance. Latency refers to the time it takes for a single request to be completed. Low latency is desirable because it means operations are completed quickly, resulting in faster response times for applications. Throughput, on the other hand, measures the rate at which multiple requests are executed simultaneously. Though ideally, we want to improve both, throughput plays a major role, as it directly affects the system's scalability.

With the current design, a large number of concurrent writes end up in database update serialization, as demonstrated with a synthetic load test in Figure 3, where all processes are waiting for an exclusive lock on the same database entry, resulting in low overall throughput, even if the rest of the system scales horizontally or vertically, i.e. scales with a number of nodes in the system or more powerful machines, respectively.



```
top - 17:52:07 up 11 days,  3:01,  2 users,  load average: 2.57, 2.18, 1.69
Tasks: 573 total,   3 running, 570 sleeping,   0 stopped,   0 zombie
%Cpu(s):  4.0 us,  1.8 sy,  0.0 ni, 93.4 id,  0.1 wa,  0.2 hi,  0.4 si,  0.0 st
MiB Mem : 128299.0 total, 109988.5 free,  12865.4 used,   6566.8 buff/cache
MiB Swap:   4096.0 total,   4096.0 free,      0.0 used. 115433.6 avail Mem
  scroll coordinates: y = 1/573 (tasks), x = 1/5 (fields)
    PID S  %CPU  %MEM COMMAND
 740149 S   4.6   0.0 postgres: dcache chimera 13[redacted]4(47406) UPDATE waiting
 740152 S   4.6   0.0 postgres: dcache chimera 13[redacted]4(47412) UPDATE waiting
 740121 S   4.3   0.0 postgres: dcache chimera 13[redacted]4(47350) idle in transaction
 740141 R   4.3   0.0 postgres: dcache chimera 13[redacted]4(47390) UPDATE
 740156 S   4.3   0.0 postgres: dcache chimera 13[redacted]4(47420) UPDATE waiting
 740179 S   4.3   0.0 postgres: dcache chimera 13[redacted]4(47466) UPDATE waiting
 740119 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47346) UPDATE waiting
 740120 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47348) UPDATE waiting
 740137 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47382) UPDATE waiting
 740146 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47400) UPDATE waiting
 740151 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47410) UPDATE waiting
 740154 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47416) UPDATE waiting
 740158 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47424) UPDATE waiting
 740160 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47428) UPDATE waiting
 740164 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47436) UPDATE waiting
 740167 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47442) UPDATE waiting
 740168 S   4.0   0.0 postgres: dcache chimera 13[redacted]4(47444) UPDATE waiting
```

Figure 3: Database write starvation

The POSIX I/O interface was defined in the late 80's for single-node systems with a single disk and, thus, was not designed for the highly concurrent computing environments we have today. To reduce the load on the metadata server some distributed storage systems don't follow POSIX semantics and provide a special application-level library that provides access to the stored data. Google's File System (GFS) [15] doesn't have a concept of a parent directory and stores the files by full name, thus file creation does not require a write lock on a parent directory[3]. Such systems might provide a high update rate of file metadata, however, they require application modification and can not be used as a general-purpose storage solution in multi-science environments.

The dCache's namespace is not the only attempt to implement the filesystem's metadata part in a relational database. The TableFS[16] follows a similar approach, though it uses a single table for the list of all inodes and the directory tree structure. However, by providing a strong consistency through back-end DB implementation the concurrent updates in the single directory TableFS are serialized as well.

To address those data management challenges, the dCache developers are investigating metadata catalogue scalability requirements to propose a solution that will have the potential to replace or significantly improve the existing namespace component of dCache, in particular:

- Estimate the typical workload on the metadata services and identify the scalability limitations of the existing solution.
- Identify workflows that require strict POSIX and near-POSIX compliance.
- Propose a design of a new metadata catalogue, that is at least a factor of 10 more scalable in the number of filesystem objects and overall throughput, i.e. operation per second, without compromising the filesystem integrity.

[3]Colossus, the successor to the Google File System might have a different architecture. The information about Colossus's design is publicly not available

Tough POSIX compliance is typically expected by the storage systems, in many cases, it is not required by all the applications. Even though there are some studies to understand the requirements of the scientific application, they are typically focused on the I/O path and pay very little attention to metadata-related operations[17]. Thus, a detailed analysis of scientific applications might provide opportunities to soften the POSIX compliance requirements and improve metadata scalability by implementing eventual consistency for some operations without compromising the filesystem integrity.

The R&D activity by the dCache team aims to utilize existing database technologies and widely available techniques to implement a new scalable metadata service for scientific data repositories.

References

[1] Amazon, Amazon S3, 2023. URL: https://docs.aws.amazon.com/s3/index.html.

[2] Ieee standard for information technology–portable operating system interface (posix(tm)) base specifications, issue 7, IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) (2018) 1–3951. doi:10.1109/IEEESTD.2018.8277153.

[3] Mkrtchyan, Tigran, Chitrapu, Krishnaveni, Garonne, Vincent, Litvintsev, Dmitry, Meyer, Svenja, Millar, Paul, Morschel, Lea, Rossi, Albert, Sahakyan, Marina, dcache: Inter-disciplinary storage system, EPJ Web Conf. 251 (2021) 02010. URL: https://doi.org/10.1051/epjconf/202125102010. doi:10.1051/epjconf/202125102010.

[4] A. S. Foundation, Apache zookeeper, 2023. URL: https://zookeeper.apache.org.

[5] P. Fuhrmann, A perfectly normal namespace for the desy open storage manager, 1997. URL: http://www-zeuthen.desy.de/CHEP97/paper/409.ps.

[6] M. Gasthuber, T. Mkrtchyan, P. Fuhrmann, Chimera - a new, fast, extensible and Grid enabled namespace service, in: 14th International Conference on Computing in High-Energy and Nuclear Physics, 2005, pp. 1180–1182. doi:10.5170/CERN-2005-002.1180.

[7] PostreSQL, PostreSQL, 2023. URL: https://www.postgresql.org/.

[8] T. H. D. Group, HSQLDB, 2023. URL: https://hsqldb.org/.

[9] M. Fowler, Strangler pattern, 2004. URL: https://martinfowler.com/bliki/StranglerFigApplication.html.

[10] T. Haerder, A. Reuter, Principles of transaction-oriented database recovery, ACM Comput. Surv. 15 (1983) 287–317. URL: https://doi.org/10.1145/289.291. doi:10.1145/289.291.

[11] S. R. Kleiman, Vnodes: An architecture for multiple file system types in sun unix, in: USENIX Summer, 1986. URL: https://api.semanticscholar.org/CorpusID:30546296.

[12] D. Noveck, M. Eisler, S. Shepler, Network File System (NFS) Version 4 Minor Version 1 Protocol, Number 5661 in Request for Comments, RFC Editor, 2010. URL: https://www.rfc-editor.org/info/rfc5661. doi:10.17487/RFC5661.

[13] CockroachLabs, CockroachDB, 2022. URL: https://github.com/cockroachdb/cockroach.

[14] A. Collaboration, ATLAS Software and Computing HL-LHC Roadmap, Technical Report, CERN, Geneva, 2022. URL: https://cds.cern.ch/record/2802918.

[15] S. Ghemawat, H. Gobioff, S.-T. Leung, The google file system, in: Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY, 2003, pp. 20–43.

[16] K. Ren, G. Gibson, TABLEFS: Enhancing metadata Efficiency in the local file system, in: 2013 USENIX Annual Technical Conference (USENIX ATC 13), USENIX Association, San Jose, CA, 2013, pp. 145–156. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/ren.

[17] C. Wang, K. Mohror, M. Snir, File system semantics requirements of hpc applications, in: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 19–30. URL: https://doi.org/10.1145/3431379.3460637. doi:10.1145/3431379.3460637.