Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

1

**Distributed Asynchronous Object Storage (DAOS)**

**Design Documentation**

**Extreme Storage Architecture and Development (ESAD)**

**INTEL FEDERAL, LLC PROPRIETARY**

**March 2019**

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

2

**NOTICES**

Generated under Argonne Contract number**: 8F-30005**

**Acknowledgment**: "This material is based upon work supported by the U.S. Department of Energy and Argonne National Laboratory and its Leadership Computing Facility, including under Contract DE-AC02-06CH11357 and Award Number 8F-30005. This work was generated with financial support from the U.S. Government through said Contract and Award Number(s), and as such the U.S. Government retains a paid-up, nonexclusive, irrevocable,world-wide license to reproduce, prepare derivative works, distribute copies to the public, and display publicly, by or on behalf of the Government, this work in whole or in part, or otherwise use the work for Federal purposes.”

**Disclaimer**: "This report was prepared as an account of work sponsored by an agency and/or National Laboratory of the United States Government. Neither the United States Government nor any agency or National Laboratory thereof,nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or

represents that its use would not infringe privately owned rights. Reference herein to any specific commercial

product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency or National Laboratory thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency or National Laboratory thereof."

Any agreement/reference numbers must include “Contract DE-AC02-06CH11357.”

**Intel Disclaimer:** Intel makes available this document and the information contained herein in furtherance of the CORAL A21 Program. None of the information contained herein is, or should be construed, as advice. While Intel makes every effort to present accurate and reliable information, Intel does not guarantee the accuracy,

completeness, efficacy, or timeliness of such information. Use of such information is voluntary, and reliance on it should only be undertaken after an independent review by qualified experts.

Access to this document is with the understanding that Intel is not engaged in rendering advice or other professional services. Information in this document may be changed or updated without notice by Intel.

This document contains copyright information, the terms of which must be observed and followed.

Reference herein to any specific commercial product, process or service does not constitute or imply endorsement,recommendation, or favoring by Intel or the US Government.

Intel makes no representations whatsoever about this document or the information contained herein. IN NO EVENT SHALL INTEL BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER

CONSEQUENTIAL DAMAGES FOR ANY USE OF THIS DOCUMENT, INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, BUSINESS INTERRUPTION, OR OTHERWISE, EVEN IF INTEL IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Company Name: Intel Federal LLC

Company Address: 4100 Monument Corner Drive, Suite 540 Fairfax, VA 22030

Copyright © 2019, Intel Corporation.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

3

**Document Revision History**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Revision Number** |  |  | **Date** |  |  | **Comments** |  |
| 0.5 | | | March 2019 | | | Initial version. | | |
| .65 | | | March 2019 | | | Document updated from new development and  feedback from Argonne | | |
| 1.0 | | | May 2019 | | | Document notices revised, first release of community  document | | |

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

4

**Contents**

1 DAOS Storage Model 9

1.1 Storage Architecture 9

1.1.1 DAOS Target 10

1.1.2 DAOS Pool 11

1.1.3 DAOS Container 12

1.1.4 DAOS Object 13

1.2 Transaction Model 14

1.2.1 Epoch & Timestamp Ordering 14

1.2.2 Container Snapshot 14

1.2.3 Distributed Transactions 15

1.3 Fault Model 16

1.3.1 Hierarchical Fault Domains 16

1.3.2 Fault Detection 16

1.3.3 Fault Isolation 17

1.3.4 Fault Recovery 17

1.4 Security Model 17

1.4.1 Authentication 17

1.4.2 Authorization 18

2 DAOS Internals 19

2.1 DAOS Components 19

2.1.1 DAOS System 20

2.1.2 Client APIs, Tools, and I/O Middleware 20

2.1.3 Agent 21

2.2 Network Transport and Communications 21

2.2.1 gRPC and Protocol Buffers 21

2.2.2 dRPC 21

2.2.3 CART 22

2.3 DAOS Layering and Services 22

2.3.1 Architecture 22

2.3.2 Source Code Structure 22

2.3.3 Infrastructure Libraries 23

2.3.4 DAOS Services 24

2.4 Software Compatibility 24

2.4.1 Protocol Compatibility 24

2.4.2 PM Schema Compatibility and Upgrade 25

3 DAOS Control Plane (aka daos\_server) 26

3.1 Documentation 26

3.2 Configuration 26

3.3 Subcommands 27

3.3.1 prep-nvme 27

3.3.2 show-storage 27

3.4 Shell Usage 27

3.5 NVMe management capabilities 28

3.5.1 NVMe Controller and Namespace Discovery 28

3.5.2 NVMe Controller Firmware Update 28

3.5.3 NVMe Controller Burn-in Validation 28

3.6 SCM management capabilities 28

3.6.1 SCM Module Discovery 28

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

5

3.6.2 SCM Module Firmware Update 28

3.6.3 SCM Module Burn-in Validation 28

3.7 Architecture 29

3.8 Development Requirements 29

3.9 Development setup 29

3.9.1 Building the app 29

3.9.2 Testing the app 29

3.10 Coding Guidelines 29

3.10.1 daos\_server and daos\_agent 29

4 Management Tool 30

5 DAOS Agent 31

6 Certificate Management 32

7 Security 33

8 DAOS Data Plane (daos\_io\_server) 34

8.1 Module Interface 34

8.2 Thread Model and Argobot Integration 34

8.3 Thread-local Storage (TLS) 34

8.4 Incast Variable Integration 34

8.5 dRPC Server 35

8.5.1 dRPC Progress 35

8.5.2 dRPC Handler Registration 35

9 Replicated Data Base (RDB) 36

9.1 Architecture 36

9.2 RPC Handling 37

10 Replicated Services 39

10.1 Introduction 39

10.1.1 Architecture 39

10.2 RSVC Module 40

11 DAOS Pool 41

11.1 Pool Service 41

11.1.1 Metadata Layout 41

11.2 Pool Operations 41

11.2.1 Pool / Pool Service Creation 41

12 DAOS Container 43

12.1 Metadata Layout 43

12.2 Target Service (TO BE UPDATED) 44

12.2.1 Target Faults 44

12.3 Container Operations 45

12.3.1 Epoch Protocol 45

13 Object 47

13.1 KV store, dkey and akey 47

13.2 Object Schema and Object Class 47

13.3 Data Protection Method 48

13.3.1 Replication 48

13.3.2 Erasure Code 49

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

6

13.4 Object Sharding 49

13.4.1 Single (unstriped) Object (DAOS\_OS\_SINGLE) 49

13.4.2 Fixed Stripe Object (DAOS\_OS\_STRIPED) 49

13.4.3 Dynamically Striped Object 49

14 Versioning Object Store 51

14.1 Persistent Memory based Storage 51

14.1.1 In-Memory Storage 51

14.1.2 Lightweight I/O Stack: PMDK Libraries 52

14.2 VOS Concepts 53

14.2.1 VOS Indexes 55

14.2.2 Object Listing 55

14.3 Key Value Stores (Single Value) 56

14.3.1 Operations Supported with Key Value Store 56

14.3.2 Key in VOS KV Stores 56

14.3.3 Internal Data Structures 57

14.4 Key Array Stores 60

14.5 Epoch Based Operations 62

14.5.1 VOS Discard 63

14.5.2 VOS Aggregate 63

14.6 VOS Checksum Management (TO BE UPDATED) 63

14.7 Metadata Overhead 64

14.8 Replica Consistency 64

14.8.1 DAOS Two-Phase Commit (DTX) 64

14.8.2 DTX Leader Election 66

15 Blob I/O 67

15.1 Storage Performance Development Kit (SPDK) 67

15.1.1 SPDK NVMe Driver 68

15.1.2 SPDK Block Device Layer (bdev) 68

15.1.3 SPDK Blobstore 68

15.2 SPDK Integration 68

15.3 Per-Server Metadata Management (SMD) 69

15.4 DMA Buffer Management 69

16 Algorithmic object placement 70

16.1 Placement Map 70

16.2 Ring Placement Map 71

16.2.1 Fault Tolerance 72

16.2.2 Failure Handling and Rebuild 72

16.2.3 Rebuild Targets 73

16.2.4 Rebalancing for Pool Extension 75

16.3 Multi-Ring Placement Map 77

16.3.1 Rebuild Declustering 79

16.3.2 Widely Striped Object 79

17 Self-healing (Rebuild) 81

17.1 Rebuild Detection 81

17.1.1 Current status and long-term goal 81

17.2 Rebuild process 81

17.2.1 Scan Phase 81

17.2.2 Pull Phase 81

17.2.3 Multiple pool and targets rebuild 82

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

7

17.3 I/O during the rebuild 84

17.4 Rebuild resource throttle 84

17.5 Rebuild status 84

17.6 Rebuild failure 85

18 DAOS Common Libraries 86

18.1 Task Scheduler Engine (TSE) 86

18.1.1 Scheduler API 86

18.1.2 Task API 86

18.2 dRPC C API 87

18.2.1 C Client 87

18.2.2 C Server 87

19 DAOS Client Library & I/O Middleware 89

19.1 DAOS Client Library 89

19.1.1 Event & Event Queue 89

19.1.2 Task Engine Integration 90

19.2 DAOS Addons 90

19.2.1 DAOS Arrays 90

19.2.2 DAOS High Level (HL) API 91

19.3 Python Bindings for DAOS API 91

19.3.1 The motivation for Python API 91

19.3.2 Architecture 92

19.3.3 Usage 94

19.4 Go language bindings for the DAOS API 96

19.4.1 Current Status 96

19.4.2 How to Build 96

19.4.3 dcmd utility 96

19.5 DFS Overview 97

19.5.1 DFS Namespace 97

19.5.2 DFS Directories 97

19.5.3 Files 98

19.5.4 Symbolic Links 99

19.5.5 Access Permissions 99

19.5.6 DFUSE 99

20 Terminology 100

**Tables**

Table 14-1. Example VOS KV Store input for Update/Punch 58

Table 14-2. Comparison of average case computational complexity for index 59

Table 14-3. Example Input for Extent Epoch Table 60

**Figures**

Figure 1-1. DAOS Storage Abstractions 10

Figure 1-2. DAOS Container Example 12

Figure 1-3. DAOS Object ID Structure 13

Figure 1-4. Example of container snapshots 15

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

8

Figure 1-5. Producer/Container workflow with DAOS Containers 15

Figure 2-1. DAOS SDS Components 19

Figure 2-2. DAOS Internal Services & Libraries 22

Figure 2-3. DAOS Internal Layering 24

Figure 3-1. Architecture diagram 29

Figure 9-1. Service replication modules 37

Figure 11-1. Pool Service Layout 41

Figure 12-1. Container Service Layout 43

Figure 13-1The object module implements the object I/O stack. 47

Figure 14-1. VOS pool storage layout 54

Figure 14-2. Red Black Tree based KV Store with Multi-Key 58

Figure 14-3. Example of extents and epochs in a Key Array object 60

Figure 14-4. Rectangles representing extent\_range.epoch\_validity arranged in 2-D space for

an order-4 EV-Tree using input in the table 61

Figure 14-5. Tree (order - 4) for the example in Table 6 3 (pictorial representation shown in

the figure above 62

Figure 14-6. Modify multiple replicated object under DTX model 66

Figure 15-1. SPDK Software Stack 68

Figure 16-1. Pool-map and placement maps 70

Figure 16-2. Ring Placement Map 72

Figure 16-3. Rebuild targets and failure sequence 73

Figure 16-4. Object distribution on rebuild targets 75

Figure 16-5. Doubling domains for a ring placement map 76

Figure 16-6. Doubling targets within domains 77

Figure 17-3. Multi-ring placement map 78

Figure 17-1. Rebuild Protocol 82

Figure 17-2. Multi-failure protocol 83

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

9

**1**  **DAOS Storage Model**

In this section, we describe the DAOS storage paradigm including its transaction, security and fault models. Acronyms are detailed in the terminology page.

This document contains the following sections:

• Storage Architecture

o DAOS Target

o DAOS Pool

o DAOS Container

o DAOS Object

• Transaction Model

o Epoch & Timestamp Ordering

o Container Snapshot

o Distributed Transactions

• Fault Model

o Hierarchical Fault Domains

o Fault Detection and Diagnosis

o Fault Isolation

o Fault Recovery

• Security Model

o Authentication

o Authorization

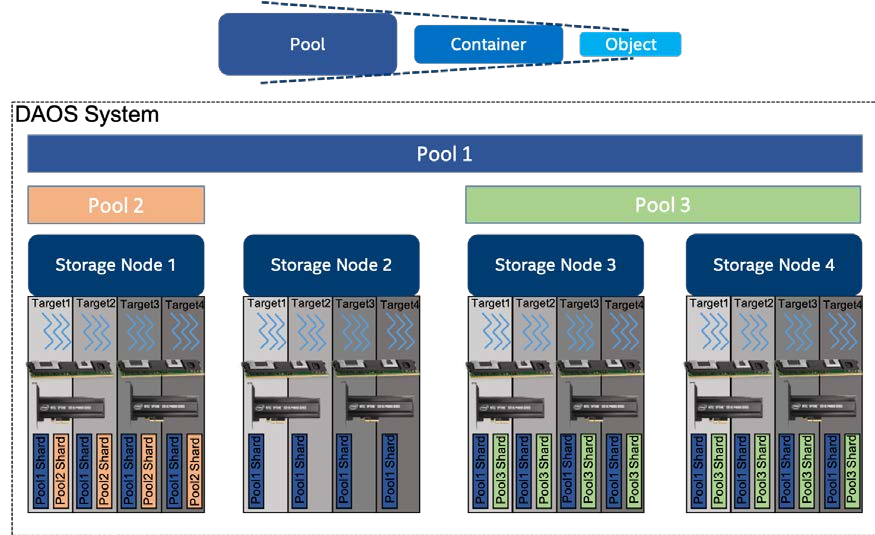
**1.1 Storage Architecture**

We consider a data center with hundreds of thousands of compute nodes interconnected via a scalable, high-performance fabric (i.e., Ethernet, RoCE or InfiniBand), where all or a subset of the nodes, called storage nodes, have direct access to byte-addressable storage-class memory (SCM) and, optionally, block-based NVMe storage. The DAOS server is a multi-tenant daemon running on a Linux instance (i.e., natively on the physical node or in a VM or container) of each storage node and exporting through the network the locally-attached storage. Inside a DAOS server, the storage is statically partitioned across multiple targets to optimize concurrency. To avoid contention, each target has its private storage,own pool of service threads and dedicated network context that can be directly addressed over the fabric independently of the other targets hosted on the same storage node. The number of targets exported by a DAOS server instance is configurable and depends on the underlying hardware (i.e., the number of SCM modules, CPUs, NVMe SSDs, and others). A target is the unit of fault. All DAOS servers connected to the same fabric are grouped to form a DAOS system, identified by a system name. Membership of the DAOS servers is recorded into the system map that assigns a unique integer rank to each server. Two different systems comprise two disjoint sets of servers and do not coordinate with each other.

The figure below represents the fundamental abstractions of the DAOS storage model.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

10

**Figure 1-1. DAOS Storage Abstractions**

A DAOS pool is a storage reservation distributed across a collection of targets. The actual space allocated to the pool on each target is called a pool shard. The total space allocated to a pool is decided at creation time and can be expanded over time by resizing all the pool shards (within the limit of the storage capacity dedicated to each target) or by spanning more targets (i.e., adding more pool shards). A pool offers storage virtualization and is the unit of provisioning and isolation. DAOS pools cannot span across multiple systems.

A pool can host multiple transactional object store called DAOS containers. Each container is a private object address space, which can be modified by transaction and independently of the other containers stored in the same pool. A container is the unit of snapshot and data management. DAOS objects belonging to a container can be distributed across any target of the pool for both performance and resilience and can be accessed through different APIs to represent structured, semi-structured and unstructured data efficiently.

The table below shows the targeted level of scalability for each DAOS concept.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **DAOS Concept** |  |  | **Order of Magnitude** |  |
| System | | | 105 Servers (hundreds of thousands) and 102 Pools (hundreds) | | |
| Server | | | 101 Targets (tens) | | |
| Pool | | | 102 Containers (hundreds) | | |
| Container | | | 109 Objects (billions) | | |

**1.1.1 DAOS Target**

A target is typically associated with a single-ported SCM module and NVMe SSD attached to a single storage node. Moreover, a target does not implement any internal data protection mechanism against storage media failure. As a result, a target is a single point of failure. A dynamic state is associated with each target and is set to either up and running, or down

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

11

and not available. A target is the unit of performance. Hardware components associated with the target, such as the backend storage medium, the server, and the network, have limited capability and capacity. Target performance parameters such as bandwidth and latency are exported to upper layers.

**1.1.2 DAOS Pool**

A pool is identified by a unique UUID and maintains target memberships in a persistent versioned list called the pool map. The membership is definitive and consistent, and

membership changes are sequentially numbered. The pool map not only records the list of active targets; it also contains the storage topology under the form of a tree that is used to identify targets sharing common hardware components. For instance, the first level of the tree can represent targets sharing the same motherboard; then the second level can represent all motherboards sharing the same rack and finally the third level can represent all racks in the same cage. This framework effectively represents hierarchical fault domains,which are then used to avoid placing redundant data on targets subject to correlated failures. At any point in time, new targets can be added to the pool map, and failed ones can be excluded. Moreover, the pool map is fully versioned, which effectively assigns a unique sequence to each modification of the map, more particularly for failed node removal.A pool shard is a reservation of persistent memory optionally combined with a pre-allocated space on NVMe storage on a specific target. It has a fixed capacity and fails operations when full. Current space usage can be queried at any time and reports the total amount of bytes used by any data type stored in the pool shard.

Upon target failure and exclusion from the pool map, data redundancy inside the pool is automatically restored online. This self-healing process is known as rebuild. Rebuild

progress is recorded regularly in special logs in the pool stored in persistent memory to address cascading failures. When new targets are added, data is automatically migrated to the newly added targets to redistribute space usage equally among all the members. This process is known as space rebalancing and uses dedicated persistent logs as well to support interruption and restart. A pool is a set of targets spread across different storage nodes over which data and metadata are distributed to achieve horizontal scalability, and replicated or erasure-coded to ensure durability and availability.

When creating a pool, a set of system properties must be defined to configure the different features supported by the pool. Also, the user can define their attributes that will be stored persistently.

A pool is only accessible to authenticated and authorized applications. Multiple security frameworks could be supported, from NFSv4 access control lists to third party-based

authentication (such as Kerberos). Security is enforced when connecting to the pool. Upon successful connection to the pool, a connection context is returned to the application

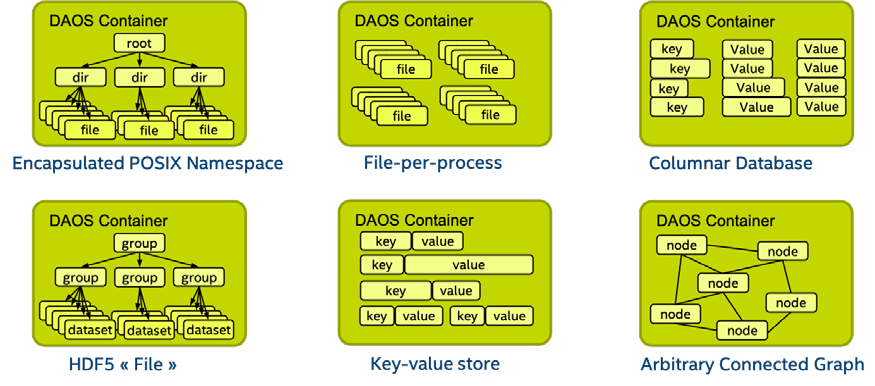
process.

As detailed previously, a pool stores many different sorts of persistent metadata, such as the pool map, authentication, and authorization information, user attributes, properties and rebuild logs. Such metadata are critical and require the highest level of resiliency.

Therefore, the pool metadata are replicated on a few nodes from distinct high-level fault domains. For very large configurations with hundreds of thousands of storage nodes, only a very small fraction of those nodes (in the order of tens) run the pool metadata service. With a limited number of storage nodes, DAOS can afford to rely on a consensus algorithm to reach agreement and to guarantee consistency in the presence of faults and to avoid split-brain syndrome.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

12

To access a pool, a user process should connect to this pool and pass the security checks.Once granted, a pool connection can be shared (via local2global() and global2local()operations) with any or all of its peer application processes (similar to the openg() POSIX extension). This collective connect mechanism allows avoiding metadata request storm when a massively distributed job run on the datacenter. A pool connection is then revoked when the original process that issued the connection request disconnects from the pool.

**1.1.3 DAOS Container**

A container represents an object address space inside a pool and is identified by a UUID.The diagram below represents how the user (i.e., I/O middleware, domain-specific data format, big data or AI frameworks ...) could use the container concept to store related datasets.

**Figure 1-2. DAOS Container Example**

Likewise, to pools, containers can store user attributes, and a set of properties must be passed at container creation time to configure different features like checksums.

An application must first connect to the pool and then open the container. If the application is authorized to access the container, a container handle is returned. This handle includes capabilities that authorize any process in the application to access the container and its contents. The opening process may share this handle with any or all of its peers. Their capabilities are revoked either on container close.

Objects in a container may have different schemas for data distribution and redundancy over targets. Dynamic or static striping, replication, or erasure code are some parameters required to define the object schema. The object class defines common schema attributes for a set of objects. Each object class is assigned a unique identifier and is associated with a given schema at the pool level. A new object class can be defined at any time with a configurable schema, which is then immutable after creation, or at least until all objects belonging to the class have been destroyed. For convenience, several object classes expected to be the most commonly used will be predefined by default when the pool is created, as shown in the table below.

Sample of Pre-defined Object Classes

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

13

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Object Class (RW = read/write, RM = read-**  **mostly** |  |  | **Redundancy** |  |  | **Layout (SC = stripe count, RC = replica**  **count, PC = parity count, TGT = target** |  |
| Small size & RW | | | Replication | | | static SCxRC, e.g. 1x4 | | |
| Small size & RM | | | Erasure code | | | static SC+PC, e.g. 4+2 | | |
| Large size & RW | | | Replication | | | static SCxRC over max #targets) | | |
| Large size & RM | | | Erasure code | | | static SCx(SC+PC) w/ max #TGT) | | |
| Unknown size & RW | | | Replication | | | SCxRC, e.g., 1x4 initially and grows | | |
| Unknown size & RM | | | Erasure code | | | SC+PC, e.g., 4+2 initially and grows | | |

As shown below, each object is identified in the container by a unique 128-bit object

address. The high 32 bits of the object address is reserved for DAOS to encode internal metadata such as the object class. The remaining 96 bits are managed by the user and should be unique inside the container. Those bits can be used by upper layers of the stack to encode their metadata as long as unicity is guaranteed. A per-container 64-bit scalable object ID allocator is provided in the DAOS API. The object ID to be stored by the

application is the full 128-bit address, which is for single use only and can be associated with only a single object schema.

<---------------------------------- 128 bits ---------------------------------->

**--------------------------------------------------------------------------------**

|DAOS Internal Bits| Unique User Bits |

**--------------------------------------------------------------------------------**

<---- 32 bits ----><------------------------- 96 bits ------------------------->

**Figure 1-3. DAOS Object ID Structure**

A container is the basic unit of transaction and versioning. All object operations are

implicitly tagged by the DAOS library with a timestamp called an epoch. The DAOS

transaction API allows combining multiple object updates into a single atomic transaction with multi-version concurrency control based on epoch ordering. All the versioned updates may periodically be aggregated to reclaim space utilized by overlapping writes and to reduce metadata complexity. A snapshot is a permanent reference that can be placed on a specific epoch to prevent aggregation.

Container metadata (i.e., a list of snapshots, open container handles, object class, user attributes, properties, and others) are stored in persistent memory and maintained by a dedicated container metadata service that either uses the same replicated engine as the parent metadata pool service or has its engine. This is configurable when creating a

container.

Likewise to a pool, access to a container is controlled by the container handle. An

application process must open the container and pass the security checks To acquire a valid handle. This container handle may then be shared with other peer application processes via the container local2global() and global2local() operations.

**1.1.4 DAOS Object**

To avoid scaling problems and overhead common to a traditional storage system, DAOS objects are intentionally simple. No default object metadata beyond the type and schema are provided. This means that the system does not maintain time, size, owner, permissions or even track openers. To achieve high availability and horizontal scalability, many object

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

14

schemas (replication/erasure code, static/dynamic striping, To acquire a valid handle, a) are provided. The schema framework is flexible and easily expandable to allow for new custom schema types in the future. The layout is generated algorithmically on the object open from the object identifier and the pool map. End-to-end integrity is assured by protecting object data with checksums during network transfer and storage.

A DAOS object can be accessed through different APIs:

• **Multi-level key-array API** is the native object interface with locality feature. The key

is split into a distribution (i.e., dkey) and an attribute (i.e., akey) keys. Both the dkey and akey can be of variables length and any types (i.e. a string, an integer or even a complex data structure). All entries under the same dkey are guaranteed to be

collocated on the same target. The value associated with akey can be either a single variable-length value that cannot be partially overwritten or an array of fixed-length values. Both the akeys and dkeys support enumeration.

• **Key-value API** provides a simple key and variable-length value interface. It supports

the traditional put, get, remove and list operations.

• **Array API** implements a one-dimensional array of fixed-size elements addressed by a

64-bit offset. A DAOS array supports arbitrary extent read, write and punch operations.

**1.2 Transaction Model**

The DAOS API supports distributed transactions that allow any update operations against objects belonging to the same container to be combined into a single ACID transaction.Distributed consistency is provided via a lockless optimistic concurrency control mechanism based on multi-version timestamp ordering. DAOS transactions are serializable and can be used on an ad-hoc basis for parts of the datasets that need it.

The DAOS versioning mechanism allows creating persistent container snapshots, which provide point-in-time distributed consistent views of a container, which can be used to build the producer-consumer pipeline.

**1.2.1 Epoch & Timestamp Ordering**

Each DAOS I/O operation is tagged with a timestamp called epoch. An epoch is a 64-bit integer that integrates both a logical and physical clock (see HLC paper). The DAOS API provides helper functions to convert an epoch to traditional POSIX time (i.e., struct

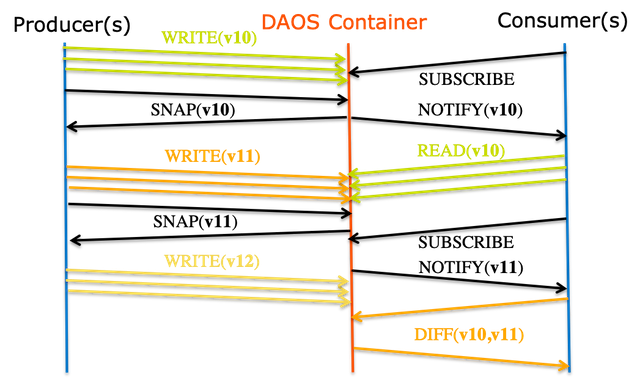
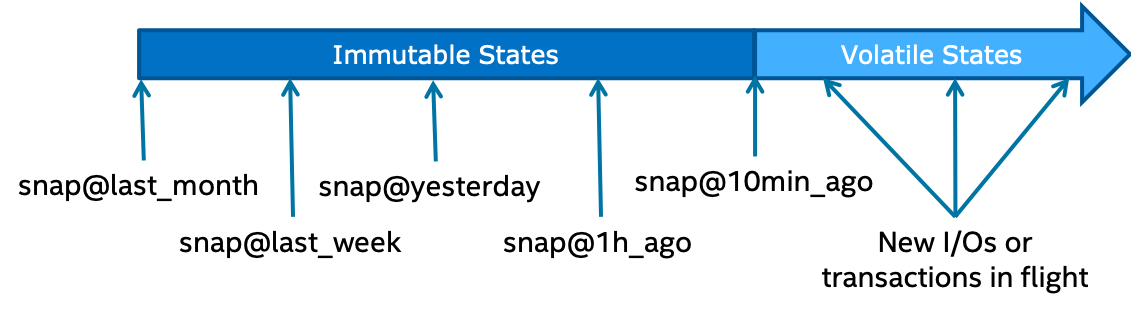
timespec, see clock\_gettime(3)).

**1.2.2 Container Snapshot**

As shown in the figure below, the content of a container can be snapshot at any time.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

15

**Figure 1-4. Example of container snapshots**

DAOS snapshots are very lightweight and are tagged with the epoch associated with the time when the snapshot was created. Once successfully created, a snapshot remains readable until it is explicitly destroyed. The content of a container can be rolled back to a particular snapshot.

The container snapshot feature allows supporting native producer/consumer pipeline as represented in the diagram below.

**Figure 1-5. Producer/Container workflow with DAOS Containers**

The producer will generate a snapshot once a consistent version of the dataset has been successfully written. The consumer applications may subscribe to container snapshot events so that new updates can be processed as the producer commits them. The immutability of the snapshots guarantees that the consumer sees consistent data, even while the producer continues with new updates. Both the producer and consumer indeed operate on different versions of the container and do not need any serialization. Once the producer generates a new version of the dataset, the consumer may query the differences between the two snapshots and process only the incremental changes.

**1.2.3 Distributed Transactions**

Unlike POSIX, the DAOS API does not impose any worst-case concurrency control

mechanism to address conflicting I/O operations. Instead, individual I/O operations are tagged with a different epoch and applied in epoch order, regardless of execution order.This baseline model delivers the maximum scalability and performance to data models and

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

16

applications that do not generate conflicting I/O workload. Typical examples are collective MPI-IO operations, POSIX file read/write or HDF5 dataset read/write.

For parts of the data model that require conflict serialization, DAOS provides distributed serializable transaction based on multi-version concurrency control. Transactions are typically needed when different user process can overwrite the value associated with a dkey/akey pair. Examples are a SQL database over DAOS or a consistent POSIX namespace accessed concurrently by uncoordinated clients. All I/O operations (include reads) submitted in the context of the same operation will use the same epoch. The DAOS transaction

mechanism automatically detects the traditional read/write, write/read and write/write conflicts and aborts one of the conflicting transactions that must be restarted by the user(i.e., transaction fails to commit with -DER\_RESTART).

In the initial implementation, the transaction API has the following limitations that will be addressed in future DAOS versions:

• No support for the array API

• Transactional object update and key-value put operations are not visible via object

fetch/list and key-value get/list operations executed in the context of the same

transaction.

**1.3 Fault Model**

DAOS relies on massively distributed single-ported storage. Each target is thus effectively a single point of failure. DAOS achieves availability and durability of both data and metadata by providing redundancy across targets in different fault domains. DAOS internal pool and container metadata are replicated via a robust consensus algorithm. DAOS objects are then safely replicated or erasure-coded by transparently leveraging the DAOS distributed

transaction mechanisms internally. The purpose of this section is to provide details on how DAOS achieves fault tolerance and guarantees object resilience.

**1.3.1 Hierarchical Fault Domains**

A fault domain is a set of servers sharing the same point of failure and which are thus likely to fail altogether. DAOS assumes that fault domains are hierarchical and do not overlap. The actual hierarchy and fault domain membership must be supplied by an external database used by DAOS to generate the pool map.

Pool metadata are replicated on several nodes from different high-level fault domains for high availability, whereas object data is replicated or erasure-coded over a variable number of fault domains depending on the selected object class.

**1.3.2 Fault Detection**

DAOS servers are monitored within a DAOS system through a gossip-based protocol called SWIM that provides accurate, efficient and scalable server fault detection. Storage attached to each DAOS target is monitored through periodic local health assessment. Whenever local storage I/O error is returned to the DAOS server, an internal health check procedure will be called automatically. This procedure will make an overall health assessment by analyzing the IO error code and device SMART/Health data. If the result is negative, the target will be marked as faulty, and further I/Os to this target will be rejected and re-routed.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

17

**1.3.3 Fault Isolation**

Once detected, the faulty target or servers (effectively a set of targets) must be excluded from the pool map. This process is triggered either manually by the administrator or

automatically. Upon exclusion, the new version of the pool map is eagerly pushed to all storage targets. At this point, the pool enters a degraded mode that might require extra processing on access (e.g., reconstructing data out of erasure code). Consequently, DAOS client and storage nodes retry RPC indefinitely until they find an alternative replacement target from the new pool map. At this point, all outstanding communications with the evicted target are aborted, and no further messages should be sent to the target until it is explicitly reintegrated (possibly only after maintenance action).

All storage targets are promptly notified of pool map changes by the pool service. This is not the case for client nodes, which are lazily informed of pool map invalidation each time they communicate with servers. To do so, clients pack in every RPC their current pool map version. Servers reply not only with the current pool map version. Consequently, when a DAOS client experiences RPC timeout, it regularly communicates with the other DAOS target to guarantee that its pool map is always current. Clients will then eventually be informed of the target exclusion and enter into degraded mode.

This mechanism guarantees global node eviction and that all nodes eventually share the same view of target aliveness.

**1.3.4 Fault Recovery**

Upon exclusion from the pool map, each target starts the rebuild process automatically to restore data redundancy. First, each target creates a list of local objects impacted by the target exclusion. This is done by scanning a local object table maintained by the underlying storage layer. Then for each impacted object, the location of the new object shard is

determined and redundancy of the object restored for the whole history (i.e., snapshots).Once all impacted objects have been rebuilt, the pool map is updated a second time to report the target as failed out. This marks the end of collective rebuild process and the exit from the degraded mode for this particular fault. At this point, the pool has fully recovered from the fault, and client nodes can now read from the rebuilt object shards.

This rebuild process is executed online while applications continue accessing and updating objects.

**1.4 Security Model**

DAOS uses a flexible security model that separates authentication from authorization. It is designed to have a very minimal impact on the I/O path.

**1.4.1 Authentication**

The DAOS security model is designed to support different authentication methods. By default, a local agent runs on the client node and authenticates the user process through AUTH\_SYS. Authentication can be handled by a third party service like munge or Kerberos.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

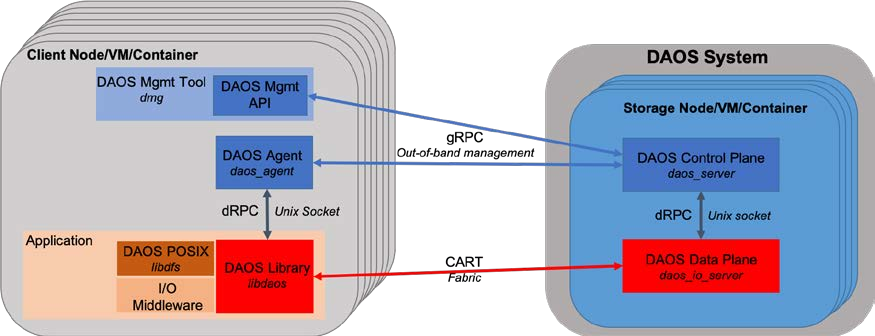
18

**1.4.2 Authorization**

DAOS supports a subset of the NFSv4 ACLs for both pools and containers through the properties API.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

19

**2**  **DAOS Internals**

The purpose of this document is to describe the internal code structure and major

algorithms used by DAOS. It assumes prior knowledge of the DAOS storage model and acronyms. This document contains the following sections:

• DAOS Components

o DAOS System

o Client APIs, Tools, and I/O Middleware

o Agent

• Network Transport and Communications

o gRPC and Protocol Buffers

o dRPC

o CART

• DAOS Layering and Services

o Architecture

o Code Structure

o Infrastructure Libraries

o DAOS Services

• Software compatibility

o Protocol Compatibility

o PM Schema Compatibility and Upgrade

**2.1 DAOS Components**

As illustrated in the diagram below, a DAOS installation involves several components that can be either collocated or distributed. The DAOS software-defined storage (SDS)

framework relies on two different communication channels: an out-of-band TCP/IP network for management and a high-performant fabric for data access. In practice, the same

network can be used for both management and data access. IP over fabric can also be used as the management network.

**Figure 2-1. DAOS SDS Components**

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

20

**2.1.1 DAOS System**

A DAOS server is a multi-tenant daemon running on a Linux instance (i.e., physical node,VM or container) and managing the locally-attached NVM storage allocated to DAOS. It listens to a management port, addressed by an IP address and a TCP port number, plus one or more fabric endpoints, addressed by network URIs. The DAOS server is configured through a YAML file (i.e.,/etc/daos\_server.yml or different path provided on the command line) and can be integrated with different daemon management or orchestration frameworks(e.g., a systemd script, a Kunernetes service or even via a parallel launcher like pdsh or srun).

A DAOS system is identified by a system name and consists of a set of DAOS servers connected to the same fabric. Two different systems comprise two disjoint sets of servers and do not coordinate with each other. DAOS pools cannot span across multiple systems.Internally, a DAOS server is composed of multiple daemon processes. The first one to be started is the control plane (binary named daos\_server for convenience) which is

responsible for parsing the configuration file, provisioning storage and eventually starting and monitoring one or multiple instances of the data plane (i.e., daos\_io\_server binary).The control plane is written in Go and implements the DAOS management API over the gRPC framework that provides a secured out-of-band channel to administrate a DAOS system. The number of data plane instances to be started by each server, as well as the storage, CPU and fabric interface affinity, can be configured through the YAML configuration file.

The data plane is a multi-threaded process written in C that runs the DAOS storage engine.It processes incoming metadata and I/O requests though the CART communication

middleware and accesses local NVM storage via the PMDK (for storage-class memory, aka SCM) and SPDK (for NVMe SSDs) libraries. The data plane relies on Argobots for event-based parallel processing and exports multiple targets that can be independently addressed via the fabric. Each data plane instance is assigned a unique rank inside a DAOS system.The control plane and data plane processes communicate locally through Unix Domain Sockets and a custom lightweight protocol called dRPC.

For further reading:

• DAOS control plane (daos\_server)

• DAOS data plane (daos\_io\_server)

**2.1.2 Client APIs, Tools, and I/O Middleware**

Applications, users, and administrators can interact with a DAOS system through two different client APIs.

The DAOS management Go package allows administrating a DAOS system from any nodes that can communicate with the DAOS servers through the out-of-band management

channel. This API is reserved for the DAOS system administrators who are authenticated through a specific certificate. The DAOS management API is intended to be integrated with different vendor-specific storage management or open-source orchestration frameworks. A CLI tool called dmg is built over the DAOS management API. For further reading on the management API and the dmg tool:

• DAOS management Go package

• DAOS Management tool (aka dmg)

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

21

The DAOS library (i.e., libdaos) implements the DAOS storage model and is primarily targeted at the application and I/O middleware developers who want to store datasets into DAOS containers. It can be used from any nodes connected to the fabric used by the targeted DAOS system. The application process is authenticated via the DAOS agent (see next section). The API exported by libdaos is commonly called the DAOS API (in opposition to the DAOS management API) and allows to manage containers and access DAOS objects through different interfaces (e.g., key-value store or array API). The libdfs library emulates POSIX file and directory abstractions over libdaos and provides a smooth migration path for applications that require a POSIX namespace. For further reading on libdaos, bindings for different programming languages and libdfs: -

• DAOS Library (libdaos) and array/kv interface built on top of the native DAOS API –

• Python API bindings

• Go bindings and API documentation

• POSIX File & Directory Emulation (libdfs)

The libdaos and libdfs libraries provide the foundation to support domain-specific data formats like HDF5 and Apache Arrow. For further reading on I/O middleware integration,please check the following external references:

• DAOS VOL connector for HDF5

• ROMIO DAOS ADIO driver for MPI-IO

**2.1.3 Agent**

The DAOS agent is a daemon residing on the client node and interacts with the DAOS client library through dRPC to authenticate the application process. It is a trusted entity that can sign the DAOS Client credentials using local certificates. The agent can support different authentication frameworks and uses a Unix Domain Socket to communicate with the client library. The DAOS agent is written in Go and communicates through gRPC with the control plane component of each DAOS server to provide DAOS system membership information to the client library and to support pool listing.

**2.2 Network Transport and Communications**

As introduced in the previous section, DAOS uses three different communication channels.

**2.2.1 gRPC and Protocol Buffers**

gRPC provides a bi-directional secured channel for DAOS management. It relies on TLS/SSL to authenticate the administrator role and the servers. Protocol buffers are used for RPC serialization, and all proto files are located in the proto directory.

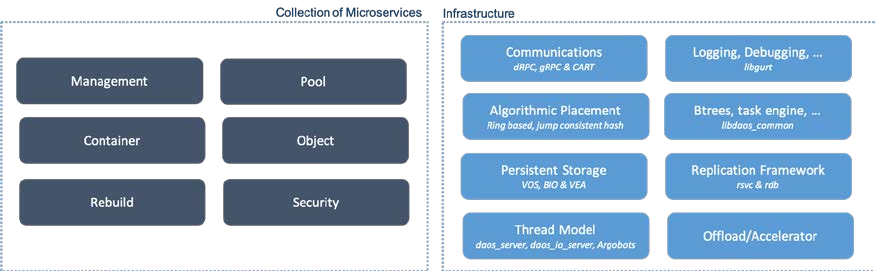
**2.2.2 dRPC**

dRPC is communication channel built over Unix Domain Socket that is used for inter-process communications. It provides both a C and Go interface to support interactions between:

• the agent and libdaos for application process authentication

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

22

• the daos\_server (control plane) and the daos\_io\_server (data plane) daemons Like

gRPC, RPC are serialized via protocol buffers.

**2.2.3 CART**

CART is a userspace function shipping library that provides low-latency high-bandwidth communications for the DAOS data plane. It supports RDMA capabilities and scalable collective operations. CART is built over Mercury and libfabric. The CART library is used for all communications between libdaos and daos\_io\_server instances.

**2.3 DAOS Layering and Services**

**2.3.1 Architecture**

As shown in the diagram below, the DAOS stack is structured as a collection of storage services over a client/server architecture. Examples of DAOS services are the pool,

container, object and rebuild services.

**Figure 2-2. DAOS Internal Services & Libraries**

A DAOS service can be spread across the control and data planes and communicate

internally through dRPC. Most services have client and server components that can

synchronize through gRPC or CART. Cross-service communications are always done through direct API calls. Those function calls can be invoked across either the client or server component of the services. While each DAOS service is designed to be fairly autonomous and isolated, some are more tightly coupled than others. That is typically the case of the rebuild service that needs to interact closely with the pool, container and object services to restore data redundancy after a DAOS server failure.

While the service-based architecture offers flexibility and extensibility, it is combined with a set of infrastructure libraries that provide a rich software ecosystem (e.g., communications,persistent storage access, asynchronous task execution with dependency graph, accelerator support, others) accessible to all the DAOS services.

**2.3.2 Source Code Structure**

Each infrastructure library and service is allocated a dedicated directory under src/. The client and server components of a service are stored in separate files. Functions that are

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

23

part of the client component are prefixed with dc\_ (stands for DAOS Client) whereas server-side functions use the ds\_ prefix (stands for DAOS Server). The protocol and RPC format used between the client and server components is usually defined in a header file named rpc.h.

All the Go code executed in the context of the control plane is located under src/control.Management and security are the services spread across the control (Go language) and data(C language) planes and communicating internally through dRPC.

Headers for the official DAOS API exposed to the end user (i.e., I/O middleware or

application developers) are under src/include and use the daos\_ prefix. Each infrastructure library exports an API that is available under src/include/daos and can be used by any services. The client-side API (with dc\_ prefix) exported by a given service is also stored under src/include/daos whereas the server-side interfaces (with ds\_ prefix) are under src/include/daos\_srv.

**2.3.3 Infrastructure Libraries**

The GURT and common DAOS (i.e., libdaos\_common) libraries provide logging, debugging and common data structures (e.g., hash table, btree, ...) to the DAOS services.

Local NVM storage is managed by the Versioning Object Store (VOS) and blob I/O (BIO)libraries. VOS implements the persistent index in SCM whereas BIO is responsible for storing application data in either NVMe SSD or SCM depending on the allocation strategy.The VEA layer is integrated into VOS and manages block allocation on NVMe SSDs.

DAOS objects are distributed across multiple targets for both performance (i.e., sharding)and resilience (i.e., replication or erasure code). The placement library implements different algorithms (e.g., ring-based placement, jump consistent hash, and others) to generate the layout of an object from the list of targets and the object identifier.

The replicated service (RSVC) library finally provides some common code to support fault tolerance. This library is used by the pool, container & management services in conjunction with the RDB library that implements a replicated key-value store over Raft.

For further reading on those infrastructure libraries, please see:

Common Library

Versioning Object Store (VOS)

Blob I/O (BIO)

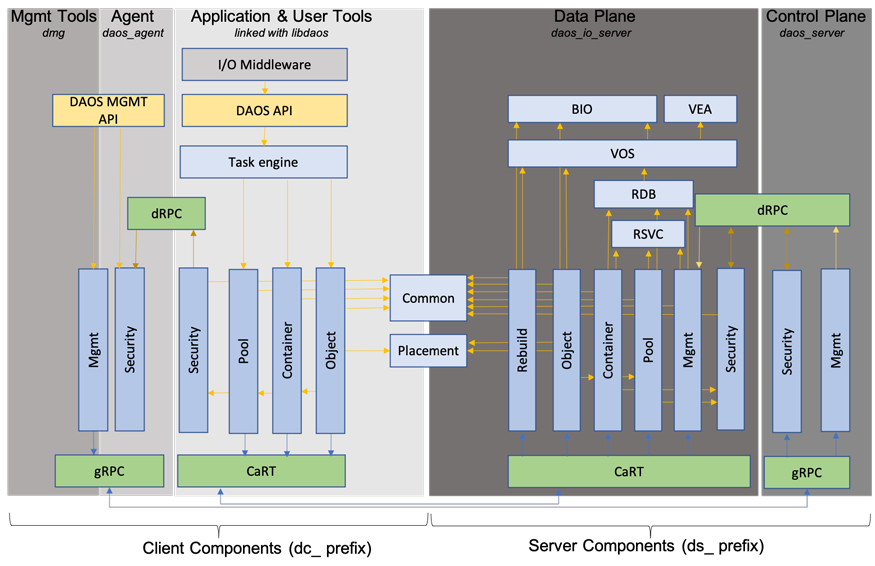
Algorithmic object placement

Replicated database (RDB)

Replicated service framework (RSVC)

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

24

**2.3.4 DAOS Services**

The diagram below shows the internal layering of the DAOS services and interactions with the different libraries mentioned above.

**Figure 2-3. DAOS Internal Layering**

Vertical boxes represent DAOS services whereas horizontal ones are for infrastructure libraries.

For further reading on the internals of each service:

• Pool service

• Container service

• Key-array object service

• Self-healing (aka rebuild)

• Security

**2.4 Software Compatibility**

Interoperability in DAOS is handled via protocol and schema versioning for persistent data structures.

**2.4.1 Protocol Compatibility**

Limited protocol interoperability is to be provided by the DAOS storage stack. Version compatibility checks will be performed to verify that:

• All targets in the same pool run the same protocol version.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

25

• Client libraries linked with the application may be up to one protocol version older than

the targets.

If a protocol version mismatch is detected among storage targets in the same pool, the entire DAOS system will fail to start up and will report failure to the control API. Similarly,the connection from clients running a protocol version incompatible with the targets will return an error.

**2.4.2 PM Schema Compatibility and Upgrade**

The schema of persistent data structures may evolve from time to time to fix bugs, add new optimizations or support new features. To that end, the persistent data structures support schema versioning.

Upgrading the schema version is not done automatically and must be initiated by the administrator. A dedicated upgrade tool will be provided to upgrade the schema version to the latest one. All targets in the same pool must have the same schema version. Version checks are performed at system initialization time to enforce this constraint.

To limit the validation matrix, each new DAOS release will be published with a list of

supported schema versions. To run with the new DAOS release, administrators will then need to upgrade the DAOS system to one of the supported schema version. The new target will always be reformatted with the latest version. This versioning schema only applies to a data structure stored in persistent memory and not to block storage that only stores user data with no metadata.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

26

**3**  **DAOS Control Plane (aka daos\_server)**

DAOS operates over two, closely integrated planes, Control and Data. The Data plane handles the heavy lifting transport operations while the Control plane orchestrates process and storage management, facilitating the operation of the Data plane.

DAOS Server implements the DAOS Control Plane and is written in Golang. It is tasked with network and storage hardware provisioning and allocation in addition to instantiation and management of the DAOS IO Servers (Data Plane wrote in C) running on the same host.Users of DAOS will interact directly only with the Control Plane in the form of the DAOS Server and associated tools.

The DAOS Server implements the gRPC protocol to communicate with client gRPC

applications and interacts with DAOS IO Servers through Unix domain sockets.

Multiple gRPC server modules are loaded by the control server. Currently included modules are security and management.

The Control Plane implements a replicated management service as part of the DAOS Server,responsible for handling distributed operations across the DAOS System.

The shell is an example client application which can connect to both the agent to perform security functions (such as providing credentials and retrieving security contexts) and to the local management server to perform management functions (such as storage device discovery).

**3.1 Documentation**

• Management API

• Management Internals

• Agent API

• Agent internals

• dRPC

**3.2 Configuration**

The daos\_server config file is parsed when starting the daos\_server process; it's location can be specified on the command line (-o option) or default location (<daos install

dir>/install/etc/daos\_server.yml).

Example config files can be found in the examples folder.

Some parameters will be parsed and populated with defaults as documented in the default daos server config if not present in the config file.

Parameters passed to daos\_server on the command line as application options (excluding environment variables) take precedence over values specified in the config file.

For convenience, active parsed config values are written to the directory where the server config file was read from or /tmp/ if that fails.

If user shell executing daos\_server has environment variable CRT\_PHY\_ADDR\_STR set, user os environment will be used when spawning daos\_io\_server instances. In this situation, an error message beginning "using os env vars..." will be printed and no environment variables will be added as specified in the env\_vars list within the per-server section of the server

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

27

config file. This behavior provides backward compatibility with the historic mechanism of specifying all parameters through environment variables.

It is strongly recommended to specify all parameters and environment for running DAOS servers in the server config file.

To clarify concerning environment variables affecting the behavior of daos\_io\_server instances:

• If the trigger environment variable is set in the user's shell, the control plane will use

the environment variables set in the shell. The config file will be ignored.

• If the trigger environment variable is NOT set in the user's shell, the shell environment

variables will be overridden by the parameters set in the config file.

**3.3 Subcommands**

daos\_server supports various subcommands (see daos\_server --help for available

subcommands) which will perform stand-alone tasks as opposed to launching as a daemon(default operation if launched without subcommand).

**3.3.1 prep-nvme**

This subcommand requires elevated permissions and needs to be run with root permissions(sudo).

NVMe access through SPDK as an unprivileged user can be enabled by first running sudo daos\_server prep-nvme -p 4096 -u bob. This will perform the required setup for daos\_server to be run by user "bob" who will own the hugepage mountpoint directory and vfio groups as needed in SPDK operations. If the target-user is unspecified (-u short option), the target user will be the issuer of the sudo command (or root if not using sudo). The specification of hugepages (-p short option) defines the number of huge pages to allocate for use by SPDK.The configuration commands that require elevated permissions are in

src/control/mgmt/init/setup\_spdk.sh (script is installed as install/share/setup\_spdk.sh).The sudoers file can be accessed with command visudo and permissions can be granted to a user to execute a specific command pattern (requires prior knowledge of daos\_server binary location):

linuxuser ALL=/home/linuxuser/projects/daos\_m/install/bin/daos\_server prep-nvme\*

See daos\_server prep-nvme --help for usage.

**3.3.2 show-storage**

List NVMe SSDs and SCM modules locally attached to the host.

See daos\_server show-storage --help for usage.

**3.4 Shell Usage**

To run the shell to perform administrative tasks, build and run the daos\_server as per the quickstart guide.

daos\_server is to be run as root to perform administrative tasks, to be run through orterun as root:

root$ orterun -np 1 -c 1 --hostfile hostfile --enable-recovery --allow-run-as-root --report-uri /tmp/urifile daos\_server -c 1

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

28

daos\_shell (the management tool to exercise the client API) is to be run on login nodes by an unprivileged user (and is designed to be lightweight without dependencies on storage libraries). The shell can be used to connect to and interact with multiple gRPC servers concurrently (running on port 10000 by default) as follows:

$ projects/daos\_m/install/bin/daos\_shell -l foo-45:10001,foo-44:10001

DAOS Management Shell

>>>

See daos\_shell --help for usage.

**3.5 NVMe management capabilities**

Operations on NVMe SSD devices are performed using go-spdk bindings to issue commands over the SPDK framework.

**3.5.1 NVMe Controller and Namespace Discovery**

The following animation is available and illustrates starting the control server and using the management shell to view the NVMe Namespaces discovered on a locally available NVMe Controller (assuming the quickstart\_guide instructions have already been performed):

Demo: List NVMe Controllers and Namespaces

**3.5.2 NVMe Controller Firmware Update**

The following animation is available and illustrates starting the control server and using the management shell to update the firmware on a locally available NVMe Controller (assuming the quickstart\_guide instructions have already been performed):

Demo: Updating NVMe Controller Firmware

**3.5.3 NVMe Controller Burn-in Validation**

Burn-in validation is performed using the fio tool which executes workloads over the SPDK framework using the fio\_plugin.

**3.6 SCM management capabilities**

This section to be updated in a future revision.

**3.6.1 SCM Module Discovery**

This section to be updated in a future revision.

**3.6.2 SCM Module Firmware Update**

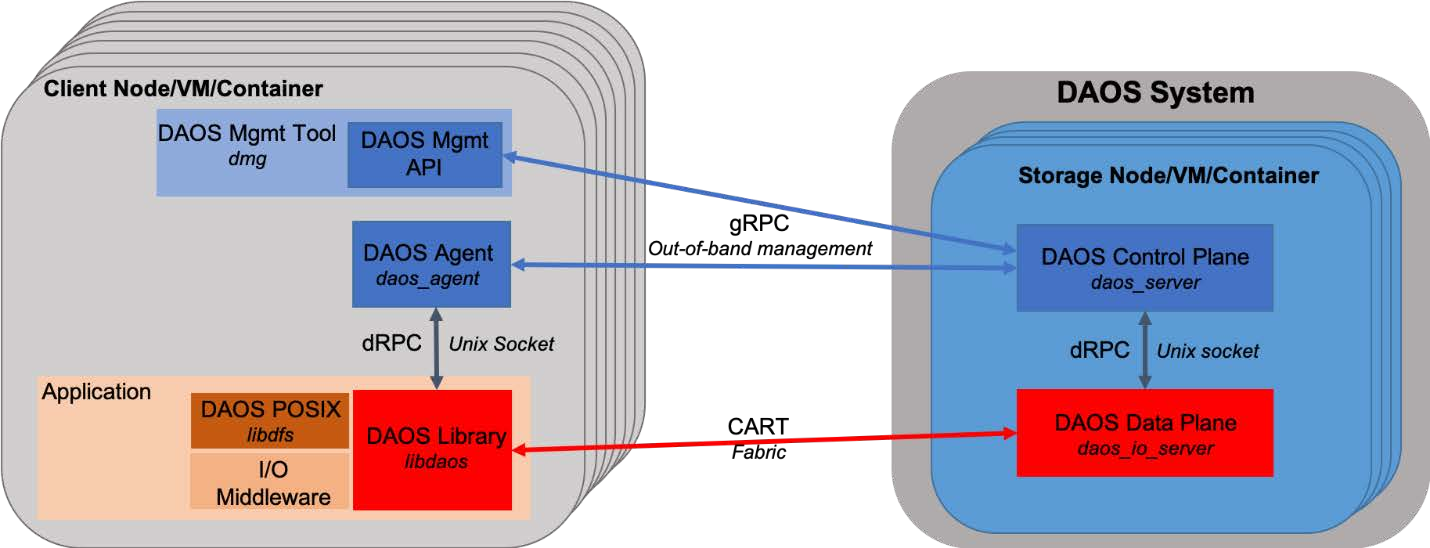
This section to be updated in a future revision.

**3.6.3 SCM Module Burn-in Validation**

This section to be updated in a future revision.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

29

**3.7 Architecture**

First a view of software component architecture:

**Figure 3-1. Architecture diagram**

**3.8 Development Requirements**

• Golang 1.9 or higher

• gRPC

• Protocol Buffers

• Dep for managing dependencies in the vendor directory.

**3.9 Development setup**

• If changing vendor package versions, edit src/control/Gopkg.toml and then run dep

ensure from src/control.

• (Optional) protoc protocol buffer compiler

**3.9.1 Building the app**

**3.9.1.1 Local**

• scons (binaries should be produced in install/bin directory)

**3.9.2 Testing the app**

• Run the tests go test within each directory containing tests

**3.10 Coding Guidelines**

**3.10.1 daos\_server and daos\_agent**

• Avoid calling os.Exit (or function with equivalent effects), except for assertion purposes.

Fatal errors shall be returned back to main, who calls os.Exit accordingly.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

30

**4**  **Management Tool**

This section will be updated in a future revision.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

31

**5**  **DAOS Agent**

This section will be updated in a future revision.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

32

**6**  **Certificate Management**

This information will be updated in a future revision.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

33

**7**  **Security**

This information will be updated in a future revision.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

34

**8**  **DAOS Data Plane (daos\_io\_server)**

**8.1 Module Interface**

The I/O server supports a module interface that allows loading server-side code on demand.Each module is effectively a library dynamically loaded by the I/O server via dlopen. The interface between the module and the I/O server is defined in the dss\_module data

structure.

Each module should specify:

• a module name

• a module identifier from daos\_module\_id

• a feature bitmask

• module initialization and finalize function

Also, a module can optionally configure:

• a setup and cleanup function invoked once the overall stack is up and running

• CART RPC handlers

• dRPC handlers

**8.2 Thread Model and Argobot Integration**

The I/O server is a multi-threaded process using Argobots for non-blocking processing.By default, one main xstream plus two offload xstreams are created per target. The actual number of offload xstream can be configured through daos\_io\_server command line

parameters. Moreover, an extra xtream is created to handle incoming metadata requests.Each xstream is bound to a specific CPU core. The main xstream is the one receiving

incoming target requests from both client and the other servers. A specific ULT is started to make progress on network and NVMe I/O operations.

**8.3 Thread-local Storage (TLS)**

Each xstream allocates private storage that can be accessed via the dss\_tls\_get() function.When registering, each module can specify a module key with a size of data structure that will be allocated by each xstream in the TLS. The dss\_module\_key\_get() function will return this data structure for a specific registered module key.

**8.4 Incast Variable Integration**

DAOS uses IV (incast variable) to share values and statuses among servers under a single IV namespace, which is organized as a tree. The tree root is called IV leader, and servers can either be leaves or non-leaves. Each server maintains its IV cache. During fetch, if the local cache can not fulfill the request, it forwards the request to its parents, until reaching the root (IV leader). As for the update, it updates its local cache first, then forwards to its parents until it reaches the root, which then propagates the changes to all the other

servers. The IV namespace is per pool, which is created during pool connection, and

destroyed during pool disconnection. To use IV, each user needs to register itself under the IV namespace to get an identification; then it will use this ID to fetch or update its IV value under the IV namespace.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

35

**8.5 dRPC Server**

The I/O server includes a dRPC server that listens for activity on a given Unix Domain Socket. See the dRPC documentation for more details on the basics of dRPC, and the low-level APIs in Golang and C.

The dRPC server polls periodically for incoming client connections and requests. It can handle multiple simultaneous client connections via the struct drpc\_progress\_context object, which manages the struct drpc objects for the listening socket as well as any active client connections.

The server loop runs in its User-Level Thread (ULT) in xstream 0. The dRPC socket has been set up as non-blocking and polling uses timeouts of 0, which allows the server to run in a ULT rather than its xstream. This channel is expected to be relatively low-traffic.

**8.5.1 dRPC Progress**

drpc\_progress represents one iteration of the dRPC server loop. The workflow is as follows:1. Poll with a timeout on the listening socket and any open client connections

simultaneously.

2. If any activity is seen on a client connection:

a. If data has come in: Call drpc\_recv to process the incoming data.

b. If the client has disconnected or the connection has been broken: Free the struct

drpc object and remove it from the drpc\_progress\_context.

3. If any activity is seen on the listener:

a. If a new connection has come in: Call drpc\_accept and add the new struct drpc

object to the client connection list in the drpc\_progress\_context.

b. If there was an error: Return -DER\_UNKNOWN to the caller, this causes an error to be

logged in the I/O server but does not interrupt the dRPC server loop. Getting an error on the listener is unexpected.

4. If no activity was seen, return -DER\_TIMEDOUT to the caller. This is purely for debugging

purposes. In practice, the I/O server ignores this error code since lack of activity is not an error case.

**8.5.2 dRPC Handler Registration**

Individual DAOS modules may implement handling for dRPC messages by registering a handler function for one or more dRPC module IDs.

Registering handlers is simple. In the dss\_server\_module field sm\_drpc\_handlers, statically allocate an array of struct dss\_drpc\_handler with the last item in the array zeroed out to indicate the end of the list. Setting the field to NULL indicates nothing to register. When the I/O server loads the DAOS module, it will register all of the dRPC handlers automatically.**Note:** The dRPC module ID is **not** the same as the DAOS module ID. This is because a given DAOS module may need to register more than one dRPC module ID, depending on the functionality it covers. The dRPC module IDs must be unique system-wide and are listed in a central header file: src/include/daos/drpc\_modules.h

The dRPC server uses the function drpc\_hdlr\_process\_msg to handle incoming messages.This function checks the incoming message's module ID, searches for a handler, executes the handler if one is found, and returns the Drpc\_\_Response. If none is found, it generates its Drpc\_\_Response indicating the module ID was not registered.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

36

**9**  **Replicated Data Base (RDB)**

Pool, container, and management services are made highly available by replicating their internal metadata using Raft-based consensus and strong leadership. A service replicated in this generic approach tolerates the failure of any minority of its replicas. By spreading

replicas of each service across the fault domains, pool and container services can, therefore,tolerate a reasonable number of target failures.

**9.1 Architecture**

A replicated service is built around a Raft replicated log. The service transforms RPCs into state queries and deterministic state updates. All state updates are committed to the replicated log first, before being applied by any of the service replicas. Since Raft

guarantees consistency among log replicas, the service replicas end up applying the same set of state updates in the same order and go through identical state histories.

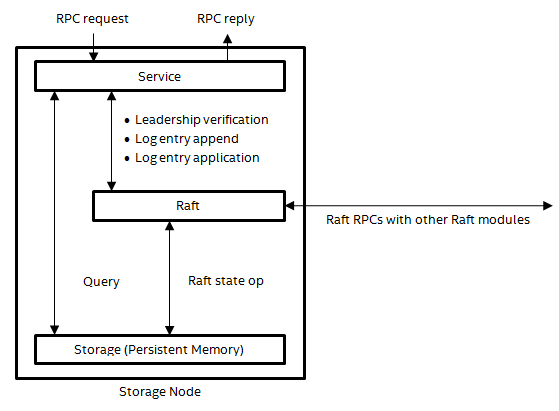
Among all replicas of a replicated service, only the current leader can handle service RPCs.The leader of a service is the current Raft leader (i.e., a Raft leader with the highest term number at the moment). Non-leader replicas reject all service RPCs and try to redirect the clients to the current leader to the best of their knowledge. Clients cache the addresses of the replicas as well as who current leader is. Occasionally, a client may not get any

meaningful redirection hints and can find current leader by communicating to a random replica.

The figure below shows the modules constituting a service replica. The service module handles RPCs by transforming them into state queries and deterministic state updates. The Raft module implements the replicated log following the Raft protocol, by communicating with Raft modules on other replicas. It provides methods for the service module to perform the queries and updates. The storage module, which in this case is the persistent memory and the file system, stores the service and Raft state. It uses VOS to update the state stored in persistent memory atomically.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

37

**Figure 9-1. Service replication modules**

**9.2 RPC Handling**

When an RPC request arrives at the leader, a service thread of the service module picks up the request and handles it by executing a handler function designed for this type of request.As far as service replication is concerned, a handler comprises state queries (e.g., reading the epoch state), state updates (writing a new version of the pool map), and RPCs to other services (e.g., TARGET\_CONTAINER\_OPEN RPCs sent to target services). Some handlers involve only queries, some involve updates as well as queries, and others involve all three kinds of actions; rarely, if ever, do handlers involve only updates but no queries.

A handler must assemble all its updates into a single log entry, commit the log entry, and wait for the log entry to become applicable before applying the updates to the service state.Using a single log entry per update RPC easily makes each update RPC atomic with regard to leader crashes and leadership changes. If RPCs that cannot satisfy this requirement is introduced in the future, additional transaction recovery mechanisms will be required. A leader’s service state therefore always represents the effects of all completed update RPCs this leader has handled so far.

Queries, on the other hand, can read directly from the service state, without going through the replicated log. However, to make sure a request sees the effects of all completed update RPCs handled by all leaders ever elected; the handler must ask the Raft module whether there have been any leadership changes. If there has been none, all queries made for this request so far are not stale. If the leader has lost its leadership, the handler aborts the request with an error redirecting the client to the new leader.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

38

RPCs to other services, if they update the state of destination services, must be idempotent.In case of a leadership change, the new leader may send them again, if the client re-sent the service request in question.

Handlers need to cope with reasonable concurrent executions. Conventional local locking on the leader is sufficient to make RPC executions linear. Once a leadership change happens,the old leader can no longer perform any updates or leadership verifications without noticing the leadership change, which causes all RPCs in execution to abort. The RPCs on the new leader is thus not in conflict with those still left on the old leader. The locks, therefore, do not need to be replicated as part of the service state.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

39

**10 Replicated Services**

The RSVC module implements a generic replicated service framework. This section covers service replication in general, before focusing specifically on the RSVC module.

**10.1 Introduction**

Certain DAOS RPC services, such as the Management Service (mgmt\_svc), Pool Service(pool\_svc), and Container Service (cont\_svc), are replicated using the state machine approach with Raft. Each of these services tolerates the failure of any minority of its

replicas. By spreading its replicas across different fault domains, the service can be highly available. Since this replication approach is self-contained in the sense that it requires only local persistent storage and point to point unreliable messaging, but not any external configuration management service, these services are mandatory for bootstrapping DAOS systems as well as managing the configuration of the lighter-weight I/O replication protocol.

**10.1.1 Architecture**

An RPC service handles incoming service requests (or just requests) based on its current service state (or just state). To replicate a service is therefore to replicate its state, so that each request is handled based on the state reached through all prior requests.

The state of a service is replicated with a Raft log. The service transforms requests into state queries and deterministic state updates. All state updates are committed to the Raft log first, before being applied to the state. Since Raft guarantees the consistency among the log replicas, the service replicas end up applying the same set of state updates in the same order and go through identical state histories.

Raft adopts a strong leadership design, so does each replicated service. A service leader of a term is the Raft leader of the same Raft term. Among the replicas of a service, only the leader of the highest term can handle requests. For the server side, the service code is similar to that of a non-replicated RPC service, except for the handling of leadership change events. For the client side, the service requests must be sent to the current leader, which must be searched for if not known already.

A replicated service is implemented using a stack of modules:

[ mgmt\_svc, pool\_svc, ... ]

[ ds\_rsvc ]

[ rdb ]

[ raft ]

[ vos ]

mgmt\_svc, pool\_svc, and cont\_svc implement the request handlers and the leadership change event handlers of the respective services. They define their respective service state in terms of the RDB data model provided by rdb, implement state queries and updates using RDB transactions, and register their leadership change event handlers into the framework rsvc offers.

RDB (daos\_srv/rdb) implements a hierarchical key-value store data model with

transactions, replicated using Raft. It delivers Raft leadership change events to ds\_rsvc,implements transactions using the Raft log, and store its data model as well as internal metadata using the VOS data model.

raft (rdb/raft/include/raft.h) implements the Raft core protocol in a library. Its integration with VOS and CaRT is done inside rdb.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

40

A replicated service client (e.g., dc\_pool) uses dc\_rsvc to search for the current service leader:

[ dc\_pool ]

[ dc\_rsvc ]

**10.2 RSVC Module**

The main purpose of RSVC is to avoid code duplication among different replicated service implementations. The callback-intensive API follows from the attempt to extract as much common code as possible, even at the expense of API simplicity. This is a key difference from how other module APIs are designed.

RSVC has two parts:

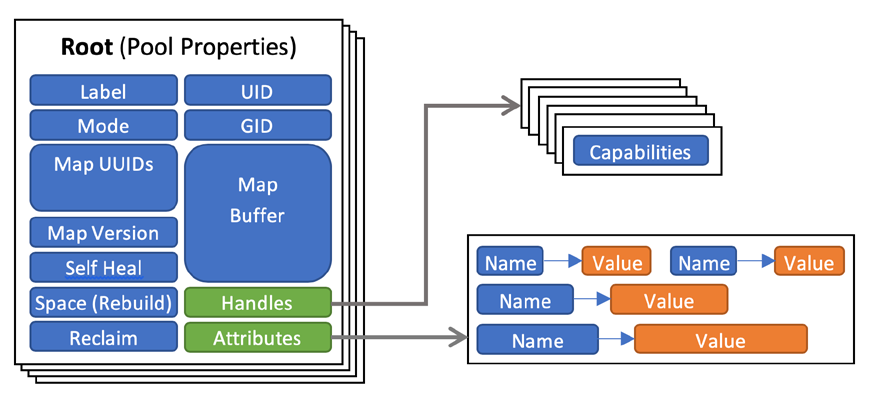
• ds\_rsvc (daos\_srv/rsvc.h): server-side framework.

• dc\_rsvc (daos/rsvc.h): client-side library.

dc\_rsvc is currently still called rsvc. A rename will be done in a future patch.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

41

**11 DAOS Pool**

A pool is a set of targets spread across different storage nodes over which data and

metadata are distributed to achieve horizontal scalability and replicated or erasure-coded to ensure durability and availability (see: Storage Model: DAOS Pool).

**11.1 Pool Service**

The Pool Service (pool\_svc) stores the metadata for pools and provides an API to query and update the pool configuration. Pool metadata are organized as a hierarchy of key-value stores (KVS) that is replicated over a number of servers backed by Raft consensus protocol,which uses strong leadership; client requests can only be serviced by the service leader while non-leader replicas merely respond with a hint pointing to the current leader for the client to retry.

**11.1.1 Metadata Layout**

**Figure 11-1. Pool Service Layout**

The top-level KVS stores the pool map, security attributes such as the UID, GID and mode,information related to space management and self-healing (see: Rebuild) as well as a second-level KVS containing user-defined attributes (see: Container Service: Metadata).Also, it also stores information on pool connections, represented by a pool handle and

identified by a client-generated handle UUID. The terms "pool connection" and "pool handle"may be used interchangeably.

**11.2 Pool Operations**

**11.2.1 Pool / Pool Service Creation**

Pool creation is driven entirely by the Management Service since it requires special

privileges for steps related to allocation of storage and querying of fault domains. After formatting all the targets, the Target component calls ds\_pool\_create of the Pool Module on each target, which simply generates a new UUID for the current target and stores it in the DSM\_META\_FILE. At this point, the management module passes the control to the pool module by calling theds\_pool\_svc\_create, which initializes service replication on the selected subset of nodes for the combined Pool and Container Service. The Pool module now sends a

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

42

POOL\_CREATE request to the service leader, which creates the service database; the list of targets and their fault domains are then converted into the initial version of the pool map and stored in the pool service, along with other initial pool metadata.

**11.2.1.1 Pool Connection**

To establish a pool connection, a client process calls the daos\_pool\_connect method in the client library with the pool UUID, connection information (such as group name and list of service ranks) and connection flags; this initiates a POOL\_CONNECT request to the Pool

Service. The Pool Service tries to authenticate the request according to the security model in use (e.g., UID/GID in a POSIX-like model), and to authorize the requested capabilities to the client-generated pool handle UUID. Before proceeding, the pool map is transferred to the client; if there are errors from this point onwards, the server can simply ask the client to discard the pool map.

At this point, the Pool Service checks for existing pool handles:

• If a pool handle with the same UUID already exists, a pool connection has already been

established and nothing else needs to be done.

• If another pool handle exists such that either the currently requested or the existing one

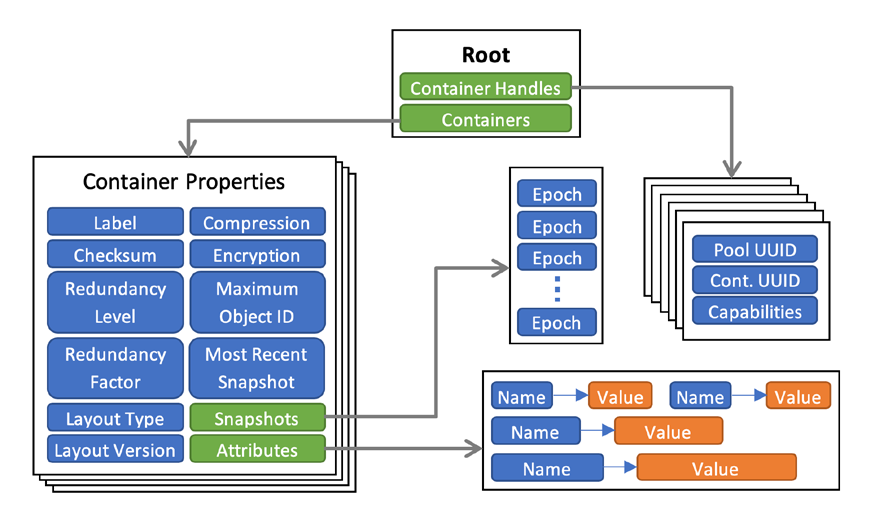
has exclusive access, the connection request is rejected with a busy status code. At this point,

If everything goes well, the pool service sends a collective POOL\_TGT\_CONNECT request to all targets in the pool with the pool handle UUID. The Target Service creates and caches the local pool objects and opens the local VOS pool for access.

To close a pool connection, a client process calls the daos\_pool\_disconnect method in the client library with the pool handle, triggering a POOL\_DISCONNECT request to the Pool Service,which sends a collective POOL\_TGT\_DISCONNECT request to all targets in the pool. These steps destroy all state associated with the connection, including all container handles. Other client processes sharing this connection should destroy their copies of the pool handle locally,preferably before the disconnect method is called on behalf of everyone. If a group of client processes terminates prematurely, before having a chance to call the pool disconnect method, their pool connection will eventually be evicted once the pool service learns about the event from the run-time environment.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

43

**12 DAOS Container**

A container represents an object address space inside a pool and is identified by a UUID. To access a container, an application must first connect to the pool and then create or open the container. If the application is authorized to access the container, it obtains a container handle. This includes capabilities that authorize any process in the application to access the container and its contents. The opening process may share this handle with any or all of its peers. Their capabilities are revoked on closing the container.

**12.1 Metadata Layout**

The Container Service (cont\_svc) stores the metadata for containers and provides an API to query and update the state as well as for managing the life-cycle of a container. Container metadata are organized as a hierarchy of key-value stores (KVS) that is replicated over a number of servers backed by Raft consensus protocol, which uses strong leadership; the service leader can only service client requests while non-leader replicas merely respond with a hint pointing to the current leader for the client to retry.

**Figure 12-1. Container Service Layout**

The top-level KVS root has two children:

1. **Containers KVS:** Holds a list of Container Properties KVSs indexed by UUID of the

Container which is supplied by the user at the time of creating a new container.

2. **Container Handles KVS:** Used for storing data about container handles opened by

various applications and indexed by a handle UUID that is generated by the client at the time of opening a container. The metadata associated with a container handle includes its capabilities (e.g., read-only or read-write) and its per-handle epoch state. When a container is closed, the corresponding entry is removed from this store.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

44

The container properties KVS is used to store per-container metadata that consists of many mutable and immutable scalar valued properties as well as other KVSs as shows in the figure above.

Users can create, delete and retrieve a list persistent snapshots, which are essentially epochs that will not be aggregated away. A snapshot remains readable until it is explicitly destroyed. A container can also be rolled back to a particular snapshot. (see: Storage Model: Snapshots)

Users can also define custom attributes for containers, which are essentially name-value pairs, with the name being a null-terminated string while the value is an arbitrary sequence of bytes. The Container Service allows clients to retrieve and update multiple attributes at a time as well as to list names of stored attributes.

**12.2 Target Service (TO BE UPDATED)**

The Target Service maps the global object address space of a DAOS container onto the local object address space of a VOS container within the target's VOS pool (vpool) and calls VOS methods on behalf of the Container Service (see: VOS Concepts). It caches per-thread information on container objects and open handles in volatile memory for ready access.

**12.2.1 Target Faults**

Given hundreds of thousands of targets, the epoch protocol must allow progress in the presence of target faults. Since pool and container services are highly available, the

problem is mainly concerned with target services. The solution is based on the assumption that losing some targets may not necessarily cause any application data loss, as there may be enough redundancy created by the DAOS-SR layer to hide the faults from applications.Moreover, an application might even want to ignore a particular data loss (which the DAOS-SR layer is unable to hide), for it has enough application-level redundancy to cope or it simply does not care.

When a write, flush or discard operation fails, the DAOS-SR layer calculates if there is sufficient redundancy left to continue with the epoch. If the failure can be hidden and

assuming that the target in question has not already been disabled in the pool map (e.g., as a result of a RAS notification), the DAOS-SR layer must disable the target before committing the epoch. For the epoch protocol, the resulting pool map update effectively records the fact that the target may store an undefined set of write operations in the epoch, and should be avoided. This also applies to applications that would like to ignore similar failures, which the DAOS-SR layer cannot hide.

**12.2.1.1 Object ID Allocator**

The OID allocator is a helper routine service that allows users to allocate a unique set of 64-bit unsigned integers within a container. This is helpful for applications or middleware that do not have a way to easily allocate a unique DAOS object ID in a scalable manner. The largest allocated ID is tracked in the Container Properties KVS for future access to that container. This service does not guarantee that the IDs allocated are sequential and several ID ranges may be discarded at container close.

The allocator is implemented using an Incast Variable on the server side that tracks the highest used object ID on a container on the root of the IV tree. A client may request a new

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

45

allocation from any server running the Container Target Service, i.e., any node in the IV tree. When a new request arrives, the server first checks whether there are any allocated IDs available locally. If not, it forwards a request to the parent (asking for a bigger range of OIDs in that case). The parent does the same check and keeps forwarding to its parent until a request is satisfied or we reach the IV root, which updates the incast variable for the max OID allocated in the container metadata. At each tree level, the number of OIDs asked for is increased to be able to satisfy future OID allocation requests faster.

**12.3 Container Operations**

A client creates a new container by sending a CONT\_CREATE request to the Container Service with the pool handle and a UUID. The client must first establish a pool connection to obtain a pool handle. Optionally, the request can also contain a list of properties to be set on the newly created container. In response, the Container Service creates the corresponding Container Properties KVS with the UUID as the key. Creating a container does not require the involvement of the Target Service.

Clients may now open a container by supplying the open pool handle and the container UUID along with flags (e.g., read-only or read-write). The client library sends a CONT\_OPEN request with a locally generated UUID to the Container Service, which translates it into a collective CONT\_TGT\_OPEN request and broadcasts it to all enabled targets in the pool. On successful completion, it creates a new entry in the Container Handles KVS.

A client can close a container handle that is no longer needed by sending a CONT\_CLOSE request to the Container Service, which it broadcasts, to all enabled targets as a collective CONT\_TGT\_CLOSE to close the container handle. It then deletes the corresponding entry from the Container Handles KVS and discards updates performed on the handle that was not committed.

A container is destroyed when the client sends a CONT\_DESTROY request to the Container Service causing it to purge all metadata. Similarly, the targets collectively receive a

CONT\_TGT\_DESTROY request from the Container Service and drop all data associated with that container including all the objects within that container. The client can optionally destroy a container forcibly in case it has handles that are currently open.

**12.3.1 Epoch Protocol**

The epoch protocol implements the epoch model described in the Transactional Model. The Container service manages the epochs of a container; it maintains the definitive epoch state as part of the container metadata, whereas the target services have little knowledge of the global epoch state. Epoch commit, discard, and aggregate procedures are therefore all driven by the container service.

On each target, the target service eagerly stores incoming write operations into the

matching VOS container. If a container handle discards an epoch, VOS helps discard all write operations associated with that container handle. When a write operation succeeds, it is immediately visible to conflicting operations in equal or higher epochs. A conflicting write operation with the same epoch will be rejected by VOS unless it is associated with the same container handle and has the same content as the one that is already executed.

Before committing an epoch, an application must ensure that a sufficient set of write operations for this epoch have been persisted by the target services. The application may decide that losing some write operations is acceptable, depending on the redundancy

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

46

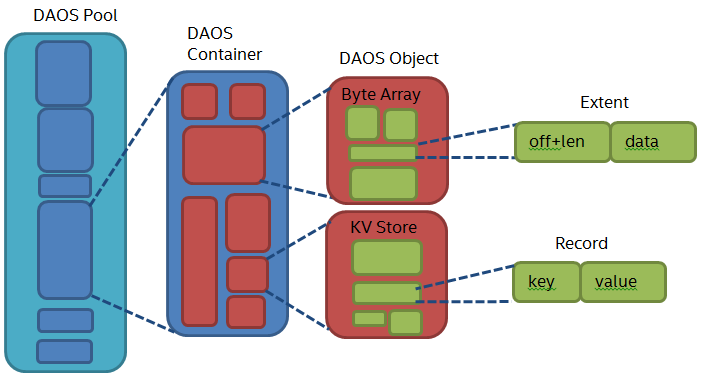
scheme each of them employs. Committing an epoch of a container handle results in a

CONT\_EPOCH\_COMMIT request to the corresponding container service, which simply updates the metadata. When the update becomes persistent, the container service replies to the client with the new epoch state.

Discarding epoch results in a CONT\_EPOCH\_DISCARD request to the corresponding container service, which in turn triggers a collective CONT\_TGT\_EPOCH\_DISCARD request to all target services. Once the discard method succeeds, all write operations from this container handle in this epoch are discarded.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

47

**13 Object**

DAOS object stores user's data; it is identified by object ID that is unique within the DAOS container it belongs to. Objects can be distributed across any target of the pool for both performance and resilience. DAOS object in DAOS storage model is shown in the diagram below.

**Figure 13-1The object module implements the object I/O stack.**

**13.1 KV store, dkey and akey**

Each DAOS object is a Key-Value store with locality feature. The key is split into a dkey(distribution key) and an akey (attribute key). All entries with the same dkey are

guaranteed to be collocated on the same targets. Enumeration of the akeys of a dkey is provided.

The value can be either atomic single value (i.e., value replaced on update) or a byte array(i.e., arbitrary extent fetch/update).

**13.2 Object Schema and Object Class**

To avoid scaling problems and overhead common to traditional storage stack, DAOS objects are intentionally very simple. No default object metadata beyond the type and schema(object class ownership) are provided. This means that the system does not maintain time,size, owner, permissions and opener tracking attributes.

The DAOS object schema describes the definitions for object types, data protection

methods, and data distribution strategies. An object class has a unique class ID, which is a16-bit value and can represent a category of objects that use the same schema and schema attributes. DAOS provides some pre-defined object class for the most common use (see daos\_obj\_classes). In addition user can register customized object class by

daos\_obj\_register\_class() (not implemented yet). A successfully registered object class is stored as container metadata; it is valid in the lifetime of the container.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

48

The object class ID is embedded in object ID. By daos\_obj\_generate\_id() the user can generate an object ID for the specific object class ID. DAOS uses this class ID to find the corresponding object class, and then distribute and protect object data based on algorithm descriptions of this class.

**13.3 Data Protection Method**

Two types of data protection methods supported by DAOS - replication and erasure coding.

**13.3.1 Replication**

Replication ensures high availability of object data because objects are accessible while any replica exists. Replication can also increase read bandwidth by allowing concurrent reads from different replicas.

**13.3.1.1 Client-side Replication**

Client replication is the mode that it is synchronous and fully in the client stack, to provide high concurrency and low latency I/O for the upper layer. - I/O requests against replicas are directly issued via DAOS client; there is no sequential guarantee on writes in the same epoch, and concurrent writes for the same object can arrive at different replicas in an arbitrary order.

• Because there is no communication between servers in this way, there is no consistent

guarantee if there are overlapped writes or KV updates in the same epoch（不同客户端之间可能用同一个epoch更新同一段数据？）. The DAOS server should detect overlapped updates in the same epoch, and return errors or warnings for the updates to the client. The only exception is multiple updates to the same extent or KV having the exact same data. In this case, it is allowed because these updates could potentially be the resending requests.

--- 不同epoch的writes 但是到Server端时乱序的 分片间一致性怎么保证

--- client分发时的epoch是在Server端确定的还是client指定的？

Furthermore, when a failure occurs, the client replication protocol still needs some extra mechanism to enforce consistency between replicas:

• If the DAOS client can capture the failure, e.g., a target failed during I/O, because the

DAOS client can complete this I/O by switching to another target and resubmitting the request. At the meanwhile, the DAOS servers can rebuild the missing replica in the background（client分发 重构前dtx resync？）. Therefore, DAOS can still guarantee data consistency between replicas. This process is transparent to the DAOS user. In the implementation, the IO completion callback (obj\_comp\_cb) will check the IO's completion status and will retry the IO when needed:

o If any shard's IO completed with a retryable error (stale pool map, timed out, or

other CaRT/HG level network error) then will refresh the pool map and retry the IO.

o If all shards' IO succeed but partial shards' replied pool map version is newer than

others, and the target location changed for any shard between old and new pool map version, then will refresh client-side pool map and retry those shards' IO with old pool map version（为什么要用老的pool map重试？）.

--- IO流程中pool map版本的检查机制？

o For read-only operation (fetch or enumerate) if the IO succeed but replied pool map

version is newer then needs to refresh the client cached pool map and needs not to retry the IO.

--- 所有分片上的数据一致性 是怎么保证的？ 对应的场景？

• If DAOS cannot capture the failure, for example, the DAOS client itself crashed before

successfully updating all replicas so some replicas may lag. Based on the current

replication protocol, the DAOS servers cannot detect the missing I/O requests, so DAOS cannot guarantee data consistency between replicas. The upper layer stack has to either re-submit the interrupted I/O requests to enforce data consistency between replicas or abort the epoch and rollback to the consistent status of the container.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

49

**13.3.1.2 Server-side Replication**

DAOS also supports server replication, which has stronger consistency of replicas with a trade-off in performance and latency. In server replication mode DAOS client selects a leader shard to send the IO request with the need-to- forward shards embedded in the RPC request when the leader shard gets that IO request it handles it as below steps:

• Firstly forwards the IO request to others shards For the request forwarding, it is

offloaded to the vos target's offload xtream to release the main IO service xtream from IO request sending and reply receiving (see shard\_req\_forward).

• Then serves the IO request locally

• Waits for the forwarded IO's completion and reply client IO request.

In server replication mode, the DAOS client-side IO error handling is relatively simpler because all operations only sent to only one server shard target, so need not to compare replied pool map version from multiple shard targets, another error handling is same as client replication mode described above.

In this mode, the conflict writes can be detected and serialized by the leader shard server. Now both modes are supported by DAOS, it can be dynamically configured by environment variable DAOS\_IO\_SRV\_DISPATCH before loading DAOS server. By default, DAOS works in server replication mode, and if the ENV set as zero then will work in client replication mode.

**13.3.2 Erasure Code**

In the case of replicating a whole object, the storage overhead would be 100% for each replica. This is unaffordable in some cases, so DAOS also provides erasure code as another option of data protection, with better storage efficiency.

Erasure codes may be used to improve resilience, with lower space overhead. This feature is still working in progress.

**13.4 Object Sharding**

DAOS supports different data distribution strategies.

**13.4.1 Single (unstriped) Object (DAOS\_OS\_SINGLE)**

Single (unstriped) objects always have one stripe, and each shard of it is a full replica, they can generate the localities of replicas by the placement algorithm. A single (unstriped)object can be either a byte-array or a KV.

**13.4.2 Fixed Stripe Object (DAOS\_OS\_STRIPED)**

A fixed stripe object has a constant number of stripes, and each stripe has a fixed stripe size. Upper levels provide values for these attributes when initializing the stripe schema,and then DAOS uses these attributes to compute object layout.

**13.4.3 Dynamically Striped Object**

A fixed stripe object always has the same number of stripes since it was created. In

contrast, a dynamically stripped object could be created with a single stripe. It will increase its stripe count as its size grows to some boundary, to achieve more storage space and better concurrent I/O performance.

Now the dynamically Striped Object schema defined in DAOS (DAOS\_OS\_DYN\_STRIPED

/DAOS\_OS\_DYN\_CHUNKED) but not implemented yet.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

50

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

51

**14 Versioning Object Store**

The Versioning Object Store (VOS) is responsible for providing and maintaining a persistent object store that supports byte-granular access and versioning for a single shard in a DAOS pool. It maintains its own metadata in persistent memory and may store data either in persistent memory or on block storage, depending on available storage and performance requirements. It must provide this functionality with minimum overhead so that

performance can approach the theoretical performance of the underlying hardware as closely as possible, both concerning latency and bandwidth. Its internal data structures, in both persistent and non-persistent memory, must also support the highest levels of concurrency so that throughput scales over the cores of modern processor architectures.Finally, and critically, it must validate the integrity of all persisted object data to eliminate the possibility of silent data corruption, both in normal operation and under all possible recoverable failures.

This section provides the details for achieving the design above goals in building a

versioning object store for DAOS.

This document contains the following sections:

• Persistent Memory based Storage

o In-Memory Storage

o Lightweight I/O Stack: PMDK Libraries

• VOS Concepts

o VOS Indexes

o Object Listing

• Key Value Stores

o Operations Supported with Key Value Store

o Key in VOS KV Stores

o Internal Data Structures

• Key Array Stores

• Epoch Based Operations

o VOS Discard

o VOS Aggregate

• VOS Checksum Management

• Metadata Overhead

• Replica Consistency

o DAOS Two-phase Commit

o DTX Leader Election

**14.1 Persistent Memory based Storage**

**14.1.1 In-Memory Storage**

The VOS is designed to use a persistent memory storage model that takes advantage of byte-granular, sub-microsecond storage access possible with new NVRAM technology. This enables a disruptive change in performance compared to conventional storage systems for application and system metadata, and small, fragmented and misaligned I/O. Direct access

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

52

to byte-addressable low-latency storage opens up new horizons where metadata can be scanned in less than a second without bothering with seek time and alignment.

The VOS relies on a log-based architecture using persistent memory primarily to maintain internal persistent metadata indexes. The actual data can be stored either in persistent memory directly or in block-based NVMe storage. The DAOS service has two tiers of

storage: Storage Class Memory (SCM) for byte-granular application data and metadata, and NVMe for bulk application data. Similar to how PMDK is currently used to facilitate access to SCM, the Storage Performance Development Kit (SPDK) is used to provide seamless and efficient access to NVMe SSDs.

The current DAOS storage model involves three DAOS server xstreams per core, along with one main DAOS server xstream per core mapped to an NVMe SSD device.

每个core上有3个xstreams

enum dss\_xs\_type // xstreams的类型

{

/\*\* current xstream \*/

DSS\_XS\_SELF = -1,

/\*\* operations need to access VOS \*/

DSS\_XS\_VOS = 0,

/\*\* forward/dispatch IO request for TX coordinator \*/

DSS\_XS\_IOFW = 1,

/\*\* EC/checksum/compress computing offload \*/

DSS\_XS\_OFFLOAD = 2,

/\*\* pool service, RDB, drpc handler \*/

DSS\_XS\_SYS = 3,

/\*\* drpc listener \*/

DSS\_XS\_DRPC = 4,

};

/\*\* Number of dRPC xstreams \*/

#define DRPC\_XS\_NR (1)

/\*\* Number of offload XS \*/

unsigned int dss\_tgt\_offload\_xs\_nr;

/\*\* Number of target (XS set) per engine \*/

unsigned int dss\_tgt\_nr;

/\*\* Number of system XS \*/

unsigned int dss\_sys\_xs\_nr = DAOS\_TGT0\_OFFSET + DRPC\_XS\_NR;

DAOS storage allocations can occur on either SCM by using a PMDK pmemobj pool, or on NVMe, using an SPDK blob.

数据通过PMDK写到SCM 或者SPDK写到NVMe SSD

All local server metadata will be stored in a per-server pmemobj pool on SCM and will include all current and relevant NVMe device, pool,and xstream mapping information.

所有的本地Server元数据存放在每个Server的pmemobj pool，包含当前和相关的NVMe SSD、pool和xstream mapping信息。

Please refer to the Blob I/O (BIO) module for more information regarding NVMe, SPDK, and per-server metadata. Special care is taken when developing and modifying the VOS layer because any software bug could corrupt data structures in persistent memory. The VOS, therefore, checksums its persistent data

structures despite the presence of hardware ECC.

The VOS provides a lightweight I/O stack fully in user space, leveraging the PMDK open source libraries developed to support this programming model.

VOS是全用户态的，PMDK也要做到。

**14.1.2 Lightweight I/O Stack: PMDK Libraries**

Although persistent memory is accessible via direct load/store,

PMEM直接使用load/store访问，代码里怎么体现的？

updates go through multiple levels of caches including the processor L1/2/3 caches and the NVRAM controller. Durability is guaranteed only after all those caches have been explicitly flushed. The VOS maintains internal data structures in persistent memory that must retain some level of consistency so that operation may be resumed without loss of durable data after an unexpected crash or power outage. The processing of a request will typically result in several memory allocations and updates that must be applied atomically.

Consequently, a transactional interface must be implemented on top of persistent memory to guarantee internal VOS consistency. It is worth noting that such transactions are

different from the DAOS transaction mechanism. Persistent memory transactions must guarantee consistency of VOS internal data structures when processing incoming requests,regardless of their epoch number. Transactions over persistent memory can be

implemented in many different ways, e.g., undo logs, redo logs, a combination of both, or copy-on-write.

PDMK提供事务接口，保证在处理请求时vos内部数据的一致性，具体可以采用undo-log 、redo-log、cow等多种方式。

PMDK is an open source collection of libraries for using persistent memory, optimized specifically for NVRAM. Among these is the libpmemobj library, which implements

relocatable persistent heaps called persistent memory pools. This includes memory

allocation, transactions, and general facilities for persistent memory programming.

The transactions are local to one thread (not multi-threaded) and rely on undo logs. Correct use of the API ensures that all memory operations are rolled back to the last committed state upon opening a pool after a server failure. VOS utilizes this API to ensure consistency of VOS internal data structures, even in the event of server failures.

事务只归属于1个线程，且使用的是redo log，异常场景向前回滚。

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

53

**14.2 VOS Concepts**

The versioning object store provides object storage local to a storage target by initializing a VOS pool (vpool) as one shard of a DAOS pool.

VOS跟target是1:1的，还是跟vpool是1:1的？

A vpool can hold objects for multiple object address spaces called containers. Each vpool is given a unique UID on creation, which is different from the UID of the DAOS pool. The VOS also maintains and provides a way to extract statistics like total space, available space, and the number of objects present in a vpool.

pool在每个target上都会有一个本地的实例化的vos pool，一个vpool可以包含多个container，每个container是一个单独的对象寻址空间。每个vpool在创建时指定了一个唯一的不同于pool的uid。VOS还提供了空间占用、对象计数等统计功能。

The primary purpose of the VOS is to capture and log object updates in arbitrary time order and integrate these into an ordered epoch history that can be traversed efficiently on

demand. This provides a major scalability improvement for parallel I/O by correctly ordering conflicting updates without requiring them to be serialized in time. For example, if two application processes agree on how to resolve a conflict on a given update, they may write their updates independently with the assurance that they will be resolved in the correct order at the VOS.

VOS的最初目标是跟踪记录不同时序的对象修改，将它们整合为一个基于epoch的有序集合，以实现高效的遍历。对并行冲突的IO正确排序后，可以实现并发执行，而不需要顺序执行，这样就可以有效提升并发。

The VOS also allows all object updates associated with a given epoch and process group to be discarded. This functionality ensures that when a DAOS transaction must be aborted, all associated updates are invisible before the epoch is committed for that process group and becomes immutable. This ensures that distributed updates are atomic - i.e. when a commit completes, either all updates have been applied or been discarded.

当一个事务异常终止时，与其相关的所有对象更新在epoch提交之前都是不可见的且不可修改的，这样保证了分布式更新的原子性。当1个提交完成时，所有的更新都持久化或者丢弃。

Finally, the VOS may aggregate the epoch history of objects to reclaim space used by inaccessible data and to speed access by simplifying indices. For example, when an array object is "punched" from 0 to infinity in a given epoch, all data updated after the latest snapshot before this epoch becomes inaccessible once the container is closed.

VOS对历史epoch的对象进行聚合，回收不可访问的数据占用的空间，减少索引以加速访问。如果某个对象在给定epoch被全部挖孔，一旦container关闭（？），在最新的快照和给定epoch之间更新的数据就变成不可访问的（可以被回收？）。

Internally, the VOS maintains an index of container UUIDs that references each container stored in a particular pool. The container itself contains three indices. The first is an object index used to map an object ID and epoch to object metadata efficiently when servicing I/O requests. The other two indices are for maintaining active and committed DTX records for ensuring efficient updates across multiple replicas.

VOS内部维护一个container uuid的索引，指示保存在某个特性pool内的每个container。每个container包含3个索引结构：

1）对象索引，使用对象ID和epoch映射到对象的元数据（**同一个对象的不同epoch是单独的index node？**），主要服务于IO

2）active DTX和committed DTX，保证多副本间高效的数据更新（**DAOS two-phase commit transaction？**）

DAOS supports two types of values, each associated with a Distribution Key (DKEY) and an Attribute Key (AKEY): Single value and Array value.

The DKEY is used for placement, determining which VOS pool is used to store the data. The AKEY identifies the data to be stored.

DKEY用于映射归置组并决定数据归属哪个VPOOL，AKEY用来定义存储的数据。

The ability to specify both a DKEY and an AKEY provides applications with the flexibility to either distribute or co-locate different values in DAOS.

A single value is an atomic value meaning that writes to an AKEY update the entire value and reads retrieve the latest value in its entirety.

Single value的value是原子的。

An array value is an index of equally sized records. Each update to an array value only affects the specified records and reads read the latest updates to each record index requested.

Array value的包含一组等大的records，每个record可以单独更新

Each VOS pool maintains the VOS provides a per container hierarchy of containers, objects, DKEYs, AKEYs, and values as shown below.

The DAOS API provides generic Key-Value and Array abstractions built on this underlying interface.

DAOS提供了2种通用的简化模型：key value和array

The former sets the DKEY to the user specified key and uses a fixed AKEY.

dkey由用户指定，akey是固定的，value是单值的

The latter uses the upper bits of the array index（**？？？不是用户指定**） to create a DKEY and uses a fixed AKEY thus evenly distributing array indices over all VOS pools in the object layout.

图形用户界面, 图示

描述已自动生成

For the remainder of the VOS description, Key-Value and Key-Array shall be used to describe the VOS layout rather than these simplifying abstractions. In other words, they shall describe the DKEY-AKEY-Value in a single VOS pool.

VOS objects are not created explicitly but are created on the first write by creating the object metadata and inserting a reference to it in the owning container's object index. All object updates log the data for each update, which may be an object, DKEY, AKEY, single value, or array value punch or an update to a single value or array value. Note that "punch"

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

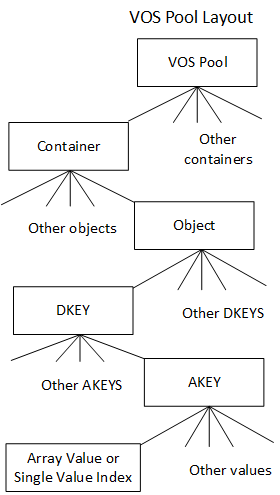
Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

54

of an extent of an array object is logged as zeroed extents, rather than causing relevant array extents or key values to be discarded.（Punch的结果是将数据置0，而不是释放对应的空间。）

A punch of an object, DKEY, AKEY, or single value is logged, so that reads at a later timestamp see no data. This ensures that the full version history of objects remains accessible.

The DAOS API, however, only allows accessing data at snapshots so VOS aggregation can aggressively remove objects, keys,and values that are no longer accessible at a known snapshot.（DAOS对外只允许访问对象的最新和快照数据，对于某个快照已经不可见的数据，VOS聚合可以删除对应的对象、keys和values）



**Figure 14-1. VOS pool storage layout**

When performing a lookup on a **single value** in an object, the object index is traversed to find the index node with the highest epoch number less than or equal to the requested epoch (near-epoch) that matches the key.（先找到离要求的epoch最近的且包含目标key的左值---obj index node） If a value or negative entry is found, it is returned. Otherwise, a "miss" is returned meaning that this key has never been updated in this VOS. This ensures that the most recent value in the epoch history of is returned irrespective of the time-order in which they were integrated and that all updates after the requested epoch are ignored.(保证读到离目标epoch最近的对象版本，命中的epoch<=目标epoch)

Similarly, when reading an **array object**, its index is traversed to create a gather descriptor that collects all object extent fragments in the requested extent with the highest epoch number less than or equal to the requested epoch. Entries in the gather descriptor either reference an extent containing data, a punched extent that the requestor can interpret as all zeroes, or a "miss," meaning that this VOS has received no updates in this extent. Again,this ensures that the most recent data in the epoch history of the array is returned for all

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

\*

55

offsets in the requested extent, irrespective of the time-order in which they were written and that all updates after the requested epoch are ignored.（**requested extent中不同的offset可以指定不同的epoch？不然的话就只需要读最近epoch的一个对象**）

**14.2.1 VOS Indexes**

The value of the object index table points to a DKEY index. The values in the DKEY index, indexed by DKEY and epoch, point to an AKEY index. The values in the AKEY index, indexed by AKEY and epoch, point to either a Single Value index or an Array index.

A single value index is referenced by epoch and will return the latest value inserted at or before the epoch. （SV Tree使用epoch索引）

An array value is indexed by the extent and the epoch and will return portions of extents visible at the epoch.（EV Tree）

Hints about the expectations of the object can be encoded in the object ID. For example, an object can be replicated, erasure coded, use checksums, or have integer or lexical DKEYs and AKEYs. （对象ID中包含对象控制元数据，key可以是整数或者string）

If an integer or lexical keys are used, the object index is ordered by keys, making queries, such as array size, more efficient. Otherwise, keys are ordered by the hashed value in the index.（使用key的哈希值进行排序）

The object ID is 128 bits. The upper 32 bits are used to encodes the object type, and key types while the lower 96 bits are a user-defined identifier that must be unique to the container.

**14.2.2 Object Listing**

VOS provides a generic iterator that can be used to iterate through containers, objects, DKEYs, AKEYs, single values, and array extents in a VOS pool. The iteration API is shown in the figure below.

/\*\*

\* Iterate VOS entries (i.e., containers, objects, dkeys, etc.) and call **\a**\* cb(**\a** **arg**) for each entry.

\* If **\a** **cb** returns a nonzero (either > 0 or < 0) value that is not

\* -DER\_NONEXIST, this function stops the iteration and returns that nonzero\* value from **\a** **cb**. If **\a** **cb** returns -DER\_NONEXIST, this function completes\* the iteration and returns 0. If **\a** **cb** returns 0, the iteration continues.\*

\* **\param[in]** **param** iteration parameters

\* **\param[in]** **type** entry type of starting level

\* **\param[in]** **recursive** iterate in lower level recursively

\* **\param[in]** **anchors** array of anchors, one for each

\* iteration level

\* **\param[in]** **cb** iteration callback

\* **\param[in]** **arg** callback argument

\*

\* **\retval** **0** iteration complete

\* **\retval** **>** 0 callback return value

\* **\retval** **-DER\_\*** error (but never -DER\_NONEXIST)

\*/

int

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

56

vos\_iterate(vos\_iter\_param\_t \*param, vos\_iter\_type\_t type, bool recursive,

**struct** vos\_iter\_anchors \*anchors, vos\_iter\_cb\_t cb, void \*arg);

The generic VOS iterator API enables both the DAOS enumeration API as well as DAOS internal features supporting rebuild, aggregation, and discard. It is flexible enough to iterate through all keys, single values, and extents for a specified epoch range. Additionally, it supports iteration through visible extents（？？？）.（可用于支持外部枚举API和内部rebuild aggregation和discard）

**14.3 Key Value Stores (Single Value)**

High-performance simulations generating large quantities of data require indexing and analysis of data, to achieve good insight. Key Value (KV) stores can play a vital role in simplifying storage of such complex data and allowing efficient processing.

VOS provides a multi-version, concurrent KV store on persistent memory that can grow dynamically and provide quick near-epoch retrieval（近时间段内快速恢复） and enumeration of key values.

Although there is an array of previous work on KV stores, most of them focus on cloud environments and do not provide effective versioning support. Some KV stores, provide versioning support but expect monotonically increasing ordering of versions and further, do not have the concept of near-epoch retrieval.

VOS must be able to accept insertion of KV pairs at any epoch and must be able to provide good scalability for concurrent updates and lookups on any key-value object. KV objects must also be able to support any type and size of keys and values.

**14.3.1 Operations Supported with Key Value Store**

VOS supports large keys and values with four types of operations namely,

update, lookup, punch, and key enumeration.

The update and punch operations add a new key to a KV store or log a new value of an existing key. Punch logs the special value "punched," effectively a negative entry, to record the epoch when the key was deleted.

Update会增加一个新的kv，punch会对已有的key log（？）一个新的特殊值（punched 是一个负值）

Sharing the same epoch for both an update and a punch of the same object, key, value, or extent is disallowed, and VOS will return an error when such is attempted.

不允许同一个epoch下即update又punch object、key、value或extent。

Lookup traverses the KV metadata to determine the state of the given key at the given epoch. If the key is not found at all, a "miss" is returned to indicate that the key is absent from this VOS. Otherwise, the value at the near-epoch or greatest epoch less than or equal to the requested epoch is returned. If this is the special "punched" value, it means the key was deleted in the requested epoch. The value here refers to the value in the internal tree-data structure. The key-value record of the KV-object is stored in the tree as the value of its node.（用户的kv存放在b+tree的node的value上） So in case of punch, this value contains a "special" return code/flag to identify the punch operation.

VOS also supports enumeration of keys belonging to a particular epoch.

**14.3.2 Key in VOS KV Stores**

VOS KV supports key sizes from small keys to extremely large keys. For AKEYs and DKEYs, VOS supports either hashed keys or one of two types of "direct" keys: lexical or integer.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

57

**14.3.2.1 Hashed Keys**

The most flexible key type is the hashed key. VOS runs two fast hash algorithms on the user supplied key and uses the combined hashed key values for the index. The intention of the combined hash is to avoid collisions between keys. The actual key still must be

compared for correctness.

使用快速的哈希算法映射用户的key，并采用combined hashed key进一步降低哈希冲突。

**14.3.2.2 Direct Keys**

The use of hashed keys results in unordered keys. This is problematic in cases where the user's algorithms may benefit from ordering. Therefore, VOS supports two types of keys that are not hashed but rather interpreted directly.

哈希之后的key顺序不能保持与用户key一致，如果用户算法对排序要求较高就不合适。

**14.3.2.3 Lexical Keys**

Lexical keys are compared using a lexical ordering. This enables usage such as sorted strings. Presently, lexical keys are limited in length, however to 80 characters.

String作为key，单个key最多80字符。

**14.3.2.4 Integer Keys**

Integer keys are unsigned 64-bit integers and are compared as such. This enables use cases such as DAOS array API using the upper bits of the index as a dkey and the lower bits as an offset. This enables such objects to use the DAOS key query API to calculate the size more efficiently.（高比特作为DKEY 低比特的最大范围就是这个DKEY对应的数据size了）

KV stores in VOS allow the user to maintain versions of the different KV pairs in random order. For example, an update can happen in epoch 10 and followed by another update in epoch 5, where HCE is less than 5. To provide this level of flexibility, each key in the KV store must maintain the epoch of update/punch along with the key. The ordering of entries in index trees first happens based on the key, and then based on the epochs.(b+trees不同的key按照key排序，相同的key按照epoch排序) This kind of ordering allows epochs of the same key to land in the same subtree, thereby minimizing search costs. Conflict resolution and tracking are performed using DTX described later. DTX ensures that replicas are consistent and failed or uncommitted updates are not visible externally.

**14.3.3 Internal Data Structures**

Designing a VOS KV store requires a tree data structure that can grow dynamically and re-main self-balanced. The tree needs to be balanced to ensure that time complexity does not increase with an increase in tree size. Tree data structures considered are red-black trees and B+ Trees, the former a binary search tree and the latter an n-ary search tree.

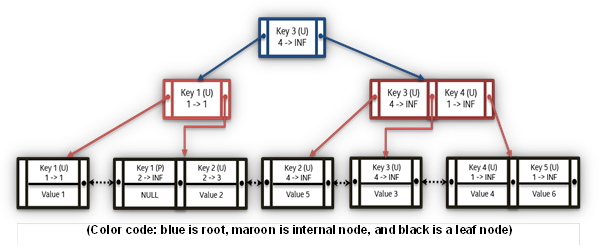
Although red-black trees provide less rigid balancing compared to AVL trees, they

compensate by having cheaper rebalancing cost. Red-black trees are more widely used in examples such as the Linux kernel, the java-util library, and the C++ standard template library. B+ trees differ from B trees in the fact that they do not have data associated with their internal nodes. This can facilitate fitting more keys on a page of memory.（inner node没有用户数据，一个内存可以存放更多的key） Also, leaf-nodes of B+ trees are linked; this means doing a full scan would require just one linear pass through all the leaf nodes, which can potentially minimize cache misses to access data in comparison to a B Tree.（所有的leaf node可以一次迭代）

To support update and punch as mentioned in the previous section (Operations Supported with Key Value Stores), an epoch-validity range is set along with the associated key for every update or punch request, which marks the key to be valid from the current epoch until the highest possible epoch（每个key后跟着一个epoch range，表示value生效的范围）.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

58

Updates to the same key on a future epoch or past epoch modifies the end epoch validity of the previous update or punch accordingly（当前修改的前一次修改对应的epoch range也需要更新）. This way only one key has a validity range for any given key-epoch pair lookup while the entire history of updates to the key is recorded.（同一个key的不同node的epoch range不重合） This facilitates the nearest-epoch search. Both punch and update have similar keys, except for a simple flag identifying the operation on the queried epoch（key后会跟一个flag标识是update还是punch）. Lookups must be able to search a given key in a given epoch and return the associated value. In addition to the epoch-validity range, the container handle cookie generated by DAOS is also stored along with the key of the tree. This cookie is required to identify behavior in case of overwrites on the same epoch.（container handle cookie是什么？？？）

A simple example of input for creating a KV store is listed in the Table below. Both a B+Tree based index and a red-black tree based index are shown in the Table and figure below,respectively. For explanation purposes, representative keys and values are used in the example.

**Table 14-1. Example VOS KV Store input for Update/Punch**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Key** |  |  | **Value** |  |  | **Epoch** |  |  | **Update (U/P)** |  |
| Key 1 | | | Value 1 | | | 1 | | | U | | |
| Key 2 | | | Value 2 | | | 2 | | | U | | |
| Key 3 | | | Value 3 | | | 4 | | | U | | |
| Key 4 | | | Value 4 | | | 1 | | | U | | |
| Key 1 | | | NIL | | | 2 | | | P | | |
| Key 2 | | | Value 5 | | | 4 | | | U | | |
| Key 3 | | | Value 6 | | | 1 | | | U | | |

**Figure 14-2. Red Black Tree based KV Store with Multi-Key**

The red black tree, like any traditional binary tree, organizes the keys lesser than the root to the left subtree and keys greater than the root to the right subtree. Value pointers are stored along with the keys in each node. On the other hand, a B+ Tree based index stores keys in ascending order at the leaves, which is where the value is stored. The root nodes and internal nodes (color-coded in blue and maroon accordingly) facilitate locating the appropriate leaf node. Each B+ Tree node has multiple slots, where the number of slots is determined from the order. The nodes can have a maximum of order-1 slots.（树的阶数决定单个node的最大slot数） The container handle cookie must be stored with every key in case of red black trees, but in case of B+

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

59

Trees having cookies only in leaf nodes would suffice, since cookies are not used in

traversing.

In the table below, n is the number of entries in a tree, m is the number of keys, k is the number of the key, epoch entries between two unique keys.

**Table 14-2. Comparison of average case computational complexity for index**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Operation** |  |  | **Reb-black tree** |  |  | **B+Tree** |  |
| Update | | | O(log2n) | | | O(logbn) | | |
| Lookup | | | O(log2n) | | | O(logbn) | | |
| Delete | | | O(log2n) | | | O(logbn) | | |
| Enumeration | | | O(m\* log2(n) + log2(n)) | | | O(m \* k + logb (n)) | | |

Although both these solutions are viable implementations, determining the ideal data structure would depend on the performance of these data structures on persistent memory hardware.

VOS also supports concurrent access to these structures, which mandates that the data structure of choice provide good scalability while there are concurrent updates. Compared to B+ Tree, rebalancing in red-black trees causes more intrusive tree structure change; accordingly, B+ Trees may provide better performance with concurrent accesses.

Furthermore, because B+ Tree nodes contain many slots depending on the size of each node, prefetching in cache can potentially be easier. Also, the sequential computational complexities in the Table above show that a B+ Tree-based KV store with a reasonable order can perform better in comparison to a Red-black tree.

VOS supports enumerating keys valid in a given epoch. VOS provides an iterator-based approach, to extract all the keys and values from a KV object. Primarily, KV indexes are ordered by keys and then by epochs. With each key holding a long history of updates, the size of a tree can be huge. Enumeration with a tree-successors approach can result in an asymptotic complexity of O(m\* log (n) + log (n)) with red-black trees, where m is the

number of keys valid in the requested epoch. It takes O(log2 (n)) to locate the first element in the tree and O(log2 (n)) to locate a successor. Because "m" keys need to be retrieved, O(m \* log2 (n)) would be the complexity of this enumeration.

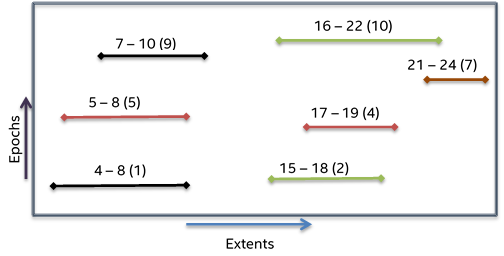
In the case of B+-trees, leaf nodes are in ascending order, and enumeration would be to parse the leaf nodes directly. The complexity would be O (m \* k + logbn), where m is the number of keys valid in an epoch, k is the number of entries between two different keys in B+ tree leaf nodes, and b is the order for the B+tree. Having "k" epoch entries between two distinct keys incurs in a complexity of O(m \* k). The additional O(logbn) is required to locate the first leftmost key in the tree. The generic iterator interfaces as shown in Figure above would be used for KV enumeration also.

In addition to the enumeration of keys for an object valid in an epoch, VOS also supports enumerating keys of an object modified between two epochs. The epoch index table

provides keys updated in each epoch. On aggregating the list of keys associated with each epoch, (by keeping the latest update of the key and discarding the older versions) VOS can generate a list of keys, with their latest epoch. By looking up each key from the list in its associated index data structure, VOS can extract values with an iterator-based approach.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

60

**14.4 Key Array Stores**

The second type of object supported by VOS is a Key-Array object. Array objects, similar to KV stores, allow multiple versions and must be able to write, read, and punch any part of the byte extent range concurrently. The figure below shows a simple example of the extents and epoch arrangement within a Key-Array object. In this example, the different lines represent the actual data stored in the respective extents and the color-coding points to different threads writing that extent range.

**Figure 14-3. Example of extents and epochs in a Key Array object**

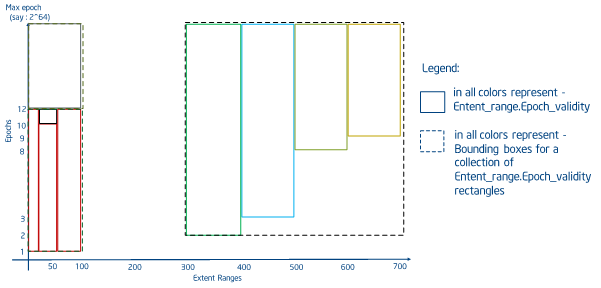
In the above example, there is significant overlap between different extent ranges. VOS supports nearest-epoch access, which necessitates reading the latest value for any given extent range. For example, in the figure above, if there is a read request for extent range 4- 10 at epoch 10, the resulting read buffer should contain extent 7-10 from epoch 9, extent5-7 from epoch 8, and extent 4-5 from epoch 1. VOS array objects also support punch over both partial and complete extent ranges.

**Table 14-3. Example Input for Extent Epoch Table**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Extent Range** |  |  | **Epoch** |  |  | **Write (or) Punch** |  |
| 0 - 100 | | | 1 | | | Write | | |
| 300 - 400 | | | 2 | | | Write | | |
| 400 - 500 | | | 3 | | | Write | | |
| 30 - 60 | | | 10 | | | Punch | | |
| 500 - 600 | | | 8 | | | Write | | |
| 600 - 700 | | | 9 | | | Write | | |

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

61

R-Trees provide a reasonable way to represent both extent and epoch validity ranges in such a way as to limit the search space required to handle a read request. VOS provides a specialized R-Tree, called an Extent-Validity tree (EV-Tree) to store and query versioned array indices. In a traditional R-Tree implementation, rectangles are bounded and

immutable. In VOS, the "rectangle" consists of the extent range on one axis and the epoch validity range on the other. However, the epoch validity range is unknown at the time of insert, so all rectangles are inserted assuming an upper bound of infinity. Originally, the DAOS design called for splitting such in-tree rectangles on an insert to bound the validity range, but a few factors resulted in the decision to keep the original validity range. First,updates to persistent memory are an order of magnitude more expensive than lookups.Second, overwrites between snapshots can be deleted by aggregation thus maintaining a reasonably small history of overlapping writes. As such, the EV-Tree implements a two-part algorithm on fetch.

1. Find all overlapping extents. This will include all writes that happened before the

requested epoch even if they are covered by a subsequent write.

2. Sort this by extent start and then by epoch

3. Walk through the sorted array, splitting extents if necessary and marking them as visible

as applicable

4. Re-sort the array. This final sort can optionally keep or discard holes and covered

extents, depending on the use case.

TODO: Create a new figure Rectangles representing extent\_range.e,poch\_validity arranged in 2-D space for an order-4 EV-Tree using input in the table above.

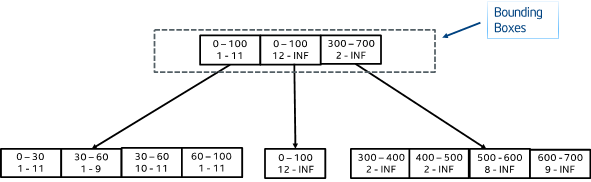
**Figure 14-4. Rectangles representing extent\_range.epoch\_validity arranged in 2-D space for an**

**order-4 EV-Tree using input in the table**

The figure below shows the rectangles constructed with splitting and trimming operations of EV-Tree for the example in the previous table with an additional write at offset {0 - 100}introduced to consider the case for extensive splitting. The figure above shows the EV-Tree construction for the same example.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

62

**Figure 14-5. Tree (order - 4) for the example in Table 6 3 (pictorial representation shown in the**

**figure above**

Inserts in an EV-Tree locate the appropriate leaf-node to insert, by checking for overlap. If multiple bounding boxes overlap, the bounding box with the least enlargement is chosen.Further ties are resolved by choosing the bounding box with the least area. The maximum cost of each insert can be O (logbn).

Searching an EV-Tree would work similar to R-Tree, aside from the false overlap issue described above. All overlapping internal nodes must be pursued, till there are matching internal nodes and leaves. Since extent ranges can span across multiple rectangles, a single search can hit multiple rectangles. In an ideal case (where the entire extent range falls on one rectangle) the read cost is O(logbn) where b is the order of the tree. The sorting and splitting phase adds additional overhead of O(n log n) where n is the number of matching extents. In the worst case, this is equivalent to all extents in the tree, but this is mitigated by aggregation and the expectation that the tree associated with a single shard of a single key will be relatively small.

For deleting nodes from an EV-Tree, the same approach as search can be used to locate nodes, and nodes/slots can be deleted. Once deleted, to coalesce multiple leaf-nodes that have less than order/2 entries, reinsertion is done. EV-tree reinserts are done (instead of merging leaf-nodes as in B+ trees) because on deletion of leaf node/slots, the size of

bounding boxes changes, and it is important to make sure the rectangles are organized into minimum bounding boxes without unnecessary overlaps. In VOS, delete is required only during aggregation and discard operations. These operations are discussed in the following section (Epoch Based Operations).

**14.5 Epoch Based Operations**

Epochs provide a way for modifying VOS objects without destroying the history of

updates/writes. Each update consumes memory and discarding unused history can help reclaim unused space. VOS provides methods to compact the history of writes/updates and reclaim space in every storage node. VOS also supports rollback of history in case

transactions are aborted. The DAOS API timestamp corresponds to a VOS epoch. The API only allows reading either the latest state or from a persistent snapshot which is simply a reference on a given epoch.

To compact epochs, VOS allows all epochs between snapshots to be aggregated, i.e., the value/extent-data of the latest epoch of any key is always kept over older epochs. This also ensures that merging history does not cause loss of exclusive updates/writes made to an epoch. To rollback history VOS provides the discard operation.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

63

int vos\_aggregate(daos\_handle\_t coh, daos\_epoch\_range\_t \*epr);

int vos\_discard(daos\_handle\_t coh, daos\_epoch\_range\_t \*epr);

int vos\_epoch\_flush(daos\_handle\_t coh, daos\_epoch\_t epoch);

Aggregate and discard operations in VOS accept a range of epochs to be aggregated normally corresponding to ranges between persistent snapshots.

**14.5.1 VOS Discard**

Discard forcefully removes epochs without aggregation. Use of this operation is necessary only when value/extent-data associated with a pair needs to be discarded.（只有成对的value或者extent需要丢弃时才使用，怎么理解？） During this operation, VOS looks up all objects associated with each cookie in the requested epoch range from the cookie index table and removes the records directly from the respective object trees by looking at their respective epoch validity. DAOS requires discard to service abort requests. Abort operations require discard to be synchronous.

During discard, keys and byte-array rectangles need to be searched for nodes/slots whose end-epoch is (discard\_epoch - 1). This means that there was an update before the now discarded epoch and its validity got modified to support near-epoch lookup. This epoch validity of the previous update has to be extended to infinity to ensure future lookups at near-epoch would fetch the last known updated value for the key/extent range.

只有修改回退才会使用discard？不然前个版本的validity upper range可能不是更新为infinity；discard成功前是否允许新的update，如果允许discard\_epoch-1对应的upper range也可能不是infinity。

**14.5.2 VOS Aggregate**

During aggregation, VOS must retain the latest update to a key/extent-range discarding the others and any updates visible at a persistent snapshot. VOS can freely remove or

consolidate keys or extents so long as it doesn't alter the view visible at the latest

timestamp or at any persistent snapshot epoch. Aggregation makes use of the vos\_iterate API to find both visible and hidden entries between persistent snapshots and removes hidden keys and extents and merges contiguous partial extents to reduce metadata overhead. Aggregation can be an expensive operation but doesn't need to consume cycles on the critical path. A special aggregation ULT processes aggregation, yielding frequently to avoid blocking continuing I/O.

聚合只保留最新和快照数据，还会对连续离散的extent聚合以减少元数据，采用单独的ULT处理并会定期yield以降低对IO的影响。

**14.6 VOS Checksum Management (TO BE UPDATED)**

One of the guarantees that VOS provides is end-to-end data integrity. Data corruption in VOS can happen while reading or writing data due to a variety of reasons, including leaks or failures in persistent memory, or during data transmission through the wire. VOS supports data integrity check with checksums.

The VOS API for updates and writes will require checksums as arguments from its upper layer(s). VOS requires checksum for both keys and values in case of KV objects. VOS stores the checksum along with the data.

A Lookup operation on a KV will verify the checksum by computing the checksum for the key and value. If reads in byte-arrays span multiple extent ranges, VOS would have to recompute the checksum at the server for each individual extent range for verifying their integrity and return the computed checksum of the entire requested extent range to the client. In case a read requests a partial byte array extent of an existing extent range, VOS would compute the checksum of the existing extent to verify correctness and then return

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

64

the requested extent range to the client with its computed checksum. When byte array extents are aggregated, VOS individually re-computes checksum of all extent ranges to be merged to verify correctness, and finally computes and saves the checksum for the merged extent range.

**14.7 Metadata Overhead**

VOS provides a tool vos\_stats.py that can take a set of assumptions about how many keys and objects and VOS pools are in use and provide an estimate of metadata overhead.TODO: Provide more discussion

To run an example, first setup the paths

[~/daos]$ source ./scons\_local/utils/setup\_local.sh

/home/jvolivie/daos

Build vars file found: ./.build\_vars.sh

Then run vos\_size to create vos\_size.yaml with metadata size information

[~/daos]$ vos\_size

Finally, execute vos\_size.py to get a meta data estimate for the use cases in an input.yaml.An example input yaml is installed to /etc.

[~/daos]$ vos\_size.py "${SL\_PREFIX}/etc/vos\_size\_input.yaml"

Metadata totals:

pool : 2 K ( 0.00%)

container : 9 K ( 0.00%)

object : 11 K ( 0.00%)

dkey : 13 K ( 0.00%)

akey : 42 K ( 0.00%)

single\_value : 40 K ( 0.00%)

array : 248 K ( 0.01%)

total\_meta : 367 K ( 0.02%)

Total bytes with user data: 2053492K

**14.8 Replica Consistency**

DAOS supports multiple replicas for data high availability. Inconsistency between replicas is possible when a target fails during an update to a replicated object and when concurrent updates are applied on replicated targets in an inconsistent order.

The most intuitive solution to the inconsistency problem is distributed lock (DLM), used by some distributed systems, such as Lustre. For DAOS, a user-space system with powerful,next-generation hardware, maintaining distributed locks among multiple, independent application spaces will introduce unacceptable overhead and complexity. DAOS instead uses an optimized two-phase commit transaction to guarantee consistency among replicas.

采用两阶段提交事务来保证副本间的一致性。

**14.8.1 DAOS Two-Phase Commit (DTX)**

When an application wants to modify (update or punch) an object with multiple replicas, the client sends the modification RPC to the leader replica (Via DTX Leader Election algorithm

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

65

discussed below). The leader dispatches the RPC to the other replicas and each replica makes its own modification in parallel. Bulk transfers are not forwarded by the leader but rather transferred directly from the client, improving load balance and decreasing latency by utilizing the full client-server bandwidth.

Before modifications are made, a local transaction, called 'DTX', is started on each replica with a client selected DTX identifier that is unique for the current RPC within the container.

1、在修改开始前每个副本都构造一个本地事务DTX，由client给DTX指定一个container内唯一的ID。

All modifications in a DTX are logged in a DTX transaction table and back references to the table are kept in each modified record.

After local modifications are done, each non-leader replica marks the DTX state as 'prepared' and replies to the leader replica.

2、每个分片上的本地修改完成后，非leader分片的dtx状态切换为prepared，并非lead回复

The leader sets the DTX state to 'committable' as soon as it has completed its own modifications and has received successful replies from all replicas.

3、leader在本地修改完成且收到所有从分片成功回复后，状态切换为committable

如果有从分片回复失败，则leader会给其他从分片发送Abort，并将状态切换为Abort

If any replica(s) failed to execute the modification, it will reply to the leader with failure and the leader will ask remaining replicas to 'abort' the DTX.

Once the DTX is set by the leader to 'committable' or 'abort', it replies to the client with the appropriate status.

4、在leader分片将DTX状态设置为committable或者abort后，给client回复成功或失败。

The client may consider a modification complete as soon as it receives a successful reply from the leader, regardless of whether the DTX is actually 'committed' or not.

It is the responsibility of the leader replica to commit the 'committable' DTX asynchronously when the 'committable' DTX count exceeds some threshold or piggybacked via dispatched RPCs due to potential conflict with subsequent modifications.

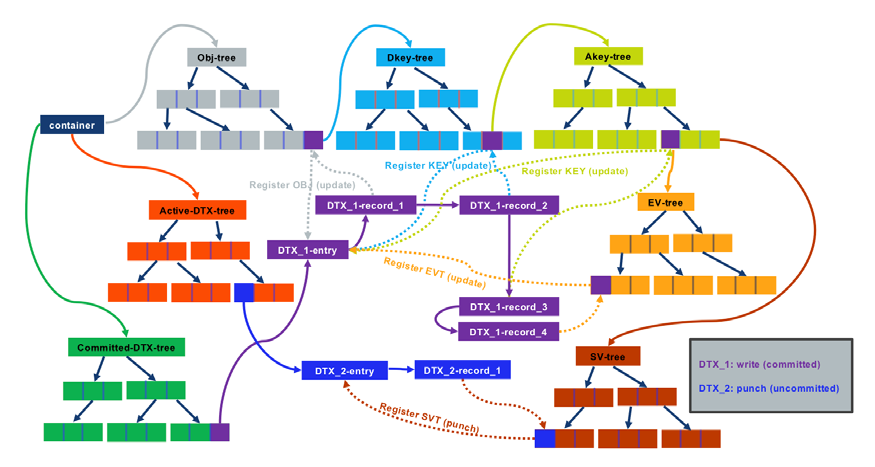
5、当committable状态的DTX超过某个阈值后，leader分片负责通知所有分片切换为committed状态如果后续的修改与已经处于committable状态的修改冲突，则leader会在分发给从分片的RPC消息中携带DTX状态切换的消息。

When an application wants to read something from an object with multiple replicas, the client can send the RPC to any replica. On the server side, if related DTX has been committed or is committable, the record can be returned to If the DTX state is prepared and the replica is not the leader, it will reply to the client telling it to send the RPC to the leader instead. If it is the leader and is in any state other than 'committed' or 'committable', the entry is ignored and the latest committed modification is returned to the client.

The DTX model is built inside the DAOS container. Each container maintains its DTX table that is organized as two B+tree in SCM: one for active DTXs and the other for committed DTXs. The following diagram represents the modification of a replicated object under DTX model.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

66

**Figure 14-6. Modify multiple replicated object under DTX model**

**1、DTX\_1 write(committed) DTX\_1-entry中包含4个record，分别对应obj dkey akey和ev tree node的修改，每个record指向对应节点和dtx-committed tree table。(图里展示的只是一个最终的静态结果)**

**2、DTX\_2 punch(uncommitted)**

**14.8.2 DTX Leader Election**

In the DTX model, the leader is a special replica that does more work than other replicas, including:

1. All modification RPCs are sent to the leader. The leader performs necessary sanity

checks before dispatching modifications to other replicas.

2. Non-leader replicas tell the client to redirect reads in 'prepared' DTX state to the leader

replica. The leader, therefore, may handle a heavier load on reads than non-leaders.To avoid general load imbalance, leader selection is done for each object or dkey following these general guidelines:

• R1: When different replicated objects share the same redundancy group, the same

leader should not be used for each object.

• R2: When a replicated object with multiple DKEYs spans multiple redundancy groups,

the leaders in different redundancy groups should be on different servers.

• R3: Servers that fail frequently should be avoided in leader selection to avoid too

frequent leader migration.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

67

**15 Blob I/O**

The Blob I/O (BIO) module was implemented for issuing I/O over NVMe SSDs. The DAOS service has two tiers of storage: Storage Class Memory (SCM) for byte-granular application data and metadata, and NVMe for bulk application data. Similar to how PMDK is currently used to facilitate access to SCM, the Storage Performance Development Kit (SPDK) is used to provide seamless and efficient access to NVMe SSDs. The current DAOS storage model involves three DAOS server xstreams per core, along with one main DAOS server xstream per core mapped to an NVMe SSD device. DAOS storage allocations can occur on either SCM by using a PMDK pmemobj pool, or on NVMe, using an SPDK blob. All local server metadata will be stored in a per-server pmemobj pool on SCM and will include all current and relevant NVMe device, pool, and xstream mapping information.

This document contains the following sections:

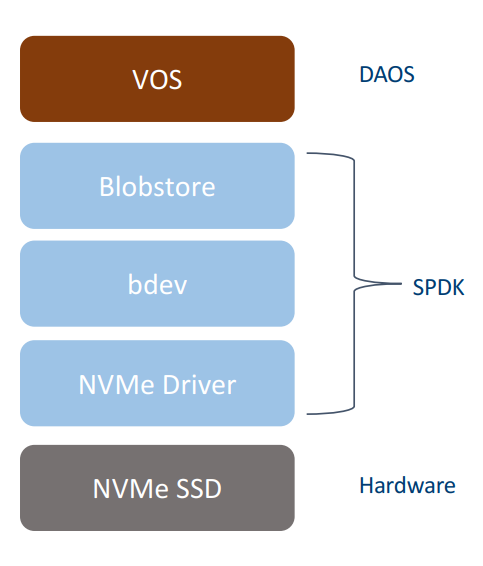
**15.1 Storage Performance Development Kit (SPDK)**

SPDK is an open source C library that when used in a storage application, can provide a significant performance increase of more than 7X over the standard NVMe kernel driver.SPDK's high performance can mainly be attributed to the user space NVMe driver,

eliminating all syscalls and enabling zero-copy access from the application. In SPDK, the hardware is polled for completions as opposed to relying on interrupts, lowering both total latency and latency variance. SPDK also offers a block device layer called bdev, which sits immediately above the device drivers like in a traditional kernel storage stack. This module offers pluggable module APIs for implementing block devices that interface with different types of block storage devices. This includes driver modules for NVMe, Malloc (ramdisk),Linux AIO, Ceph RBD, and others.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

68

**Figure 15-1. SPDK Software Stack**

**15.1.1 SPDK NVMe Driver**

The NVMe driver is a C library linked to a storage application providing direct, zero-copy data transfer to and from NVMe SSDs. Other benefits of the SPDK NVMe driver are that it is entirely in user space, operates in polled-mode vs. interrupt-dependent, is asynchronous and lock-less.

**15.1.2 SPDK Block Device Layer (bdev)**

The bdev directory contains a block device abstraction layer used to translate from a

common block protocol to specific protocols of backend devices, such as NVMe. Additionally,this layer provides automatic queuing of I/O requests in response to certain conditions,lock-less sending of queues, device configuration and reset support, and I/O timeout trafficking.

**15.1.3 SPDK Blobstore**

The blobstore is a block allocator for a higher-level storage service. The allocated blocks are termed 'blobs' within SPDK. Blobs are designed to be large (at least hundreds of KB), and therefore another allocator is needed in addition to the blobstore to provide efficient small block allocation for the DAOS service. The blobstore provides asynchronous, un-cached, and parallel blob read and write interfaces

**15.2 SPDK Integration**

The BIO module relies on the SPDK API to initialize/finalize the SPDK environment on the DAOS server start/shutdown. The DAOS storage model is integrated with SPDK by the following:Management of SPDK blobstores and blobs: NVMe SSDs are assigned to each DAOS server xstream. SPDK blobstores are created on each NVMe SSD. SPDK blobs are created and attached to each per-xstream VOS pool.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

69

Association of SPDK I/O channels with DAOS server xstreams: Once SPDK I/O channels are properly associated with the corresponding device, NVMe hardware completion pollers are integrated into server polling ULTs.

**15.3 Per-Server Metadata Management (SMD)**

One of the major subcomponents of the BIO module is per-server metadata management.The SMD submodule consists of a PMDK pmemobj pool stored on SCM used to track each DAOS server's local metadata.

Currently, the persistent metadata tables tracked are:

• **NVMe Stream Table**: NVMe SSD to DAOS server xstream mapping (local PCIe attached

NVMe SSDs are assigned to different server xstreams to avoid hardware contention)

• **NVMe Pool Table**: NVMe SSD, DAOS server xstream, and SPDK blob ID mapping (SPDK

blob to VOS pool:xstream mapping)

• **NVMe Device Table**: (in progress) NVMe SSD to device status mapping On DAOS server

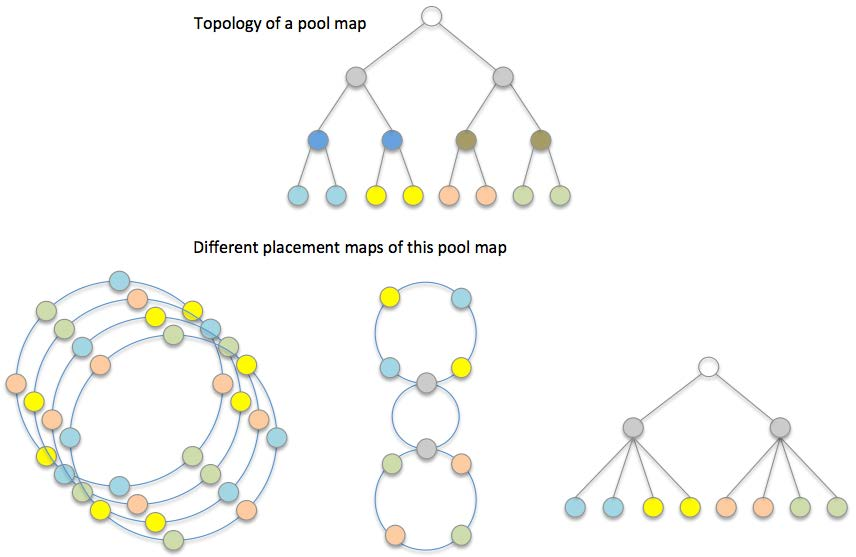
start, these tables are loaded from persistent memory and used to initialize new, and load any previous blobstores and blobs. Also, there is potential to expand this module to support other non-NVMe related metadata in the future.

**15.4 DMA Buffer Management**

BIO internally manages a per-xstream DMA safe buffer for SPDK DMA transfer over NVMe SSDs. The buffer is allocated using the SPDK memory allocation API and can dynamically grow on demand. This buffer also acts as an intermediate buffer for RDMA over NVMe SSDs,meaning on DAOS bulk update, client data will be RDMA transferred to this buffer first, then the SPDK blob I/O interface will be called to start local DMA transfer from the buffer directly to NVMe SSD. On DAOS bulk fetch, data present on the NVMe SSD will be DMA transferred to this buffer first, and then RDMA transferred to the client.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

70

**16 Algorithmic object placement**

DAOS-SR uses the pool map to create a set of placement maps that are used to compute algorithmic object layouts and to drive consensus data distribution. This approach uses consistent hash-based algorithms to generate object layout based on object ID, object schema, and one of the placement maps. The major placement map algorithm discussed in this document is the Ring Placement Map, which is a concept developed in collaboration with Argonne National Laboratory, see preliminary work in P. Carns et al., Consistent Hashing Distance Metrics for Large-Scale Object Storage

(http://sc15.supercomputing.org/sites/all/themes/SC15images/tech\_poster/tech\_poster\_pa ges/post117.html).

**16.1 Placement Map**

A placement map essentially is an abstracted and permutated pool map; it does not

necessarily include all details of the pool map. Instead, it only retains component

relationships that can be used to distribute object shards for the resilience and performance requirements of the application.

**Figure 16-1. Pool-map and placement maps**

A placement map does not maintain a copy of status or any characteristics of the

corresponding pool map components, but only references pool map components. Each time DAOS-SR computes an object distribution based on a placement map, it also needs to check the corresponding component status and attributes from the pool map. This adds an extra step for indirect memory access, but can significantly reduce cache pollution, and memory consumption in case there are many placement maps but only one pool map in a DAOS pool.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

71

As shown in the figure, a storage pool may have multiple types of placement maps because different applications can have various fault tolerance and performance requirements. Also,there can be many instances of the same placement map to accelerate rebuild and

rebalance by workload declustering. This topic will be introduced in Multi-ring Placement Map".

**16.2 Ring Placement Map**

The first approach of building a placement map is to place storage targets on a consistent hashing ring, called a ring placement map. Ring placement can handle one level of fault domain to avoid data loss in the event of multiple correlated target failures. If the number of fault domains is more than the size of the redundancy group of an object, then a ring placement map can guarantee that data of this object will survive even if all targets within a fault domain fail together. To achieve this goal, the ring placement map should evenly distribute targets from the same fault domain on the placement ring. For example, if there are four targets within a domain, then the distance between each two targets from this domain should be an arc of 90 degrees on the ring.

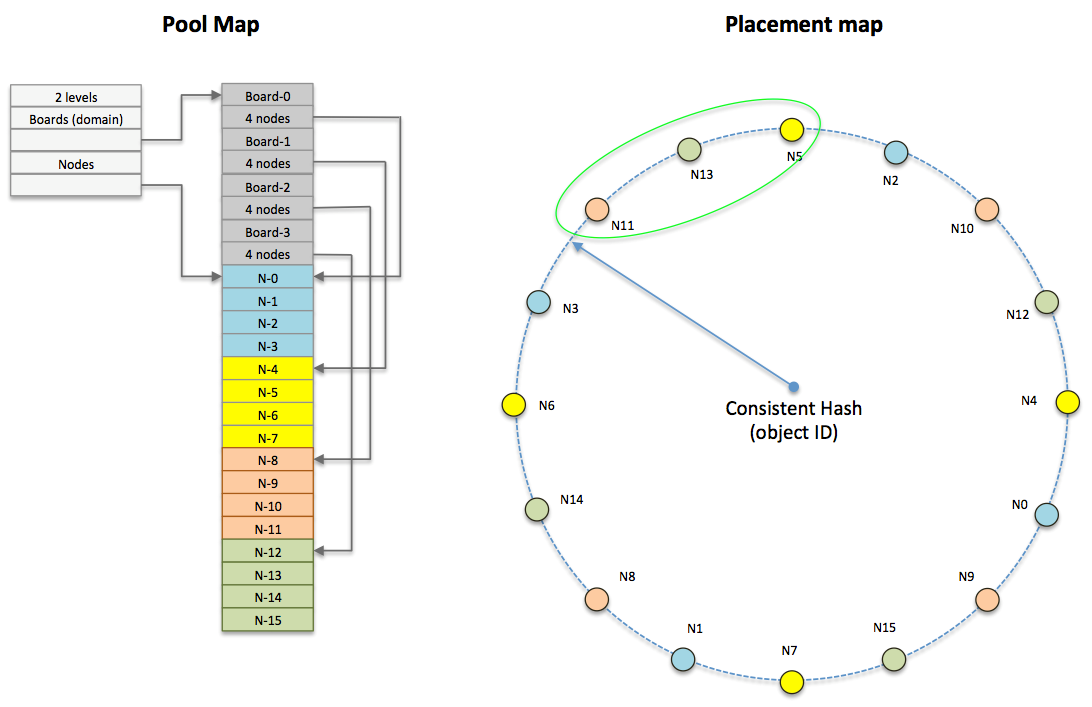
The figure below shows how to build a ring placement map. Targets in the same color are within the same fault domain and are evenly placed on the ring so that arc-distances between them are always identical. To simplify the description, all domains have the same number of targets in this example, but this is not a mandatory requirement. A proposal of eliminating this limit will be introduced in the section Multi-Ring Placement Map.

It is also worth mentioning that sequences of targets are randomized in the example in the figure below. This is unnecessary if only one ring placement map is built for the DAOS pool,but it is important if multiple placement rings are created for the pool. The reason for this will also be explained in Multi-Ring Placement Map as well.

To generate the distribution of an object by using a ring placement map, DAOS-SR hashes the ID of the object to a position on the consistent hashing ring. If the object is to be striped S ways over redundancy groups of size R, it then places the object shards on the next N=S\*R targets on the ring in the direction of consistent hash. In the figure below,three targets in the green ellipse have been selected for a 3-way replicated object.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

72

**Figure 16-2. Ring Placement Map**

**16.2.1 Fault Tolerance**

Since the ring placement map ensures targets in the same fault domain are widely spaced,then provided there are enough fault domains, the ring map will not place object shards in the same redundancy group on targets in the same fault domain. Redundant objects placed using the ring placement map can, therefore, tolerate the correlated failure of targets

sharing the same fault domain. The number of such correlated failures that can be tolerated depends on the size of the objects redundancy groups. As shown in the figure above,although four targets will fail on any board failure, any two boards may fail without data loss.

**16.2.2 Failure Handling and Rebuild**

Although redundant object data may not be lost on target failure, the objects with shards that became inaccessible due to the failure must now operate in degraded mode. It is important to rebuild this object data as soon as possible after failure is detected for the following reasons.

• Object data is more vulnerable to loss from further failures in degraded mode since

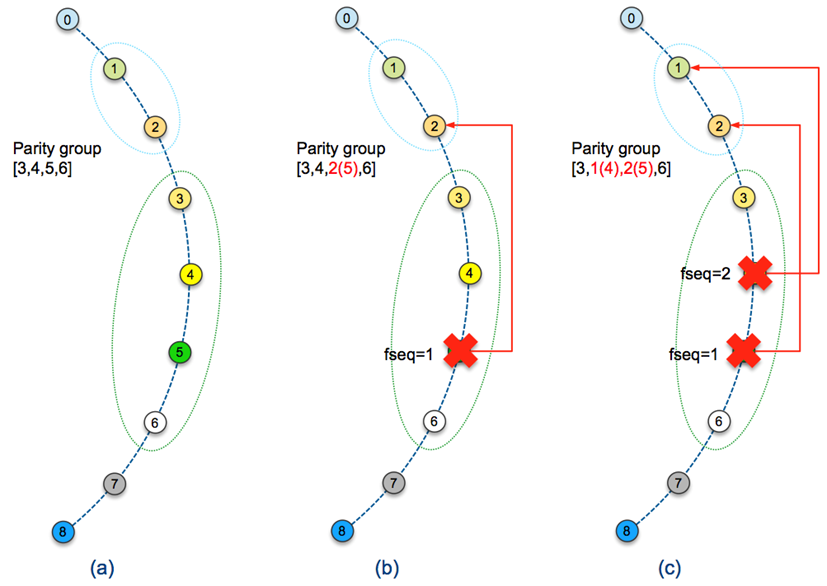
redundancy has been reduced or may even have been eliminated.

• Performance is impacted in degraded mode. For replicated objects, reads must now be

distributed over fewer shards, and for erasure-coded objects, data from the failed shard may have to be reconstructed from many other shards sharing the same redundancy group.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

73

**16.2.3 Rebuild Targets**

Rebuild targets are targets that will be used to reconstruct object shards that become inaccessible due to target failures. In the figure below, an object is distributed on targets-[3, 4, 5, 6] and rebuild targets for this object are targets-[1, 2]. Rebuild targets may be selected from prior targets, in reverse consistent hash order, to the first member of a given redundancy group (RDG). Successive failures may continue to select the next rebuild target in reverse hash ring order until a target would be selected that does not share the same fault domain as any RDG member.

When an object is striped across many RDGs, and RDGs are placed contiguously on the ring,selecting a rebuild target immediate before the RDG with failure will cause shards from the same object, albeit in different RDGs, to be located on the same target. This will degrade performance on failure because I/O will no longer be balanced.

To avoid this imbalance, widely distributed striped objects may be placed with gaps be-tween RDGs to leave spare targets for rebuild. Because leaving such a gap between all RDGs in an object striped over the whole DAOS pool will leave a significant proportion of targets spare before the first failure, gaps may be introduced at less regular intervals and shared by groups of RDGs. This increases overall performance at the expense of locating rebuild targets more distantly on the hash ring from the degraded RDG, therefore a balance must be found between these competing requirements.

**Figure 16-3. Rebuild targets and failure sequence**

Note that after all spare targets in a gap have become rebuild targets, further failures can still be handled by selecting rebuild targets immediately before the first member of RDGs affected by the failure, albeit at the expense of load balance. This also means that multiple shards of the same object can be stored on the same target. To distinguish these collocated

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

74

shards from the same object, DSR can construct DSM object ID by appending object shard index to DSR object ID.

On very small DAOS pools with relatively few fault domains, it may not be possible to allocate gaps of spare rebuild targets at all, because the proportion of gaps and

performance loss may be unacceptably high. However, on large DAOS pools with many fault domains, sufficiently large gaps needed to ensure balanced I/O after many failures could be spaced widely and therefore occupy a sufficiently small proportion of the ring to ensure good performance.

**16.2.3.1 Failure Sequence and Rebuild**

Because members of an erasure-coded RDG are all different, it is vital to track the mapping of RDG member to rebuild target correctly when the RDG member is made inaccessible by a failure. This must be done without generating any additional per-RDG or per-object

metadata for truly algorithmic object layouts.

RDG members are numbered consecutively from zero and are initially placed in order on adjacent placement ring targets. As targets fail, the next available rebuild RDG must be selected, initially from spares in any prior gap and then from remaining rebuild target candidates. This must be done deterministically, irrespective of the order in which targets fail. As shown in the figure above, DAOS-SR must determine which of target-[1, 2] is the replacement of target-4, and which is the replacement of target-5.

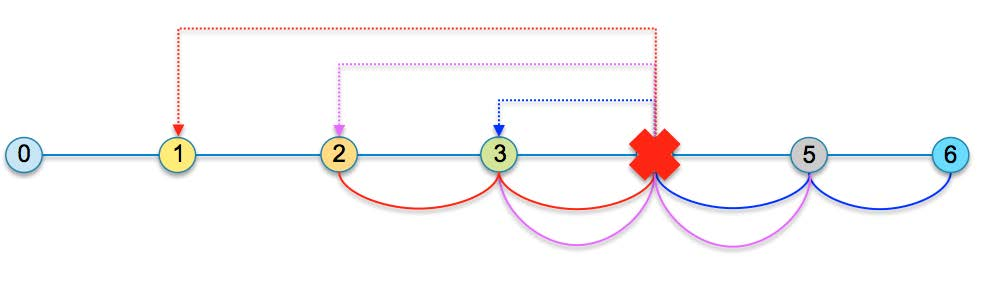
Target failure-order is used to ensure deterministic mapping of RDG member to targets.DAOS pools increment the version number of the pool map on all target state changes and record this version number with the target on failure so that failed targets can be ordered by the map version at which they failed. This sequence, in turn, determines the order in which candidate rebuild targets are selected. In the example of (c) in the figure above, both target-4 and target-5 failed, but it does not mean target-4 selected the near spare target-2as a replacement, because target-4 has higher failure sequence than target-5 so it should select the spare target after target-5.

**16.2.3.2 Load Balancing for Failure**

When there is no failure, the ring placement map can evenly distribute objects to all targets.However, when there is a failure, the missing object shards should be rebuilt on surviving spare targets. Because these spare targets are also regular targets for other objects, they could store more objects than other targets and have to handle more I/O requests. In this case, the I/O workload of storage cluster is imbalanced, and this could be a major source of jitter and affect the overall performance.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

75

**Figure 16-4. Object distribution on rebuild targets**

In the figure above, if all objects are 3-way replicated, and the layout of each of them is the targets connected by arcs in the same color. When target-4 fails, as each object will choose the prior target as a spare target:

• A new replica of the red object will be rebuilt on target-1.

• A new replica of the purple object will be rebuilt on target-2.

• A new replica of the blue object will be rebuilt on target-3.

Therefore, targets 1, 2, and 3 will each take 33% of the objects from target-4, and they have to handle 33% more I/O requests than other targets.

One way to reduce data imbalance like this is to create multiple rings, which will be

introduced later. The potential problem with multiple rings is they can consume a lot of memory and CPU cache. For example, if there are hundreds of thousands of targets in the DAOS pool, each ring may consume one megabyte of RAM, and a multi-ring approach would not be a good option. The other approach to reduce imbalance is to allow objects to choose rebuild targets at varying distances on the placement ring, instead of always selecting the nearest preceding target as a rebuild target. For example, DAOS-SR can hash the object ID and use the modulo of the hashed ID as the distance between the first object shard and the first rebuild target. In in the figure above, if the calculated distance is zero, target-1 is still the rebuild target for the red object. However, if the calculated distance is 1, then target-0is the rebuild target for the red object. In this way, if the ring is big enough, then the distance to a rebuild target can vary between 0 and a larger value. It means that missing object shards on the failed target can be rebuilt on many different targets, and each of these targets only takes over a small amount of data and I/O requests from the failed target.

**16.2.4 Rebalancing for Pool Extension**

When new storage targets are added to a DAOS pool, containers within this pool should rebalance data by migrating some of the existing object shards to the new targets.

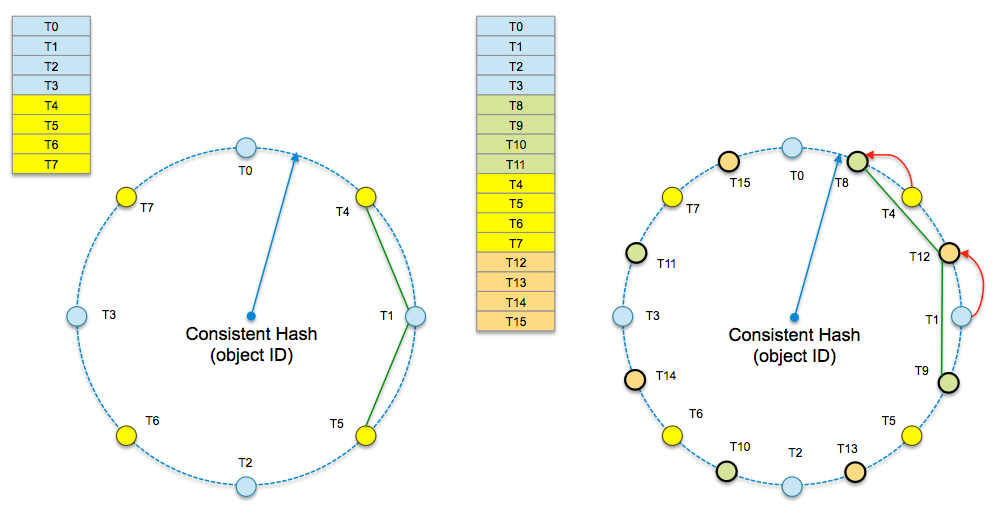
One of the major limits of the single-ring placement map is that it requires all domains to have exactly the same number of targets to build a symmetric ring. It also imposes some restrictions to pool change operations, to avoid reshuffling of all data. The simpler use cases considered here are, either the total number of domains or the total number of targets is doubled. Other attempts to change the DAOS pool will trigger a significant amount of data movement.

**16.2.4.1 Doubling Domains**

As previously mentioned, the ring placement map can be aware of one type of domain. If the number of domains of this type in the DAOS pool is doubled while extending the DAOS

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

76

pool, then ideally, half of all objects should be moved to those new domains. To achieve this goal, targets from those new domains should always be inserted between two original targets on the consistent hashing ring. In this way, the hash distance between any two original targets is unchanged, and hash distance between two new targets is the same as the original targets.

As shown in the figure below, objects placed on the ring should either stay in place or be moved to new targets. If overall objects are evenly distributed on all targets before

extending, then only 50% of the data will migrate.

**Figure 16-5. Doubling domains for a ring placement map**

The shards of objects created before the new domains are added are no longer contiguous on the ring after doubling to avoid having to move all shards except the first one. This may be achieved by keeping the hash stride between object shards constant. However, this necessitates storing the initial hash stride with the object metadata as described in the next section. Note that the distribution of an object will become increasingly sparse on the placement ring with each extending operation.

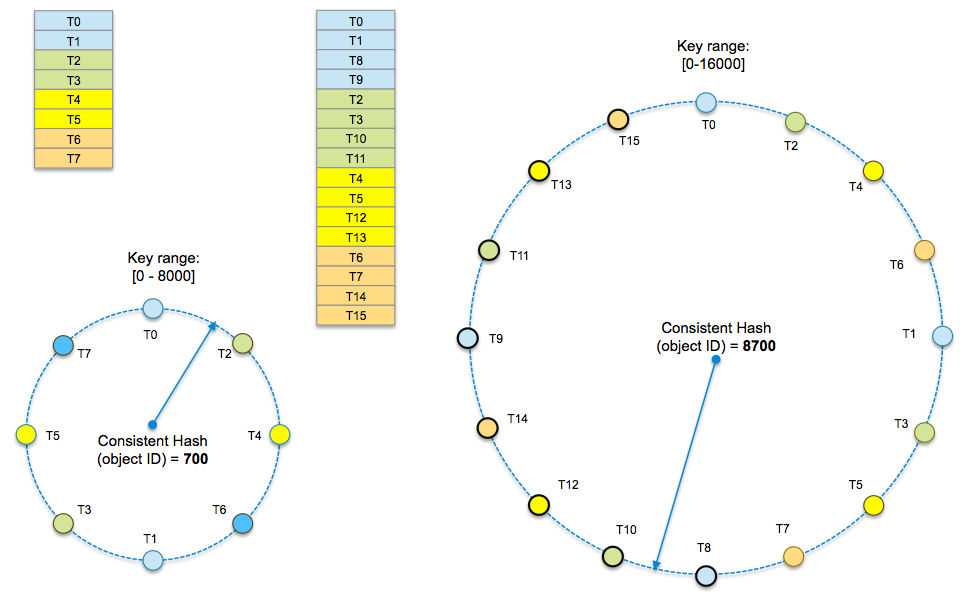
**16.2.4.2 Doubling Targets within Domains**

Another way of symmetrically extending a DAOS pool is by doubling the number of targets within each domain. The number of domains is still the same after doubling, but the hash distance between targets in the same domain becomes ý of the original distance since new targets have to be inserted between all original targets of the same domain. However, at the position for the new target, there could already be a target from a different domain. In the example in the figure below, on the left ring, if a new target is added to the same domain of T0 and T1, to keep the same distance between targets from the same domain, it has to be put on the position of T4 or T5. It means T4 and T5 have to be rotated and objects on them have to migrate to other targets.

To avoid this situation, DAOS-SR doubles the consistent hashing key range. For example, in the figure below, the key range of the hash ring is 8000 before doubling and is increased to16000 after doubling targets. All new targets are placed on the new extended key range of the ring (left half of the large ring). When DAOS-SR places an object by hashing its ID, the hashed result should either be the same, which is 700 in this example or become

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

77

8000+700=8700. With this approach, DAOS-SR can guarantee that only 50% of the data will be moved after the doubling of targets.

**Figure 16-6. Doubling targets within domains**

**16.3 Multi-Ring Placement Map**

As described earlier, although the single ring placement map consumes less memory, it has a few significant drawbacks:

• It can only be built from a DAOS pool that has exactly the same number of targets in all

domains.

• It cannot guarantee even data distribution and reasonable data movement for arbitrary

extending of a DAOS pool.

• It cannot decluster the rebuilding workload on failure; only a limited number of targets

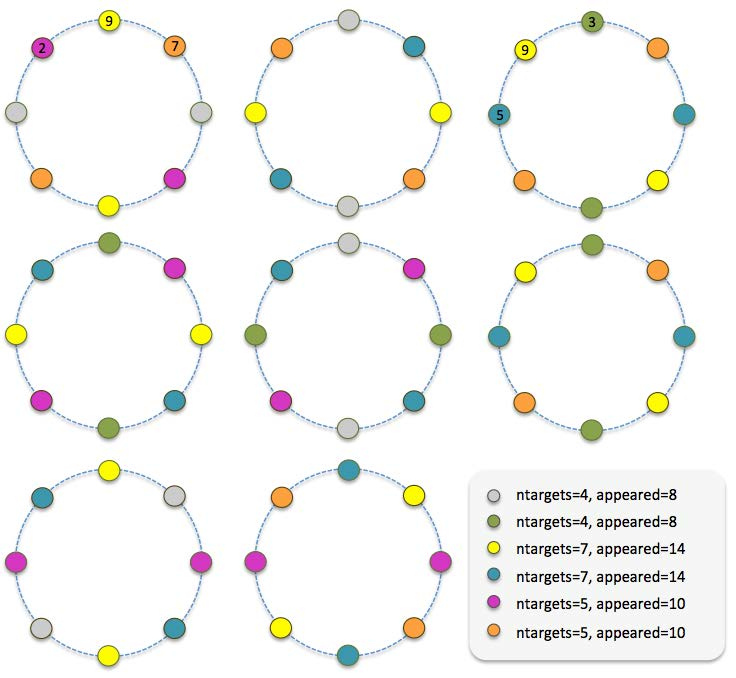
in the distance of redundancy group size can contribute to rebuild.

These drawbacks are not crucial for the NVM based storage tier because it should have extremely high bandwidth, which can support quick data movement; it also has symmetric node allocation for the best of application performance. However, these limits could be an issue for the cold storage tier, which may have spinning disk as storage devices. Spinning disk-based targets have limited bandwidth and could take a long time to rebuild if only a few targets can contribute on any rebuild. Also, for the cold storage tier, doubling is an unlikely way of extending storage, whereas progressively extending is the more likely approach.

Multi-ring placement map can resolve these issues at some level. In a multi-ring placement map, each ring is created similarly as a single-ring placement map, and a ring still includes the same number of targets from each domain. However, because a DAOS pool could have an asymmetric distribution of domain targets, a ring should not include all domains, and it

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

78

does not always select all targets from a domain. Different rings of the placement map should include a different set of domains and targets, giving enough rings, the placement map can make fair use of all targets in the storage pool.

In the example in the figure below, there are six domains, and these domains have a

different number of targets. If all rings are created with all domains, then it is impossible to guarantee that appearances of targets and the hash distances between targets are the same. If rings are created with 4 domains and 2 targets from each domain, then it is possible to build 8\*N rings to guarantee appearances of each target are the same.

**Figure 16-7. Multi-ring placement map**

Given a large enough number of rings, each of them is created by a pseudo-random

algorithm that selects M domains from the total of N domains and selects T targets from each domain.

• Each domain has a weight number that is equal to the number of targets within it, for

example, the weight of the gray domain is 4, and the weight of the yellow domain is 7.

• The domain ring-key is a value in a fixed range that is computed by a pseudo-random

hash algorithm which takes the ring ID as a random seed and the domain ID as hash input. DAOS-SR scales ring-keys of all domains by their weights then sorts the scaled results and selects the first M domains to build a placement ring. Because different rings

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

79

have different random seeds for the hash algorithm, so each of them selects a different set of domains. The domains that have more targets should have a greater chance of being selected because their ring-keys will be scaled by larger weights.

• Similarly, the target ring-key is also a value in a fixed range that is computed by a

pseudo-random hash algorithm which takes the ring ID as a random seed and the target ID as hash input. DAOS-SR sorts all target ring-keys and selects the first T targets for a placement ring.

As more targets are added to a domain, the domain has more chances to be selected while creating placement rings, because it has a higher weight value as the scale factor of the domain ring-key. The extended domain will, therefore, replace other domains on a relatively few rings, and data movement for rebalancing is reduced because only those domains being replaced have to move objects to the domain with new targets.

Also, when adding a new target, if its target ring-key is higher than all other targets in the same domain, and if DAOS-SR uses the sorted target list to build a ring, then all other targets will be shifted because the new target will be put at the head of the target list. For example, in a pool map version 1, DAOS-SR needs to select 6 targets out from 8 targets in a domain, after sorting them by their ring-keys, assuming the selected targets are:

3(v=1), 1(v=1), 7(v=1), 5(v=1), 2(v=1), 4(v=1)

Another 2 targets (target-8 and target-9) were added to this domain in version 2, and target-8 has the highest ring-key, so the selected targets will be:

8(v=2), 3(v=1), 1(v=1), 7(v=1), 5(v=1), 2(v=1)

Positions for the rest of the 5 targets are changed on the ring, so objects residing on these targets will be reshuffled, which is undesirable. This situation can be avoided by the

following approach:

Targets are still selected by the sorted target ring-key, but the output list is sorted again by version and ring-key where the version is the major key for sort. Targets with lower version number take precedence over targets with a higher version number, so in the previous example, the output list becomes:

3(v=1), 1(v=1), 7(v=1), 5(v=1), 2(v=1), 8(v=2)

In this approach, only objects on target-2 are moved to target-8.

**16.3.1 Rebuild Declustering**

In a multi-ring placement map, targets have different neighbors on the different rings. If one target fails, all neighbors on all rings can contribute to data rebuild, which can

significantly reduce the time of recovery.

In the previous figure, neighbors of target-9 are target-2 and target-7 on the first ring and target-3 and target-5 on the third ring. Assuming all objects are 2-way replicated, if target-9 fails, then these four targets can contribute to rebuild data for target-9 because they all have replicas for objects on target-9.

**16.3.2 Widely Striped Object**

In the example in the previous figure, all 8 rings have 8 targets. An object cannot,

therefore, have more than 8 shards, and applications cannot efficiently use the full

bandwidth of the storage cluster for large objects. To resolve this problem, a multi-ring placement map may have rings with different sizes. For example, it may include a few large rings besides hundreds or thousands of small rings.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

80

In the above example, DAOS-SR may create a few large rings with 24 targets; each of these rings has 4 targets from all 8 domains. Only widely-striped objects will be placed on these rings to achieve better I/O concurrency.

Furthermore, while adding more and more targets to the DAOS pool over time, DAOS-SR can create even larger rings. Rings that were originally large could become small because they can only cover a small set of targets in the DAOS pool, and the original small rings could be eliminated gradually.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

81

**17 Self-healing (Rebuild)**

In DAOS, if the data is replicated with multiple copies on different targets, once one of the targets failed, the data on it will be rebuilt on the other targets automatically, so the data redundancy will not be impacted due to the target failure. In the future, DAOS will also support Erasure Coding to protect the data; then the rebuild process might be updated accordingly.

**17.1 Rebuild Detection**

When a target failed, it should be detected promptly and notify the pool (Raft) leader; then the leader will exclude the target from the pool and trigger the rebuild process immediately.

**17.1.1 Current status and long-term goal**

Currently, since the raft leader cannot exclude the target automatically, the sysadmin has to manually exclude the target from the pool, which then triggers the rebuild.

In the future, the leader should be able to detect the target failure promptly and then trigger the rebuild automatically by itself, without the help of sysadmin.

**17.2 Rebuild process**

The rebuild is divided into 2 phases, scan and pull.

**17.2.1 Scan Phase**

Initially, the leader will propagate the failure notification to all other surviving targets by a collective RPC. Any target that receives this RPC will start to scan its object table to

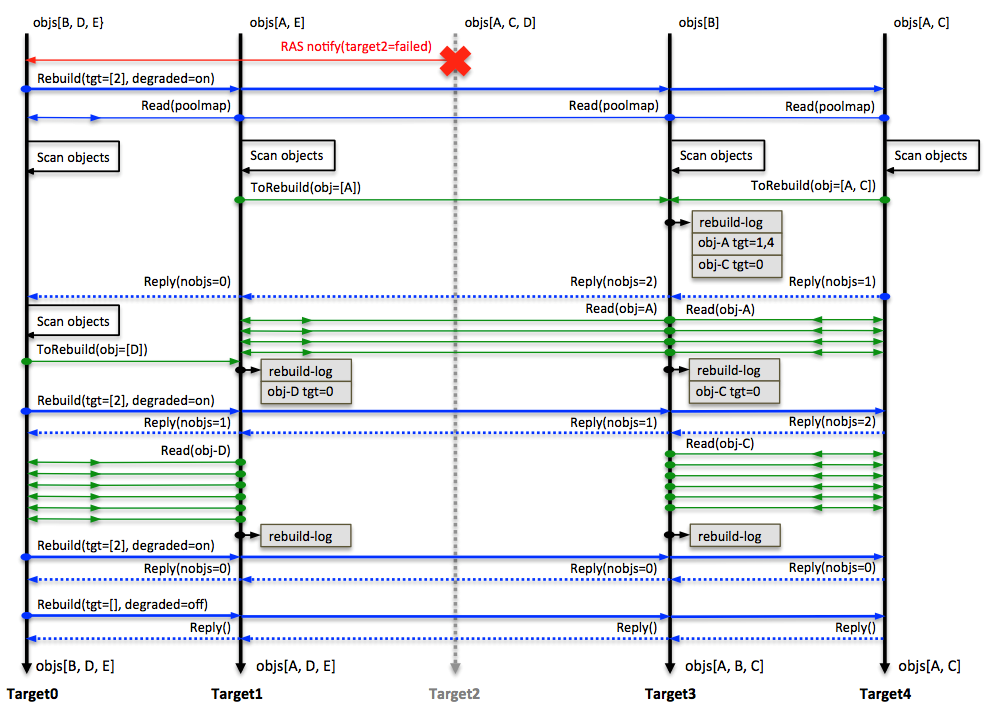
determine the object loss data redundancy on the faulty target, if it does, then send their IDs and related metadata to the rebuild targets(rebuild initiator). As for how to choose the rebuild target for the faulty target, it will be described in placement/README.md.

**17.2.2 Pull Phase**

Once the rebuild initiators get the object list from the scanning target, it will pull the data of these objects from other replicas and then write data locally. Each target will report its rebuild status, rebuilding objects, records, completion status, etc, to the pool leader. Once the leader learned all of the targets finished its scanning and rebuilding phase, it will notify all targets the rebuild has finished, and they can release all of the resources hold during the rebuild process.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

82

**Figure 17-1. Rebuild Protocol**

The figure above is an example of this process: There are five objects in the cluster: object A is 3-way replicated, object B, C, D, and E are 2-way replicated. When target-2 failed,target-0, which is the Raft leader, broadcasted the failure to all surviving targets to notify them to enter the degraded mode and scan:

• Target-0 found that object D lost a replica and calculated out target-1 is the rebuild

target for D, so it sent object Dâs ID and its metadata to target-1.

• Target-1 found that object A lost a replica and calculated out target-3 is the rebuild

target for A, so it sent object Aâs ID and its metadata to target-3.

• Target-4 found objects A and C lost replicas and it calculated out target-3 is the rebuild

target for both objects A and C, so it sent IDs for objects A and C and their metadata to target-3.

• After receiving these object IDs and their metadata, target-1 and target-3 can compute

out surviving replicas of these objects, and rebuild these objects by pulling data from these replicas.

**17.2.3 Multiple pool and targets rebuild**

In a large-scale storage cluster, multiple failures might occur when a rebuild from a

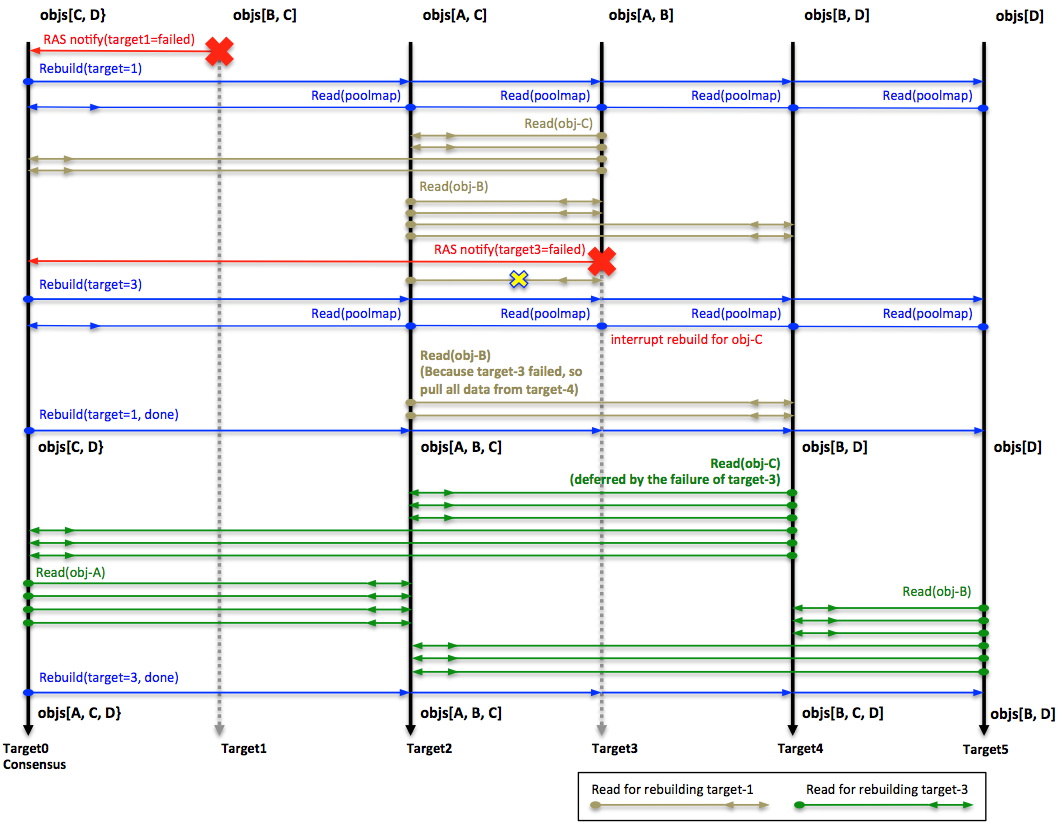
previous failure is still in progress. In this case, DAOS should neither simultaneously handle these failures, nor interrupt and reset the earlier rebuilding progress for later failures.Otherwise, the time consumed for rebuilds for each failure might grow significantly and rebuilds may never end if new failures overlap with ongoing rebuilds. So for multiple failures, these rules are applied

• If the rebuild initiator fails during the rebuild, then the object shards being rebuilt on the

initiator should be ignored, which will be handled by the next rebuild.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

83

• If rebuild initiator cannot fetch the data from other replicas due to the failure, it will

switch to other replicas if available.

• A target in a rebuild does not need to re-scan its objects or reset rebuild progress for the

current failure if another failure has occurred.

• When there are multiple failures, if the number of failed targets from different domains

exceeds the fault tolerance level, then there could be unrecoverable errors and

applications could suffer from data loss. In this case, upper layer stack software could see errors while sending I/O to the object that could have missing data.

The following figure is an example of this protocol.

**Figure 17-2. Multi-failure protocol**

• In this example, object A is 2-way replicated, object B, C and D are 3-way replicated.

• After the failure of target-1, target-2 is the initiator of rebuilding object B; it is pulling

data from target-3 and target-4; target-3 is the initiator of rebuilding object C, it is pulling data from target-0 and target-2.

• Target-3 failed before completing rebuild for target-1, so rebuild of object C should be

abandoned at this point, because target-3 is the initiator of it. The missing data

redundancy of object C will be reconstructed while rebuilding target-3.

• Because target-3 is also a contributor to rebuilding object B, based on the protocol, the

initiator of object B, which is target-2, should switch to target-4 and continue rebuild of object B.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

84

• Rebuild process of target-1 can complete after finishing rebuild of object B. By this time,

object C still lost a replica. This is quite understandable because even if two failures have no overlap, object C will still lose the replica on target-3.

• In the process of rebuilding target-3, target-4 is the new initiator of rebuilding object C.

If there are multiple pools being impacted by the failing target, these pools can be rebuilt concurrently.

**17.3 I/O during the rebuild**

If there are concurrent writes during the rebuild, the rebuild protocol should guarantee that new writes will never be lost. Those writes should be either directly stored in the new object shard or pulled to the new object shard by the rebuild initiator. Moreover, also fetch should be guaranteed to get the correct data. To achieve these, these protocols are applied

1. Fetch will always skip the rebuilding target.

2. The update can complete only if updates of all the object shards have successfully

completed.

a. If any of these updates failed, the client would infinitely retry until it succeeds, or

there is a pool map change which shows the target failed. In the second case, the client will switch to the new pool map, and send the update to the new destination,which is the rebuild target of this object.

3. There is no synchronization between normal I/O and rebuild process, so during the

rebuild process, data writes might be duplicated by the rebuild initiator and normal I/O.

**17.4 Rebuild resource throttle**

During the rebuild process, the user can set the throttle to guarantee the rebuild will not use more resource than the user setting. The user can only set the CPU cycle for now. For example, if the user set the throttle to 50, then the rebuild will at most use 50% of the CPU cycle to do the rebuild job. The default rebuild throttle for CPU cycle is 30.

**17.5 Rebuild status**

As described earlier, each target will report its rebuild status to the pool leader by IV; then the leader will summarize the status of all targets, and print out the whole rebuild status by every 2 seconds, for example, these messages.

Rebuild [started] (pool 8799e471 ver=41)

Rebuild [scanning] (pool 8799e471 ver=41, toberb\_obj=0, rb\_obj=0, rec= 0, done 0 stat us 0 duration=0 secs)

Rebuild [queued] (419d9c11 ver=2)

Rebuild [started] (pool 419d9c11 ver=2)

Rebuild [scanning] (pool 419d9c11 ver=2, toberb\_obj=0, rb\_obj=0, rec= 0, done 0 statu s 0 duration=0 secs)

Rebuild [pulling] (pool 8799e471 ver=41, toberb\_obj=75, rb\_obj=75, rec= 11937, done 0status 0 duration=10 secs)

Rebuild [completed] (pool 419d9c11 ver=2, toberb\_obj=10, rb\_obj=10, rec= 1026, done 1status 0 duration=8 secs)

Rebuild [completed] (pool 8799e471 ver=41, toberb\_obj=75, rb\_obj=75, rec= 13184, done1 status 0 duration=14 secs)

There are 2 pools being rebuilt (pool 8799e471 and pool 419d9c11, note: only first 8 letters of the pool uuid are shown here).

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

re

};

85

The 1st line means the rebuild for pool 8799e471 is started, whose pool map version i s 41.

The 2nd line means the rebuild for pool 8799e471 is in scanning phase, and no objects& records are being rebuilt yet.

The 3rd line means a rebuild job for pool 419d9c11 is being queued.

The 4th line means the rebuild for pool 419d9c11 is started, whose pool map version i s 2.

The 5th line means the rebuild for pool 419d9c11 is in scanning phase, and no objects& records are being rebuilt yet.

The 6th line means the rebuild for pool 8799e471 is in pulling phase, and there are 75 objects to be rebuilt(toberb\_obj=75), and all of them are rebuilt(rb\_obj=75), but r ecords rebuilt for these objects are not finished yet(done 0) and only 11937 records(rec = 11937) are rebuilt.

The 7th line means the rebuild for pool 419d9c11 is done (done 1), and there are tota lly 10 objects and 1026 records are rebuilt, which costs about 8 seconds.

The 8th line means the rebuild for pool 8799e471 is done (done 1), and there are tota lly 75 objects and 13184 records are rebuilt, which costs about 14 seconds.

During the rebuild, if the client query the pool status to the pool leader, which will return its rebuild status to client as well.

**struct** daos\_rebuild\_status {

/\*\* pool map version in rebuilding or last completed rebuild \*/

uint32\_t rs\_version;

/\*\* padding bytes \*/

uint32\_t rs\_pad\_32;

/\*\* errno for rebuild failure \*/

int32\_t rs\_errno;

/\*\*

\* rebuild is done or not, it is valid only if **@rs\_version** is non-zer

o

\*/

int32\_t rs\_done;

/\*\* # total to-be-rebuilt objects, it's non-zero and increase when

\* rebuilding in progress, when rs\_done is 1 it will not change anymo\* and should equal to rs\_obj\_nr. With both rs\_toberb\_obj\_nr and\* rs\_obj\_nr the user can know the progress of the rebuilding.

\*/

uint64\_t rs\_toberb\_obj\_nr;

/\*\* # rebuilt objects, it's non-zero only if rs\_done is 1 \*/

uint64\_t rs\_obj\_nr;

/\*\* # rebuilt records, it's non-zero only if rs\_done is 1 \*/

uint64\_t rs\_rec\_nr;

**17.6 Rebuild failure**

If the rebuild fails, it will be aborted, and the related message will be shown on the leader console. For example:

Rebuild [aborted] (pool 8799e471 ver=41, toberb\_obj=75, rb\_obj=75, rec= 11937, done 1status 0 duration=10 secs)

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

86

**18 DAOS Common Libraries**

Common functionality and infrastructure shared across all DAOS components are provided in external shared libraries. This includes the following features:

• Hash and checksum routines

• Event and event queue support for non-blocking operations

• Logging and debugging infrastructure

• Locking primitives

• Network transport

**18.1 Task Scheduler Engine (TSE)**

The TSE is a generic library to create generic tasks with function callbacks, optionally add dependencies between those tasks, and schedule them in an engine that is progressed to execute those tasks in an order determined by a dependency graph in which they were inserted in. The task dependency graph is an integral part of the scheduler to allow users to create several tasks and progress them in a non-blocking manner.

The TSE is not DAOS specific, but used to be part of the DAOS core and was later extracted into the common src as a standalone API. The API is generic and allows the creation of tasks in an engine without any DAOS specific functionality. The DAOS library does provide a task API that is built on top of the TSE. For more information on that see here. Furthermore,DAOS uses the TSE internally to track and progress all API tasks that are associated with the API event and, in some cases, to schedule several inflight "child" tasks corresponding to a single API task and add a dependency on that task to track all those inflight "child" tasks.An example of that would be the Array API in the DAOS addons library and the object update with multiple replicas.

**18.1.1 Scheduler API**

The scheduler API allows a user to create a generic scheduler and add tasks to it. At the time of scheduler creation, the user can register a completion callback to be called when the scheduler is finalized.

The tasks that are added to the scheduler do not progress on their own. There have to be explicit calls to a progress function (daos\_sched\_progress) on the scheduler to make progress on the tasks in the engine. This progress function can be called by the user

occasionally in their program, or a single thread can be forked that calls the progress function repeatedly.

**18.1.2 Task API**

The task API allows the creation of tasks with generic body functions and adding them to a scheduler. Once a task is created within a scheduler, it will not be scheduled to run without an explicit call from the user to the task schedule function unless it's part of a task

dependency graph where in this case the explicit schedule call is required only to the first task in the graph. After creating the task, the user can register any number of dependencies for the task that would be required to complete before the task can be scheduled to run.Also, the user will be able to register preparation and completion callback on the task:

• Preparation Callbacks are executed when the task is ready to run but has not been

executed yet, meaning the dependencies that the task was created with are done and the scheduler is ready to schedule the task. This is useful when the task to be scheduled needs information that is not available at the time of task creation but will be available

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

87

after the dependencies of the task complete; for example, setting some input

parameters for the task body function.

• Completion Callbacks are executed when the task is finished executing, and the user

needs to do more work or handling when that happens. An example where this would be useful is setting the completion of a higher level event or request that is built on top of the TSE, or to track error status of multiple tasks in a dependency list.

Several other functionalities on the task API exists to support:

• setting some private data on the task itself that can be queried.

• pushing and popping data on/from task stack space without data copy

• generic task lists

More detail about that functionality can be found in the TSE header in the DAOS code here.

**18.2 dRPC C API**

For a general overview of dRPC concepts and the corresponding Golang API, see here.In the C API, an active dRPC connection is represented by a pointer to a context object(struct drpc). The context supplies all the state information required to communicate over the Unix Domain Socket. When finished with a context, the object should be freed by using drpc\_close().

dRPC calls and responses are represented by the Protobuf-generated structures Drpc\_\_Call and Drpc\_\_Response.

**18.2.1 C Client**

Connecting to a valid Unix Domain Socket returns a dRPC context, which can be used to execute any number of dRPC calls to the server that set up the socket.

**Note:** Currently synchronous calls (using flag R\_SYNC) are the only type supported.

Asynchronous calls receive an instantaneous response but are never truly processed.

**18.2.1.1 Basic Client Workflow**

1. Open a connection to the server's Unix Domain Socket:

struct drpc \*ctx = drpc\_connect("/var/run/my\_socket.sock");

2. Send a dRPC call:

Drpc\_\_Call \*call;

/\* Alloc and set up your Drpc\_\_Call \*/

Drpc\_\_Response \*resp = NULL; /\* Response will be allocated by drpc\_call \*/int result = drpc\_call(ctx, R\_SYNC, call, &resp);

o An error code returned from drpc\_call() indicates that the message could not be

sent, or there was no response. If drpc\_call() returned success, the content of the response still needs to be checked for errors returned from the server.

3. Send as many dRPC calls as desired.

4. When finished with the connection, close it: drpc\_close(ctx); Note: After drpc\_close() is

called, the dRPC context pointer has been freed and is no longer valid.

**18.2.2 C Server**

The dRPC server sets up a Unix Domain Socket and begins listening on it for client

connections. In general, this means creating a listening dRPC context to detect any

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

88

incoming connections. Then, when a client connects, a new dRPC context is generated for that specific session. The session context is the one that sends and receives data. It is possible to have multiple session contexts open at the same time.

The socket is always set up as non-blocking, so it is necessary to poll for activity on the context's file descriptor (ctx->comm->fd) using a system call like poll() or select() on POSIX-compliant systems. This applies not only to the listening dRPC context but also to any dRPC context generated for a specific client session.

The server flow is dependent on a custom handler function, whose job is to dispatch

incoming Drpc\_\_Call messages appropriately. The handler function should inspect the module and method IDs, ensure that the desired method is executed, and create a

Drpc\_\_Response based on the results.

**18.2.2.1 Basic Server Workflow**

1. Set up the Unix Domain Socket at a given path and create a listening context using a

custom handler function:

void my\_handler(Drpc\_\_Call \*call, Drpc\_\_Response \*\*resp) {

/\* Handle the message based on module/method IDs \*/

}

...

struct drpc \*listener\_ctx = drpc\_listen("/var/run/drpc\_socket.sock", my\_handler);

2. Poll on the listener context's file descriptor (listener\_ctx->comm->fd).

3. On incoming activity, accept the connection:

struct drpc \*session\_ctx = drpc\_accept(listener\_ctx);

This creates a session context for the specific client. All of that client's communications will come over the session context.

4. Poll on the session context's file descriptor (session\_ctx->comm->fd) for incoming data.5. On incoming data, handle the message:

int result = drpc\_recv(session\_ctx);

This unmarshals the incoming Drpc\_\_Call, calls the handler function defined when the listener context was created and sends the Drpc\_\_Response back over the channel to the client. An error will only be returned if the incoming data is not a Drpc\_\_Call or the handler failed to allocate a response (for example, if the system is out of memory).

6. If the client has closed the connection, close the session context to free the pointer:

drpc\_close(session\_ctx);

7. When it's time to shut down the server, close any open session contexts, as noted

above. Then drpc\_close() the listener context.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

89

**19 DAOS Client Library & I/O Middleware**

This section contains the following client and middleware topics:

• DAOS Library (libdaos) and addons

• Python API bindings

• Go bindings and API documentation

• POSIX File & Directory Emulation (libdfs)

**19.1 DAOS Client Library**

The DAOS API is divided along several functionalities to address the different features that DAOS exposes:

• Management API: pool and target management

• Pool Client API: pool access

• Container API: container management and access, container snapshots

• Transaction API: transaction model and concurrency control

• Object API: object and data management and access

• Event, Event Queue, and Task API: non-blocking operations

• Addons API: array and KV operations built over the DAOS object API

• DFS API: DAOS file system API to emulate a POSIX namespace over DAOS

• DUNS API: DAOS unified namespace API for integration with an existing system

namespace.

Each of those components has associated README.md files that provide more details about the functionality they support except for APIs to support non-blocking operations which are discussed here.

The libdaos API is available under /src/include/daos\_\* and associated man pages under https://github.com/daos-stack/daos/blob/master/doc/man/man3.

**19.1.1 Event & Event Queue**

DAOS API functions can be used in either blocking or non-blocking mode. This is determined through a pointer to a DAOS event passed to each API call that:

• If NULL indicates that the operation is to be blocking. The operation will return after

completing the operation. The error codes for all failure cases will be returned through the return code of the API function itself.

• If a valid event is used, the operation will run in non-blocking mode and return

immediately after scheduling the operation in the internal scheduler and after RPCs are submitted to the underlying stack. The return value of the operation is success if the scheduling succeeds, but does not indicate that the actual operation succeeds. The errors that can be caught on return are either invalid parameters or scheduling

problems. The actual return code of the operation will be available in the event error code (event.ev\_error) when the event completes.

A valid event to be used must be created first with a separate API call. To allow users to track multiple events at a time, an event can be created as part of an event queue, which is a collection of events that can be progressed and polled together. Alternatively, an event can be created without an event queue, and be tracked individually. Once an event is

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

90

completed, it can be re-used for another DAOS API call to minimize the need for event creation and allocations inside the DAOS library.

**19.1.2 Task Engine Integration**

The DAOS Task API provides an alternative way to use the DAOS API in a non-blocking manner and at the same time build a task dependency tree between DAOS API operation.This is useful for applications and middleware libraries using DAOS and needing to build a schedule of DAOS operations with dependencies between each other (N-1, 1-N, N-N).To leverage the task API, the user would need to create a scheduler where DAOS tasks can be created as a part of. The task API is generic enough to allow the user to mix DAOS specific tasks (through the DAOS task API) and other user-defined tasks and add

dependencies between those.

For more details on how TSE is used in the client library, see TSE internals documentation for more details.

**19.2 DAOS Addons**

The DAOS addons constitute APIs that are built on top of the DAOS API. The purpose is to provide some simplified or additional functionality that can be achieved with the current API and would be commonly used by several applications and middleware libraries to warrant a common implementation within the DAOS library to avoid multiple implementations

externally.

Currently, the DAOS addons include:

• An Array object and API built on top of the DAOS Key-Array object.

• A simplified DAOS Key Value API.

**19.2.1 DAOS Arrays**

A DAOS Array is a special DAOS object to expose a logical 1-dimensional array to the user.The array is created by the user with an immutable record size and chunk size. Additional APIs are provided to access the array (read, write, punch).

**19.2.1.1 Array Representation**

The Array representation over the DAOS KV API is done with integer typed DKeys, where each DKey holds chunk\_size records. Each DKey has 1 AKey with a NULL value that holds the user array data in an array type extent. The only exception is the first DKey (which is 0)that holds a special AKey to hold the array metadata:

DKey: 0

AKey: "daos\_array\_metadata"

Single Value: 3 uint64\_t

[0] = magic value (0xdaca55a9daca55a9)

[1] = array cell size

[2] = array chunk size

To illustrate the array mapping, suppose we have a logical array of 10 elements and chunk size being 3. The DAOS KV representation would be:

DKey: 0

Array records: 0, 1, 2

DKey: 1

Array records: 3, 4, 5

DKey: 2

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

Array records: 6, 7, 8

DKey: 3

Array records: 9

91

**19.2.1.2 API and Implementation**

The API (include/daos\_addons.h) includes operations to:

• create an array with the required, immutable metadata of the array.

• open an existing array which returns the metadata associated with the array.

• read from an array object.

• write to an array object.

• set size (truncate) of an array. Note this is not equivalent to pre-allocation.

• get size of an array.

• punch a range of records from the array.

• destroy/remove an array.

The Array API is implemented using the DAOS Task API. For example, the read and write operations create an I/O operation for each DKey and inserts them into the task engine with a parent task that depends on all the child tasks that do the I/O.

The API is currently tested with daos\_addons\_test.

**19.2.2 DAOS High Level (HL) API**

The HL API simplifies the DAOS Object API and exposes a simple API to manipulate a Key-Value object with simple put/get/remove operations. The API exposes only a single Key (no multi-level keys) and a value associated with that key which is overwritten entirely anytime the key is updated. So internally the mapping of the HL KV object looks like:

Key -> DKey

NULL AKEY

Value -> Single Value

The API provides three functions to access the DAOS object:

int daos\_kv\_put(daos\_handle\_t oh, daos\_handle\_t th, const char \*key,

daos\_size\_t size, const void \*buf, daos\_event\_t \*ev);

int daos\_kv\_get(daos\_handle\_t oh, daos\_handle\_t th, const char \*key,

daos\_size\_t \*size, void \*buf, daos\_event\_t \*ev);

int daos\_kv\_remove(daos\_handle\_t oh, daos\_handle\_t th, const char \*key,

daos\_event\_t \*ev);

The API is currently tested with daos\_addons\_test.

**19.3 Python Bindings for DAOS API**

**19.3.1 The motivation for Python API**

The Python API for DAOS provides access to DAOS API functionality with an emphasis on test use cases. While the majority of unit tests are written in C, higher-level tests are written primarily using the Python API. Interfaces are provided for accessing DAOS

management and DAOS API functionality from Python. This higher level interface allows a faster turnaround time on implementing test cases for DAOS.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

92

**19.3.2 Architecture**

**19.3.2.1 Layout**

The Python API is split into several files based on functionality:

• The Python object API, located in the source tree at src/utils/py/daos\_api.py.

• The mapping of C structures to Python classes, located in the source tree at

src/utils/py/daos\_cref.py

High-level abstraction classes exist to manipulate DAOS storage:

class DaosPool(object)

class DaosContainer(object)

class DaosObj(object)

class IORequest(object)

DaosPool is a Python class representing a DAOS pool. All pool-related functionality is exposed from this class. Operations such as creating/destroying a pool, connecting to a pool, and adding a target to a storage pool are supported.

DaosContainer is a Python class representing a DAOS container. As with the DaosPool class, all container-related functionality is exposed here. This class also exposes abstracted wrapper functions for the flow of creating and committing an object to a DAOS container.DaosObj is a Python class representing a DAOS object. Functionality such as

creating/deleting objects in a container, 'punching' objects (delete an object from the specified transaction only), and object query.

IORequest is a Python class representing a read or write request against a DAOS object.Several classes exist for management purposes as well:

class DaosContext(object)

class DaosLog

class DaosApiError(Exception)

DaosContext is a wrapper around the DAOS libraries. It is initialized with the path where DAOS libraries can be found.

DaosLog exposes functionality to write messages to the DAOS client log.

DaosApiError is a custom exception class raised by the API internally in the event of a failed DAOS action.

Most functions exposed in the DAOS C API support both synchronous and asynchronous execution, and the Python API exposes this same functionality. Each API takes an input event. DaosEvent is the Python representation of this event. If the input event is NULL, the call is synchronous. If an event is supplied, the function will return immediately after submitting API requests to the underlying stack and the user can poll and query the event for completion.

**19.3.2.2 Ctypes**

Ctypes is a built-in Python module for interfacing Python with existing libraries written in C/C++. The Python API is built as an object-oriented wrapper around the DAOS libraries utilizing ctypes.

Ctypes documentation can be found here https://docs.python.org/3/library/ctypes.html

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

93

The following demonstrates a simplified example of creating a Python wrapper for the C function daos\_pool\_tgt\_exclude\_out, with each input parameter to the C function being cast via ctypes. This also demonstrates struct representation via ctypes:

// daos\_exclude.c

#include <stdio.h>

int

daos\_pool\_tgt\_exclude\_out(const uuid\_t uuid, const char \*grp,

const d\_rank\_list\_t \*svc, struct d\_tgt\_list \*tgts,daos\_event\_t \*ev);

All input parameters must be represented via ctypes. If a struct is required as an input parameter, a corresponding Python class can be created. For struct d\_tgt\_list:

struct d\_tgt\_list {

d\_rank\_t \*tl\_ranks;

int32\_t \*tl\_tgts;

uint32\_t tl\_nr;

};

class DTgtList(ctypes.Structure):

\_fields\_ = [("tl\_ranks", ctypes.POINTER(ctypes.c\_uint32)),

("tl\_tgts", ctypes.POINTER(ctypes.c\_int32)),

("tl\_nr", ctypes.c\_uint32)]

The shared object containing daos\_pool\_tgt\_exclude\_out can then be imported and the function called directly:

# api.py

import ctypes

import uuid

import conversion # utility library to convert C <---> Python UUIDs

# init python variables

p\_uuid = str(uuid.uuid4())

p\_tgts = 2

p\_ranks = DaosPool.\_\_pylist\_to\_array([2])

# cast python variables via ctypes as necessary

c\_uuid = str\_to\_c\_uuid(p\_uuid)

c\_grp = ctypes.create\_string\_buffer(b"daos\_group\_name")

c\_svc = ctypes.POINTER(2) # ensure pointers are cast/passed as such

c\_tgt\_list = ctypes.POINTER(DTgtList(p\_ranks, p\_tgts, 2))) # again, DTgtList must be passed as pointer

# load the shared object

my\_lib = ctypes.CDLL('/full/path/to/daos\_exclude.so')

# now call it

my\_lib.daos\_pool\_tgt\_exclude\_out(c\_uuid, c\_grp, c\_svc, c\_tgt\_list, None)

**19.3.2.3 Error Handling**

The API was designed using the EAFP (Easier to Ask Forgiveness than get Permission)idiom. A given function will raise a custom exception on error state, DaosApiError. A user of the API is expected to catch and handle this exception as needed:

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

94

# catch and log

try:

daos\_some\_action()

except DaosApiError as e:

self.d\_log.ERROR("My DAOS action encountered an error!")

**19.3.3 Usage**

**19.3.3.1 Python API Usage in Tests**

The following example demonstrates a simple use case for the Python API for DAOS,creating and connecting to a pool, creating a container within the pool, and inserting a single value:

# initialize DAOS environmental context

with open('../../../.build\_vars.json') as f:

data = json.load(f)

context = DaosContext(data['PREFIX'] + '/lib/')

print("Initialized!")

# create a DAOS pool

pool = DaosPool(context)

pool.create(448, os.getuid(), os.getgid(), 1024 \* 1024 \* 1024,

b'daos\_server')

print("Pool UUID is {0}".format(pool.get\_uuid\_str()))

# connect to it

pool.connect(1 << 1)

# query the pool

pool\_info = pool.pool\_query()

print("Pool has {0} storage targets".format(pool\_info.pi\_ntargets))

print("Pool created with {0} permissions".format(pool\_info.pi\_mode))

# create a container in the pool and open it

container = DaosContainer(context)

container.create(pool.handle)

container.open()

# prep an object to write to the container

thedata = "data to write to this object"

size = 28

dkey = "this is the dkey"

akey = "this is the akey"

# write it

obj, tx = container.write\_an\_obj(thedata, size, dkey, akey, None, 5)

**19.3.3.2 Changing the API**

Extending DAOS Python API

Once a function has been added to the DAOS C API, it must be represented in the Python API. In the following example, the function table is extended to reference a new C API function hello\_world(), and a corresponding Python function is created.

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

}

# call it

rc = func(a, b)

if rc != 0:

95

1. A C function is added to the DAOS API:

void

hello\_world(int \*a, int b);

2. The C function is added to the function table in the DaosContext class in daos\_api.py:class DaosContext(object):

def \_\_init\_\_(self, path):

# table defining relationship between Python and C function calls self.ftable = {

'create-pool' : self.libdaos.daos\_pool\_create,

'hello-world' : self.libdaos.daos\_hello\_world # this is the

new function

3. A corresponding Python function is added in daos\_api.py. Consideration must be given

to whether the added function is an operation on an existing Python class or if a new class must be created:

# a corresponding hello\_world Python API function is added

def hello\_world(self):

# retrieve new function from function table

func = self.context.get\_function('hello-world')

# ensure arguments passed are of the correct type

a = ctypes.byref(1)

b = ctypes.c\_int(2)

raise DaosApiError("function hello\_world encountered an error!")

C API Modifications

If an existing function is modified, a corresponding update must be made to the Python API.For example, if the member foo of my\_struct were to change from int foo to int \*foo,the Python class representing my\_struct must also be modified:

typedef struct {

int \*foo;

} my\_struct;

class MyStruct(ctypes.Structure):

# field 'foo' updated to correctly cast as pointer vs. int

\_fields\_ = [("foo", ctypes.POINTER(ctypes.c\_int))]

Similarly, if existing APIs add or remove an input parameter, the relevant parameters must be modified in the respective Python APIs.

C API Removal

If an existing C API is removed, the corresponding Python function must also be removed.

**19.3.3.3 Logging**

The Python DAOS API exposes functionality to log messages to the DAOS client log.

Messages can be logged as INFO, DEBUG, WARN, or ERR log levels. The DAOS log object must be initialized with the environmental context in which to run:

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

96

from daos\_api import DaosLog

self.d\_log = DaosLog(self.context)

self.d\_log.INFO("FYI")

self.d\_log.DEBUG("Debugging code")

self.d\_log.WARNING("Be aware, may be issues")

self.d\_log.ERROR("Something went very wrong")

**19.4 Go language bindings for the DAOS API**

This is a Go interface for DAOS, which is also a work in progress. Building this requires a local DAOS build and DAOS server running, so start there first.

**19.4.1 Current Status**

• Covers most of the DAOS API, even the unimplemented bits (e.g., punch, etc.)

• daosfs implements basic POSIX filesystem semantics

• The daos-fuse utility exposes a daosfs container via FUSE mount

• libdaosfs exports a C-compatible API for use by things like nfs-ganesha

**19.4.2 How to Build**

This is a Go project, so a Go development tools are naturally required. We recommend the most current Go release available. As of April 2017, the project has been built and tested with Go 1.8.

Setup environment and build. This assumes $daospath is set as it was in the DAOS README export GOPATH=$HOME/go

export LD\_LIBRARY\_PATH=${daospath}/install/lib

export PATH=${daospath}/install/bin:$PATH

export CGO\_CPPFLAGS="-I${daospath}/install/include -I${daospath}/src/include"

export CGO\_LDFLAGS=-L${daospath}/install/lib

go get -u github.com/daos-stack/go-daos/cmd/dcmd

The last command will clone this repo and its dependencies, build, and install in the Go workspace. The dcmd binary will be in $GOPATH/bin/dcmd, and the source for the package will be $GOPATH/src/github.com/daos-stack/go-daos. You can build again using:

go install github.com/daos-stack/go-daos/cmd/dcmd

**19.4.3 dcmd utility**

Also includes a simple CLI tool for interacting with DAOS. This tool requires the orterun command that was built for DAOS to be on the command line and will handle running itself with orterun if is run directly. When dcmd calls orterun itself, the default uri file is

/tmp/daos-uri or it can be customized with --uri option. See command help for more options and command details.

Example comands to create an container and manipulate objects:

cont=$(uuidgen)

export DAOS\_GROUP="" # if needed

export DAOS\_POOL=$(dcmd pool create)

dcmd cont create --name mydb

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

97

dcmd object update --cont mydb --dkey "a" --akey "1" --value "foo"

dcmd object update --cont mydb --dkey "a" --akey "2" --file /path/to/file

dcmd object fetch --cont mydb --dkey "a" --akey "1"

dcmd object fetch --cont mydb --dkey "a" --akey "2" --binary | sha1sum

By default, fetch will display binary data in "hexdump -C" format. Use the --binary option to read the raw data.

N.B. dcmd does not include a trailing NUL in the string keys and assumes existing keys do not have them, either.

**19.5 DFS Overview**

DFS stands for DAOS File System. The DFS API provides an encapsulated namespace with a POSIX like API directly on top of the DAOS API. The namespace is encapsulated under a single DAOS container where directories and files are objects in that container.

The encapsulated namespace will be located in one DAOS Pool and a single DAOS

Container. The user provides a valid (connected) pool handle and an open container handle where the namespace will be located.

**19.5.1 DFS Namespace**

When the file system is created (i.e., when the DAOS container is initialized as an

encapsulated namespace), a reserved object (with a predefined object ID) will be added to the container and will record superblock information about the namespace. The SB object is replicated and has the reserved OID 0.0.

The SB object contains an entry with a magic value to indicate it is a POSIX filesystem. The SB object will also contain an entry to the root directory of the filesystem, which will be another reserved object with a predefined oid (1.0) and will have the same representation as a directory (see next section). The oid of the root id will be inserted as an entry in the superblock object.

The SB will look like this:

D-key: "DFS\_SB\_DKEY"

A-key: "DFS\_SB\_AKEY"

single-value (uint64\_t): SB\_MAGIC (0xda05df50da05df50)

D-key: "/"

// rest of akey entries for root are same as in directory entry described below.

**19.5.2 DFS Directories**

A POSIX directory will map to a DAOS object with multiple dkeys, where each dkey will correspond to an entry in that directory (for another subdirectory, regular file, or symbolic link). The dkey value will be the entry name in that directory. The dkey will contain several akeys of type DAOS\_IOD\_SINGLE (single value), where each akey contains an attribute of that entry. The mapping table will look like this (includes two extended attributes: xattr1,xattr2):

Directory Object

D-key "entry1\_name"

A-key "mode" // mode\_t (permission bit mask + type of entry)

A-key "oid" // object id of entry (bogus if symlink)

A-key "syml" // symlink value (akey does not exist if not a symlink)

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

98

A-key "atime" // access time

A-key "mtime" // modify time

A-key "ctime" // change time

A-key "x:xattr1" // extended attribute name (if any)

A-key "x:xattr2" // extended attribute name (if any)

The extended attributes are all prefixed with "x:".

This summarizes the mapping of a directory testdir with a file, directory, and symlink:testdirdir$ ls

dir1

file1

syml1 -> dir1

Object testdir

D-key "dir1"

A-key "mode" , permission bits + S\_IFDIR

A-key "oid" , object id of dir1

...

D-key "file1"

A-key "mode" , permission bits + S\_IFREG

A-key "oid" , object id of file1

...

D-key "syml1"

A-key "mode" , permission bits + S\_IFLNK

A-key "oid" , empty

A-key "syml", dir1

...

For files, we will have an optimization in the entry by storing the first 4K of data in the entry itself under another akey "file\_data" for the file entry. In this case, if the file size is less than or equal to 4K, the object ID akey will be empty, and the file data will be in the akey with array type of file\_size records. Otherwise, the "oid" akey will contain a valid object ID for the file data.

Note that with this mapping, the inode information is stored with the entry that it

corresponds to in the parent directory object. Thus, hard links won’t be supported, since it won’t be possible to create a different entry (dkey) that points to the same set of akeys that the current ones are stored within. This limitation was agreed upon and makes the

representation simple as described above.

**19.5.3 Files**

As shown in the directory mapping above, the entry of a file will be inserted in its parent directory object with an object ID that corresponds to that file. The object ID for a regular file will be of a DAOS array object, which itself is a DAOS object with some properties being the element size and chunk size. In the POSIX file case, the cell size will always be 1 byte.The chunk size can be set at create time only, with the default being 1 MB. The array object itself is mapped onto a DAOS object with integer dkeys, where each dkey contains

chunk\_size elements. So for example, if we have a file with size 10 bytes, and chunk size is3 bytes, the array object will contain the following:

Object array

D-key 0

A-key NULL , array elements [0,1,2]

D-key 1

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

99

A-key NULL , array elements [3,4,5]

D-key 2

A-key NULL , array elements [6,7,8]

D-key 3

A-key NULL , array elements [9]

For more information about the array object layout, please refer to the README.md file for Array Addons.

Access to that object is done through the DAOS Array API. All read and write operations to the file will be translated to DAOS array read and write operations. The file size can be set(truncate) or retrieved by the DAOS array set/get\_size functions. Increasing the file size;however in this case does not guarantee that space is allocated. Since DAOS logs I/Os across different epoch, space allocation cannot be supported by a naïve set size operation.

**19.5.4 Symbolic Links**

As mentioned in the directory section, symbolic links will not have an object for the symlink itself but will have a value in the entry itself of the parent directory containing the actual value of the symlink.

**19.5.5 Access Permissions**

All DFS objects (files, directories, and symlinks) inherit the access permissions of the DFS pool that they are created with. So when a user is trying to access an object in the DFS namespace, it's real/effective uid/gid are compared against those of the pool's uid and gid that are obtained when connecting to the pool. The check then is done with the stored object mode and depending on the type of access being requested (R, W, X) and the object mode, access permission is determined. In the source code, this is implemented in the function check\_access().

setuid(), setgid() programs, supplementary groups, ACLs are not supported at the moment.

**19.5.6 DFUSE**

A simple high-level fuse plugin (dfuse) is implemented to test the DFS API and functionality with existing POSIX tests and benchmarks (IOR, mdtest, etc.). The DFS fuse exposes one mount point as a single DFS namespace with a single pool and container. To test dfuse, the following steps need to be done:

1. Launch DAOS server(s): orterun --mca mtl ^psm2,ofi --enable-recovery -np 1 --report-

uri ~/uri.txt daos\_server -c 8

2. Create a DAOS Pool with dmg tool: this will return a pool uuid "puuid" and service rank

list "svcl"

3. Create an empty directory for the fuse mount point. For example let's use /tmp/dfs\_test4. Mount dfuse with the following command: orterun -np 1 --ompi-server file:~/uri.txt

dfuse /tmp/dfs\_test -s -f -p puuid -l svcl -p specifies the pool uuid and -l specifies the service rank list (from dmg).

5. Other arguments to dfuse: -r: option to destroy the container associated with the

namespace when you umount. -d: prints debug messages at the fuse mount terminal6. Now /tmp/dfs\_test can be used as a POSIX file system (can run things like IOR/mdtest

on it)

7. when you are done, unmount the file system: fusermount -u /tmp/dfs\_test

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

Milestone No. 151 CORAL A21 NRE Subcontract No. 8F-30005

**20 Terminology**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Acronym** |  | **Expansion** |
| ABT | | | Argobots |
| BIO | | | Blob I/O |
| CART | | | Collective and RPC Transport |
| DAOS | | | Distributed Asynchronous Object Storage |
| dRPC | | | DAOS Remote Procedure Call |
| gRPC | | | gRPC Remote Procedure Calls |
| GURT | | | Gurt Useful Routines and Types |
| HLC | | | Hybrid Logical Clock |
| ISA-L | | | Intel Storage Acceleration Library |
| KV store | | | Key-Value store |
| MTBF | | | Mean Time Between Failures |
| OFI | | | Open Fabrics Interface |
| NVM | | | Non-Volatile Memory |
| NVMe | | | Non-Volatile Memory express |
| PM/PMEM | | | Persistent Memory |
| PMDK | | | Persistent Memory Development Kit |
| RAS | | | Reliability, Availability & Serviceability |
| RDMA/RMA | | | Remote (Direct) Memory Access |
| SCM | | | Storage-Class Memory |
| SWIM | | | Scalable Weakly-consistent Infection-style process group Membership |
| SPDK | | | Storage Performance Development Kit |
| SSD | | | Solid State Drive |
| ULT | | | User Level Thread |
| UUID | | | Universal Unique Identifier |
| RDG | | | Redundancy Group |
| VOS | | | Versioning Object Store |

Intel Corporation Proprietary Information

Copyright © 2019, Intel Corporation. All rights reserved – unpublished work.

100