Abstract: This SNIA document defines recommended behavior for hardware and software that supports Computational Storage.

Publication of this Working Draft for review and comment has been approved by the Computational Storage TWG. This draft represents a “best effort” attempt by the Computational Storage TWG to reach preliminary consensus, and it may be updated, replaced, or made obsolete at any time. This document should not be used as reference material or cited as other than a “work in progress.” Suggestions for revisions should be directed to http://www.snia.org/feedback/.

Working Draft

August 6th, 2020
USAGE
Copyright © 2020 SNIA. All rights reserved. All other trademarks or registered trademarks are the property of their respective owners.

The SNIA hereby grants permission for individuals to use this document for personal use only, and for corporations and other business entities to use this document for internal use only (including internal copying, distribution, and display) provided that:

1. Any text, diagram, chart, table or definition reproduced shall be reproduced in its entirety with no alteration, and,

2. Any document, printed or electronic, in which material from this document (or any portion hereof) is reproduced, shall acknowledge the SNIA copyright on that material, and shall credit the SNIA for granting permission for its reuse.

Other than as explicitly provided above, you may not make any commercial use of this document or any portion thereof, or distribute this document to third parties. All rights not explicitly granted are expressly reserved to SNIA.

Permission to use this document for purposes other than those enumerated above may be requested by e-mailing tcmd@snia.org. Please include the identity of the requesting individual and/or company and a brief description of the purpose, nature, and scope of the requested use.

All code fragments, scripts, data tables, and sample code in this SNIA document are made available under the following license:

BSD 3-Clause Software License

Copyright (c) 2020, The Storage Networking Industry Association.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of The Storage Networking Industry Association (SNIA) nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
DISCLAIMER

The information contained in this publication is subject to change without notice. The SNIA makes no warranty of any kind with regard to this specification, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The SNIA shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this specification.
## Revision History

<table>
<thead>
<tr>
<th>Major</th>
<th>Revision</th>
<th>Editor</th>
<th>Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>R1</td>
<td>Stephen Bates</td>
<td>02-Feb-2019</td>
<td>Initial conversion of the NVM Programming Model document.</td>
</tr>
<tr>
<td>0.1</td>
<td>R2</td>
<td>Stephen Bates</td>
<td>11-Mar-2019</td>
<td>Added initial draft of Scope section with some initial feedback</td>
</tr>
<tr>
<td>0.1</td>
<td>R3</td>
<td>Stephen Bates</td>
<td>02-April-2019</td>
<td>Added diagram to Scope section and introduced “Computational Function” as a term.</td>
</tr>
<tr>
<td>0.1</td>
<td>R4</td>
<td>Nick Adams</td>
<td>04-Apr-2019</td>
<td>Updates per Morning Session on 03-Apr-2019</td>
</tr>
<tr>
<td>0.1</td>
<td>R5</td>
<td>Scott Shadley</td>
<td>24-Apr-2019</td>
<td>Updates added per F2F meeting in Boston</td>
</tr>
<tr>
<td>0.1</td>
<td>R6</td>
<td>Stephen Bates and Bill Martin</td>
<td>21-Aug-2019</td>
<td>Removed resolved comments, added Theory of Operation, added illustrative diagrams, added placeholder for an illustrative example. Also performed a large amount of editorial clean-up. Added forward and removed taxonomy and abstract (approved by TWG on August 28th 2019).</td>
</tr>
<tr>
<td>0.1</td>
<td>R7</td>
<td>Stephen Bates and Bill Martin</td>
<td>XX Oct 2019</td>
<td>Update foreword section. Add references, abbreviations and definitions.</td>
</tr>
<tr>
<td>0.1</td>
<td>R8</td>
<td>David Slik</td>
<td>12 Nov 2019</td>
<td>Moved fields into tables, updated diagrams.</td>
</tr>
<tr>
<td>0.1</td>
<td>R9</td>
<td>Stephen Bates</td>
<td>14 Nov 2019</td>
<td>Minor editorial changes</td>
</tr>
<tr>
<td>0.1</td>
<td>R10</td>
<td>Bill Martin</td>
<td>20 Nov 2019</td>
<td>Incorporated approved dictionary terms, abbreviations, and reference modifications</td>
</tr>
<tr>
<td>0.1</td>
<td>R11</td>
<td>Bill Martin</td>
<td>11 Dec 2019</td>
<td>Incorporated NetInt proposal for Video Compression CSS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Incorporate ScaleFlux proposal for Database Filter CSS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Removed previously accepted changes</td>
</tr>
<tr>
<td>0.1</td>
<td>R12</td>
<td>Stephen Bates</td>
<td>15 Dec 2019</td>
<td>Incorporated DellEMC proposal for Hashing and CRC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Removed list of CSSes at start of Section 5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Remove reference to how to add new CSSes to Section 5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Update the list of contributors as per the list on the Wiki</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Resolved and removed comments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added illustrative examples from Eideticom and Arm/NGD Systems in their current form and started merging them correctly into this document.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Incorporated Seagate proposal for large dataset PCSS</td>
</tr>
<tr>
<td>0.1</td>
<td>R13</td>
<td>Stephen Bates</td>
<td>18 Dec 2019</td>
<td>Removed Annex B (Illustrative Examples)</td>
</tr>
<tr>
<td>0.3</td>
<td>R1</td>
<td>Stephen Bates</td>
<td>18 Dec 2019</td>
<td>Update author list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Slight fix to Philip’s PCSS example</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Add Raymond’s FCSS example</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Minor corrections to document.</td>
</tr>
<tr>
<td>0.4</td>
<td>R1</td>
<td>Bill Martin</td>
<td>8 April 2020</td>
<td>Added PCSS on a Large Multi-Device Dataset</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added PCSS – Linux Container Example</td>
</tr>
<tr>
<td>0.4</td>
<td>R2</td>
<td>Bill Martin</td>
<td>20 May 2020</td>
<td>Added FCSS – Data Deduplication</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added PCSS – OPEN CL example</td>
</tr>
<tr>
<td>0.4</td>
<td>R3</td>
<td>Bill Martin</td>
<td>29 July 2020</td>
<td>Updated example in B.4 Data Deduplication Example</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added Data Compression Illustrative Example</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added Data Filtration Illustrative Example</td>
</tr>
<tr>
<td>0.4</td>
<td>R4</td>
<td>Bill Martin</td>
<td>5 August 2020</td>
<td>Added Scatter Gather illustrative example</td>
</tr>
<tr>
<td>0.5</td>
<td>R1</td>
<td>Bill Martin</td>
<td>10 Aug 2020</td>
<td>Minor editorial cleanup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Removed editor’s notes in preparation for public review release</td>
</tr>
</tbody>
</table>
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOREWORD</td>
<td>10</td>
</tr>
<tr>
<td>1 SCOPE</td>
<td>11</td>
</tr>
<tr>
<td>2 REFERENCES</td>
<td>13</td>
</tr>
<tr>
<td>3 DEFINITIONS, ABBREVIATIONS, AND CONVENTIONS</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Definitions</td>
<td>14</td>
</tr>
<tr>
<td>3.2 Keywords</td>
<td>15</td>
</tr>
<tr>
<td>3.3 Abbreviations</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Conventions</td>
<td>16</td>
</tr>
<tr>
<td>4 THEORY OF OPERATION</td>
<td>17</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>17</td>
</tr>
<tr>
<td>4.2 CSS Discovery</td>
<td>17</td>
</tr>
<tr>
<td>4.3 CSS Configuration</td>
<td>19</td>
</tr>
<tr>
<td>4.4 CSS Usage</td>
<td>20</td>
</tr>
<tr>
<td>5 COMPUTATION STORAGE SERVICES</td>
<td>23</td>
</tr>
<tr>
<td>5.1 Documented Fixed Computational Storage Services</td>
<td>23</td>
</tr>
<tr>
<td>5.2 Documented Programmable Computational Storage Services</td>
<td>25</td>
</tr>
<tr>
<td>ANNEX A. (INFORMATIVE) EXAMPLE CONFIGURATIONS</td>
<td>27</td>
</tr>
<tr>
<td>ANNEX B. (INFORMATIVE) ILLUSTRATIVE EXAMPLES</td>
<td>28</td>
</tr>
<tr>
<td>B.1 PCSS on a Large Multi-Device Dataset Using Ceph</td>
<td>28</td>
</tr>
<tr>
<td>B.1.1 Introduction</td>
<td>28</td>
</tr>
<tr>
<td>B.1.2 Theory of Operation</td>
<td>30</td>
</tr>
<tr>
<td>B.1.3 Discovery</td>
<td>30</td>
</tr>
<tr>
<td>B.1.4 Configuration</td>
<td>31</td>
</tr>
<tr>
<td>B.1.5 Usage</td>
<td>31</td>
</tr>
<tr>
<td>B.1.6 Example Application Deployment</td>
<td>32</td>
</tr>
<tr>
<td>B.2 Linux Container Programmable CSS</td>
<td>34</td>
</tr>
</tbody>
</table>
B.2.1 THEORY OF OPERATION ................................................................. 34
B.2.2 DISCOVERY .................................................................................. 34
B.2.3 CONFIGURATION .......................................................................... 35
B.2.4 USAGE .......................................................................................... 36
B.2.5 EXAMPLE OF APPLICATION DEPLOYMENT .................................. 38
B.3 DATA DEDUPLICATION FIXED CSS ................................................ 40
B.3.1 OVERVIEW .................................................................................... 40
B.3.2 THEORY OF OPERATION ............................................................... 42
B.3.3 DISCOVERY .................................................................................... 43
B.3.4 CONFIGURATION ........................................................................... 43
B.3.5 OPERATION ..................................................................................... 45
B.3.6 MONITORING ................................................................................ 45
B.3.7 FCSS DATA DEDUPLICATION EXAMPLE ....................................... 45
B.4 A PROGRAMMABLE COMPUTATIONAL STORAGE SERVICE ON A NVM EXPRESS AND OPENCL BASED COMPUTATIONAL STORAGE DRIVE ILLUSTRATIVE EXAMPLE ................................................. 51
B.4.1 PCSS EXAMPLE ............................................................................. 51
B.4.2 COMPUTATIONAL STORAGE DRIVE ............................................. 52
B.4.3 THEORY OF OPERATION ............................................................... 53
B.4.4 DISCOVERY .................................................................................... 53
B.4.4.1 NVMe FUNCTION DISCOVERY .................................................. 53
B.4.4.2 OPENCL CSP FUNCTION DISCOVERY ....................................... 54
B.4.5 CONFIGURATION – EXPLICIT MODE ........................................... 54
B.4.6 USAGE – STORAGE DIRECT .......................................................... 54
B.4.7 USAGE – EXPLICIT MODE COMPUTATIONAL STORAGE ............. 54
B.4.8 CONFIGURATION – TRANSPARENT MODE .................................... 55
B.4.9 USAGE – COMPUTATIONAL STORAGE TRANSPARENT MODE ........ 55
B.5 FCSS DATA COMPRESSION EXAMPLE .......................................................... 56
  B.5.1 OVERVIEW ......................................................................................... 56
  B.5.2 THEORY OF OPERATION ...................................................................... 56
  B.5.3 DISCOVERY ....................................................................................... 56
  B.5.4 CONFIGURATION ............................................................................. 57
  B.5.5 MONITORING .................................................................................... 57
B.6 FCSS DATA FILTER EXAMPLE ................................................................. 58
  B.6.1 OVERVIEW ......................................................................................... 58
  B.6.2 THEORY OF OPERATION ...................................................................... 59
  B.6.3 DISCOVERY ....................................................................................... 59
  B.6.4 CONFIGURATION ............................................................................. 59
  B.6.5 OPERATION ....................................................................................... 59
  B.6.6 MONITORING .................................................................................... 60
B.7 SCATTER-GATHER FCSS ........................................................................ 61
  B.7.1 OVERVIEW ......................................................................................... 61
  B.7.1.1 REPRESENTATION OF A DATA MOVEMENT REQUEST (DMR) ......... 61
  B.7.1.2 ENDPOINTS .................................................................................. 62
  B.7.1.3 FLOWS .......................................................................................... 62
  B.7.1.4 EXAMPLE DMRs .......................................................................... 63
  B.7.2 SCATTER-GATHER CSS THEORY OF OPERATION ............................... 65
  B.7.2.1 DISCOVERY .................................................................................... 65
  B.7.2.2 CONFIGURATION .......................................................................... 65
  B.7.2.3 OPERATION .................................................................................... 66
  B.7.2.4 MONITORING ................................................................................ 66
### Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>An Architectural view of Computational Storage</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Discovery Interactions</td>
<td>17</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Configuration Interactions</td>
<td>19</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Direct Usage Interactions</td>
<td>21</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Transparent Usage Interactions</td>
<td>21</td>
</tr>
<tr>
<td>Figure B.1.1</td>
<td>Ceph, a scale-out storage system</td>
<td>29</td>
</tr>
<tr>
<td>Figure B.1.2</td>
<td>Computational Storage architectural diagram</td>
<td>29</td>
</tr>
<tr>
<td>Figure B.1.3</td>
<td>Ceph CSA</td>
<td>30</td>
</tr>
<tr>
<td>Figure B.1.4</td>
<td>Ceph CSDs</td>
<td>30</td>
</tr>
<tr>
<td>Figure B.1.5</td>
<td>Skyhook Example 1</td>
<td>32</td>
</tr>
<tr>
<td>Figure B.1.6</td>
<td>Skyhook Example 2</td>
<td>33</td>
</tr>
<tr>
<td>Figure B.2.1</td>
<td>The Computational Storage architectural diagram from Section X.Y</td>
<td>34</td>
</tr>
<tr>
<td>Figure B.2.2</td>
<td>CSx discovery process</td>
<td>35</td>
</tr>
<tr>
<td>Figure B.2.3</td>
<td>CSx Configuration process?</td>
<td>36</td>
</tr>
<tr>
<td>Figure B.2.4</td>
<td>CSx usage</td>
<td>37</td>
</tr>
<tr>
<td>Figure B.3.1</td>
<td>Simplified Data Deduplication Process</td>
<td>40</td>
</tr>
<tr>
<td>Figure B.3.2</td>
<td>Post Process Data Deduplication</td>
<td>41</td>
</tr>
<tr>
<td>Figure B.3.3</td>
<td>Inline Data Deduplication</td>
<td>42</td>
</tr>
<tr>
<td>Figure B.3.4</td>
<td>Configuration Parameters</td>
<td>44</td>
</tr>
<tr>
<td>Figure B.3.5</td>
<td>Volume Creation</td>
<td>46</td>
</tr>
<tr>
<td>Figure B.3.6</td>
<td>Enable Deduplication</td>
<td>46</td>
</tr>
<tr>
<td>Figure B.3.7</td>
<td>Retrieve Status</td>
<td>47</td>
</tr>
<tr>
<td>Figure B.3.8</td>
<td>Disable Deduplication Scheduling</td>
<td>47</td>
</tr>
<tr>
<td>Figure B.3.9</td>
<td>Copy files</td>
<td>48</td>
</tr>
<tr>
<td>Figure B.3.10</td>
<td>Verify space utilized</td>
<td>48</td>
</tr>
<tr>
<td>Figure B.3.11</td>
<td>Schedule deduplication</td>
<td>49</td>
</tr>
<tr>
<td>Figure B.3.12</td>
<td>Monitor Progress</td>
<td>49</td>
</tr>
<tr>
<td>Figure B.3.13</td>
<td>verify space savings</td>
<td>50</td>
</tr>
<tr>
<td>Figure B.4.1</td>
<td>The Computational Storage architectural diagram from Section X.Y adapted to be specific to the illustrative example discussed in this section</td>
<td>52</td>
</tr>
<tr>
<td>Figure B.6.1</td>
<td>An example to illustrate the function of data filter FCSS</td>
<td>58</td>
</tr>
<tr>
<td>Figure B.7.1</td>
<td>Top-level DMR structure</td>
<td>62</td>
</tr>
<tr>
<td>Figure B.7.2</td>
<td>Example endpoint definition structure for an NVMe endpoint</td>
<td>62</td>
</tr>
<tr>
<td>Figure B.7.3</td>
<td>Flow definition structure</td>
<td>63</td>
</tr>
<tr>
<td>Figure B.7.4</td>
<td>Scatter DMR</td>
<td>63</td>
</tr>
<tr>
<td>Figure B.7.5</td>
<td>Example Scatter DMR</td>
<td>64</td>
</tr>
<tr>
<td>Figure B.7.6</td>
<td>Pipeline DMR</td>
<td>64</td>
</tr>
</tbody>
</table>
Figure B.7. 5 Example Pipeline DMR

65
FOREWORD

The SNIA Computational Storage Technical Working Group was formed to establish architectures and software computation in its many forms to be more tightly coupled with storage, at both the system and drive level. An architecture model and a programing model are necessary to allow vendor-neutral, interoperable implementations of this industry architecture.

This SNIA specification outlines the architectural models that are defined to be Computational Storage, specific Computational Storage Services, and a programming model for those architectures and services. As this specification is developed, requirements in interface standards and specific APIs may be proposed as separate documents and developed in the appropriate organizations.

Acknowledgements

The SNIA Computational Storage Technical Working Group, which developed and reviewed this standard, would like to recognize the significant contributions made by the following members:

<table>
<thead>
<tr>
<th>Organization Represented</th>
<th>Name of Representative</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Jason Molgaard</td>
</tr>
<tr>
<td>DellEMC</td>
<td>Amnon Izhar</td>
</tr>
<tr>
<td>Eideticom</td>
<td>Stephen Bates</td>
</tr>
<tr>
<td>IBM</td>
<td>Raymond Swank</td>
</tr>
<tr>
<td>Intel</td>
<td>Nick Adams</td>
</tr>
<tr>
<td>NetApp</td>
<td>David Slik</td>
</tr>
<tr>
<td>NetlNT</td>
<td>John Plasterer</td>
</tr>
<tr>
<td>NGD Systems</td>
<td>Scott Shadley</td>
</tr>
<tr>
<td>Samsung</td>
<td>Bill Martin</td>
</tr>
<tr>
<td>ScaleFlux</td>
<td>JB Baker</td>
</tr>
<tr>
<td>ScaleFlux</td>
<td>Yang Liu</td>
</tr>
<tr>
<td>Seagate</td>
<td>Philip Kufeldt</td>
</tr>
</tbody>
</table>
1 Scope

This specification focuses on defining the capabilities and actions that are able to be implemented across the interface between Computational Storage Devices (CSxes) (e.g. Computational Storage Processors, Computational Storage Drives and Computational Storage Arrays) and either Host Agents or other CSxes.

The actions mentioned above are associated with several aspects of a CSx:

- **Management.** Actions that allow Host Agent(s), based on security policies, to perform:
  - **Discovery.** Mechanisms to identify and determine the capabilities and Computational Storage Services (CSSes).
  - **Configuration.** Programming parameters for initialization, operation, and/or resource allocation
  - **Monitoring.** Reporting mechanisms for events and status
- **Security.** Actions that allow Host Agent(s) and/or CSx to perform:
  - **Authentication.** Host agent to CSx and CSx to host agent
  - **Authorization.** Mechanisms for secure data access and permissions control
  - **Encryption.** Mechanisms to perform computation on encrypted data that was not encrypted by the CSx. Mechanisms that exchange information necessary for the CSx to encrypt/decrypt data.
  - **Auditing.** Mechanisms that allow for generating and retrieving of a secure log
- **Operation.** Mechanisms for the CSx to store and retrieve data. May allow a Host Agent or CSx to offload certain Computational Storage tasks to a CSx, including providing the target CSx with information about data locality both local to the CSx or resident on one or more non-local locations.

This specification makes no assumptions about the physical nature of the interface between the Host Agent and CSx(s). This specification and the actions associated with it will be
implemented across a range of different physical interfaces. This specification also makes no assumptions about the storage protocols used by Host Agents and CSx(s).

It is expected that the following storage protocols between the Host Agent and the CSx will be addressed:

- **Logical Block Address.** Data is grouped into fixed-size logical units and operations are atomic at that unit size. Data is indexed via a numerical index into the Logical Block Address.
- **Key-Value.** Data is not fixed-size and is indexed by a key.
- **Persistent Memory.** Byte addressable non-volatile memory.

Section 5.1 of this specification defines actions for Fixed Computational Storage Services. These interfaces are specified in such a way that they are able to be consumed by a Host Agent in a well-defined manner.

Section 5.2 of his specification defines actions for configuring Programmable Computational Storage Services. These interfaces are specified in such a way that they are able to be programmed by a Host Agent in a well-defined manner.

This specification defines actions for passing data through multiple Computational Storage Services that may or may not reside on a single CSx. Additionally, it defines actions for requesting multiple Computational Storage Services to perform a set of tasks.
2 References

The following referenced documents are indispensable for the application of this document.

For references available from ANSI, contact ANSI Customer Service Department at (212) 642-4900/4980 (phone), (212) 302-1286 (fax) or via the World Wide Web at http://www.ansi.org.

NVMe 1.4 NVM Express Revision 1.4,
Approved standard, available from http://nvmexpress.org
3 Definitions, abbreviations, and conventions
For the purposes of this document, the following definitions and abbreviations apply.

3.1 Definitions

3.1.1 Computational Storage
Architectures that provide Computational Storage Services coupled to storage, offloading host processing or reducing data movement.

note 1 to entry:
These architectures enable improvements in application performance and/or infrastructure efficiency through the integration of compute resources (outside of the traditional compute & memory architecture) either directly with storage or between the host and the storage. The goal of these architectures is to enable parallel computation and/or to alleviate constraints on existing compute, memory, storage, and I/O.

3.1.2 Computational Storage Array (CSA)
collection of Computational Storage Devices, control software, and optional storage devices.

3.1.3 Computational Storage Device (CSx)
Computational Storage Drive, Computational Storage Processor, or Computational Storage Array.

3.1.4 Computational Storage Drive (CSD)
storage element that provides Computational Storage Services and persistent data storage.

3.1.5 Computational Storage Processor (CSP)
component that provides Computational Storage Services for an associated storage system without providing persistent data storage.

3.1.6 Computational Storage Service (CSS)
data service or information service that performs computation on data where the service and the data are associated with a storage device. The Computational Storage Service may be a Fixed Computational Storage Service or a Programmable Computational Storage Service.

3.1.7 Filesystem
software component that imposes structure on the address space of one or more physical or virtual disks so that applications may deal more conveniently with abstract named data objects of variable size called files

3.1.8 Fixed Computational Storage Service (FCSS)
CSS that provides a given function that may be configured and used. (Service examples: compression, RAID, erasure coding, regular expression, encryption).
3.1.9 Key Value
Storage that stores and retrieves user data based on a key that is associated with that data

3.1.10 Object Store
storage that accesses data as objects and provides services for storing, searching and returning that data based on content of the storage device

3.1.11 Programmable Computational Storage Service (PCSS)
CSS that is able to be programmed to provide one or more CSSes. (Service examples: this service may host an operating system image, container, Berkeley packet filter, FPGA bitstream).

3.2 Keywords
In the remainder of the specification, the following keywords are used to indicate text related to compliance:

3.2.1 mandatory
a keyword indicating an item that is required to conform to the behavior defined in this standard

3.2.2 may
a keyword that indicates flexibility of choice with no implied preference; “may” is equivalent to “may or may not”

3.2.3 may not
keywords that indicate flexibility of choice with no implied preference; “may not” is equivalent to “may or may not”

3.2.4 need not
keywords indicating a feature that is not required to be implemented; “need not” is equivalent to “is not required to”

3.2.5 optional
a keyword that describes features that are not required to be implemented by this standard; however, if any optional feature defined in this standard is implemented, then it shall be implemented as defined in this standard

3.2.6 shall
a keyword indicating a mandatory requirement; designers are required to implement all such mandatory requirements to ensure interoperability with other products that conform to this standard

3.2.7 should
a keyword indicating flexibility of choice with a strongly preferred alternative
3.3 Abbreviations

CSA  Computational Storage Array
CSD  Computational Storage Drive
CSP  Computational Storage Processor
CSS  Computational Storage Service
CSx  Computational Storage devices
FCSS Fixed Computational Storage Service
NVM  Non-Volatile Memory
PCSS Programmable Computational Storage Service
PM   Persistent Memory
SSD  Solid State Disk

3.4 Conventions

Representation of modes in figures

Modes are represented by red, wavy lines in figures, as shown below:

The wavy lines have labels identifying the mode name (which in turn, identifies a clause of the specification).
4 Theory of Operation

4.1 Overview

This clause describes the theory of operations for Computational Storage Devices (CSxes) and Computational Storage Services (CSSes).

4.2 CSS Discovery

4.2.1 CSS Discovery Overview

To utilize Computational Storage Services, the CSSes must be discovered. This requires first performing a fabric-specific discovery process to identify CSxes, and second CSS discovery process to identify CSSes for each discovered CSx. The fabric specific CSx discovery process is specified in Clause 4.2.2.

Discovery is accomplished through the process illustrated in Figure 4.1.

Figure 4.1 - Discovery Interactions

The steps shown in Figure 4.1 to discover CSSes are:

(1) The host sends a CSS discovery request over the fabric to one or more CSxes.
(2) CSxes may repeat this discovery process for internally accessible CSxes, which may use the standardized CSx/CSS discovery process.
(3) Each CSx that accepts the discovery request returns a CSS discovery response to the requesting host.
### 4.2.2 CSS Discovery Request

A CSS discovery request contains the following information:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSx Identifier</td>
<td>String</td>
<td>Unique identifier of the CSx target</td>
<td>Yes</td>
</tr>
<tr>
<td>Host Identifier</td>
<td>String</td>
<td>Unique identifier of the Host</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS filter</td>
<td>Object</td>
<td>Restricts discovery to only CSSes that match the specified filter (e.g., all CSSes, specific types of CSSes, a specific CSS).</td>
<td>Yes</td>
</tr>
</tbody>
</table>

A discovery request may be sent to an already discovered CSS to discover the current state.

### 4.2.3 CSS Discovery Response

A CSx returns zero or more CSS Discovery Responses, depending on the CSS filter in the corresponding CSS Discovery Request.

A CSS discovery response contains the following information:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSx Identifier</td>
<td>String</td>
<td>Unique identifier of the CSx target</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Identifier</td>
<td>String</td>
<td>Unique identifier of the CSS</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Vendor</td>
<td>String</td>
<td>Unique identifier of the CSS Vendor</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Type</td>
<td>String</td>
<td>Indicates the type of the CSS <em>(Indicated Fixed/Programmable here?)</em></td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Subtype</td>
<td>String</td>
<td>Indicates a subtype of the CSS</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS State</td>
<td>String</td>
<td>Indicates the state of the CSS: (initializing, ready, configuring, available, busy, error)</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Reservation</td>
<td>String</td>
<td>Indicates if the CSS is reserved by a host: (Empty or Host ID)</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Active Configuration Descriptor</td>
<td>Object</td>
<td>A CSS-specific data structure that indicates the currently active configuration, including options and parameters</td>
<td>No</td>
</tr>
<tr>
<td>CSS Configuration Schema</td>
<td>Object</td>
<td>A self-describing data structure that indicates valid configurations, options and parameters for the CSS</td>
<td>No</td>
</tr>
<tr>
<td>CSS Error</td>
<td>Array of Strings</td>
<td>Indicates if there are any CSS errors</td>
<td>Yes</td>
</tr>
</tbody>
</table>
4.3 CSS Configuration

4.3.1 CSS Configuration Overview

Once a CSS is discovered, a Computational Storage Service driver may configure the discovered Computational Storage Service to prepare it for use. This is accomplished through the configuration process illustrated in Figure 4.2.

![Configuration Interactions Diagram](image)

The steps shown in figure 4.2 to configure a CSS are:

1. The host sends a CSS configuration request over the fabric to a target CSx.
2. The target CSx may repeat the configuration process with internally accessible CSxes.
3. For Programmable CSSes, the configuration process may result in the creation of one or more new Programmable and/or Fixed CSSes.
4. The target CSx returns a CSS configuration response to the requesting host.

4.3.2 CSS Configuration Request

A CSS configuration request contains the following information:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Identifier</td>
<td>String</td>
<td>Globally uniquely identifier for the host</td>
<td>Yes</td>
</tr>
<tr>
<td>Host Credentials</td>
<td>Object</td>
<td>Allows the host to authenticate with the CSS</td>
<td>Yes</td>
</tr>
<tr>
<td>CSx Identifier</td>
<td>String</td>
<td>Unique identifier of the CSx target</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Identifier</td>
<td>String</td>
<td>Indicates which CSS is being configured</td>
<td>Yes</td>
</tr>
</tbody>
</table>
CSS Configuration Object

Configuration information, as defined by the CSS configuration descriptor

CSS Reservation String

Indicates that the CSS shall be reserved for the host

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSx Identifier</td>
<td>String</td>
<td>Unique identifier of the CSx target</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Identifier</td>
<td>String</td>
<td>Unique identifier of the CSS</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS State</td>
<td>String</td>
<td>Indicates the state of the CSS: (initializing, ready, configuring, available, busy, error)</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Reservation</td>
<td>String</td>
<td>Indicates if the CSS is reserved by a host: (Empty or Host ID)</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Configuration</td>
<td>Object</td>
<td>Accepted configuration information, which may differ from what was requested by the host</td>
<td>Yes</td>
</tr>
<tr>
<td>CSS Error</td>
<td>Array of Strings</td>
<td>Indicates if there are any CSS errors</td>
<td>Yes</td>
</tr>
</tbody>
</table>

CSS configuration may take some time. The Host must either poll by sending discovery requests or have asynchronous notification to find out when the configuration is complete.

Newly created CSSes as a result of the configuration process of a PCSS are discovered using the standard CSS discovery process as described in 4.2.

4.4 CSS Usage

4.4.1 CSS Usage Overview

Once configured, Computational Storage Service drivers may directly use the Computational Storage Service, or indirectly use the Computational Storage Service by performing standard storage operations against an associated Storage Interface. This is accomplished through the usage processes illustrated in Figure 4.3 and Figure 4.4.
The steps shown in figure 4.3 to directly use a CSS are:

1. The host sends a CSS command to a target CSS.
2. The target CSS may send one or more commands to other internally accessible CSSes.
(3) The target CSS may send one or more commands to other storage or memory devices to retrieve and/or store data.
(4) The target CSS returns a CSS command response to the requesting host.

The steps shown in figure 4.4 to transparently use a CSS through a standard storage interface are:

(1) The host sends unmodified storage interface operations to a target Storage Interfaced that is associated with the target CSS.

4.4.2 CSS Command
A CSS command is specific to the type of CSS (e.g., for a compression CSS, a command may instruct the CSS to read from a given location in system memory, compress the data, and store the resulting data to a specified location in a storage device).

4.4.3 CSS Command Response
A CSS command response is specific to the corresponding command.
5 Computation Storage Services

5.1 Documented Fixed Computational Storage Services

5.1.1 Compression FCSS
A compression CSS reads data from a source location, compresses or decompresses the data, and writes the result to a destination location.

CSS configuration specifies the compression algorithm and associated parameters.

CSS command specifies the source address and length and the destination address and maximum lengths.

5.1.2 Database Filter FCSS
A database filter CSS reads data from source location(s), performs a database projection (column selection) and filter (row selection) on the data according to projection and filter conditions, and writes the result(s) to destination location(s).

CSS configuration specifies the database format, table schema, selection and filter conditions, and associated parameters.

CSS command specifies the source address and length, and the destination addresses and lengths.

5.1.3 Encryption FCSS
An encryption CSS reads data from a source location, encrypts or decrypts the data, and writes the result to a destination location.

CSS configuration specifies the encryption algorithm, keying information, and associated parameters.

CSS command specifies the source address and length and the destination address and length.

5.1.4 Erasure Coding FCSS
An erasure coding CSS reads data from source location(s), performs a EC encode or decode on the data, and writes the result(s) to destination location(s).

CSS configuration specifies the EC algorithm and associated parameters.

CSS command specifies the source address and length and the destination addresses and lengths.
5.1.5 RegEx FCSS
A regex CSS reads data from source location(s), performs a regular expression patterning matching or transformation on the data, and writes the result(s) to the destination location.

CSS configuration specifies the RegEx string(s) and associated parameters.

CSS command specifies the source address and length and the destination address and length.

5.1.6 Scatter-Gather FCSS
A Scatter-Gather CSS reads data from set of source location(s) and writes the data to a set of destination location(s).

CSS configuration does not have any parameters.

CSS command specifies the source addresses and lengths and the destination addresses and lengths.

5.1.7 Pipeline FCSS
A Pipeline CSS performs a series of operations on data according to a data flow specification, allowing different CSS commands to be combined together in a standardized way.

CSS configuration does not have any parameters.

CSS command specifies a collection of commands, their order and dependencies, and calculations defining the relationships of the addresses between commands.

5.1.8 Video Compression FCSS
A video compression CSS reads data from a source location, compresses or decompresses the video, and writes the result to a destination location. In order to accommodate multiple parallel compressions, the video compression CSS may support a single compression stream or multiple compression stream.

CSS configuration specifies the stream, compression algorithm and associated parameters.

CSS command specifies the stream, source address and length and the destination address and maximum lengths.

5.1.9 Hash/CRC FCSS
A hash/CRC CSS reads data from a source location, calculates a hash or CRC value based on the source data, and writes the result to a destination location.

CSS configuration specifies the hashing/CRC algorithm and associated parameters.

CSS command specifies the source address and length and the destination address.
As an optional feature the CSS can calculate the hash/CRC value based on the source data and compare the hash/CRC result to a pre-calculated value supplied by the initiator. The CSS will notify the initiator whether the calculated value matches the supplied value.

**5.1.10 Data Deduplication FCSS**

A data deduplication CSS reads data from source location(s), performs deduplication or duplication on the data, and writes the result(s) to the destination location(s).

CSS configuration specifies the data deduplication algorithm and associated parameters.

CSS command specifies the source address and length and the destination address and maximum lengths.

**5.2 Documented Programmable Computational Storage Services**

**5.2.1 Operating System Image Loader PCSS**

An Operating System Image CSS allows an operating system image to be loaded and executed. The operating system may implement one or more additional CSSes.

CSS configuration specifies the location where the image is able to be obtained, and integrity/security verification information.

CSS command specifies pause/resume/start/stop/unload operations.

**5.2.2 Container Image Loader PCSS**

A Container Image CSS allows a container image to be loaded and executed. The container may implement one or more additional CSSes.

CSS configuration specifies the location where the container is able to be obtained, and integrity/security verification information.

CSS command specifies pause/resume/start/stop/unload operations.

**5.2.3 BPF Loader PCSS**

A Berkeley Packet Filter (BPF) CSS allows a BPF program to be loaded and executed. The BPF program may implement one or more additional CSSes.

CSS configuration includes the program to be run, and integrity/security verification information.

CSS command specifies pause/resume/start/stop/unload operations.

**5.2.4 FPGA Bitstream Loader PCSS**

A FPGA Bitstream CSS allows an FPGA bitstream to be programmed into an FPGA and executed. The FPGA program may implement one or more additional CSSes.
CSS configuration specifies the location where the FPGA bitstream is able to be obtained, and integrity/security verification information.

CSS command specifies the unload operation.

5.2.5 Large Data Set PCSS

A large data set wherein the data is sharded as objects across a plurality of computational storage devices (CSxes) and these objects are further tagged as belonging to a named object class. The object class being defined as a set of methods that act on those named objects.

CSS configuration includes the object class definition, the location where the object class is able to be obtained, integrity/security verification information and the set of CSxes where it is to be distributed.

CSS command specifies the objects to be acted upon, the object class, the object class method to enact and pause/resume/start/stop/unload operations.
Annex A. (Informative) Example Configurations
Annex B. (Informative) Illustrative Examples

B.1 PCSS on a Large Multi-Device Dataset using Ceph

B.1.1 Introduction

A large multi-device dataset is a dataset that:

1. may not fit into a single storage device;
2. may be large enough to require hundreds or thousands of devices; and
3. may require scalable performance by utilizing many storage devices.

Due to the size of these datasets, a single server may be insufficient to house all of the necessary storage devices. Consequently, these datasets may also span servers. This illustrative example uses TCP/IP as it provides scaling for a large number of devices.

A large dataset is able to be sharded into chunks and stored across a set of storage devices. To act on that data in the computational storage sense it is necessary to map the data shards to the devices where they are stored and then deliver a program/workload to a CSS on each CSx. This may be done simultaneously on thousands of drives.

There are many systems that enable the scaling of storage to thousands of devices. One such system used for this example is Ceph. Ceph allows many applications to jointly share shards of data called objects across potentially thousands of devices. Although intermediary servers called object storage daemons (OSDs) use local storage interconnects, the primary application interconnect is TCP/IP. Applications locate and interact with an object by a unique key that translates to a unique IP and port address. Applications do not dictate this address but rather let Ceph manage the location of the object, abstracting the clients from the actual location of the object.

Below is a diagram of Ceph showing client applications running in containers (App CT) using a variety of APIs (File, Block, S3) that are all implemented using the underlying Ceph RADOS API. This API permits the storing and retrieving of arbitrary sized objects as well as executing methods against objects.
The Ceph OSD servers in the diagram are responsive to the application’s object requests. Although a single OSD satisfies a single object request, a dataset may be sharded into many objects and those objects will be stored across all of the available OSDs. Since the application is abstracted away from an object’s location, the application should also be abstracted away from the CSS location. Furthermore, operating on a large dataset means engaging with many OSDs so abstracting the workload above the device level is also appropriate (i.e., an application should be able to have a distributed dataset while still operating on it as a single dataset).

Looking at the Computational Storage Architecture (see , Ceph could be viewed as the Storage Controller and the OSDs could be viewed as Computational Storage Processors.

Current Ceph deployment architectures have many OSDs running in the same server. In these configurations you can view a Ceph OSD server as a Computational Storage Array (CSA).
However, nothing in the software architecture prevents running a Ceph OSD as a single device server. In this case, the OSD server would be viewed as a Computational Storage Drive (CSD).

Figure B.1.4 – Ceph CSDs

B.1.2 Theory of Operation

In this section, we walk through the Computational Storage Theory of Operation (See Section 4) and apply it in a specific manner to this illustrative example. The three main phases of operation are:

a) Discovery (see section 4.2)

b) Configuration (see section 4.3)

c) Usage (see section 4.4)

B.1.3 Discovery

Most of the device discovery in this example is handled by Ceph; however, discovery of the Ceph system and the available PCSS needs to be done. Built into Ceph is the ability to run code in the OSDs against an object by means of an object class definition. An object class definition contains methods that can be executed within an OSD against an object. There are built-in object classes for interpreted languages (e.g. LUA) allowing for non-compiled PCSSes. Standard compiled shared objects are able to be created to implement custom object classes (i.e., custom compiled PCSSes), so discovery of these object classes will be necessary.
B.1.4 Configuration

A Ceph system shall be preinstalled and configured. The workloads are deployed by delivering the custom shared object classes or LUA scripts to each of the OSD servers via standard network copying mechanisms (e.g., Secure Copy).

Note: At the time of this writing, the list of available object classes are defined on the OSD. Although new object classes and LUA scripts are able to be added, they are required to be manually copied to each OSD. However, this manual step may be automated via the creation of new Ceph APIs permitting the remote loading of new object classes and scripts.

B.1.5 Usage

The application calls the Ceph RADOS rados_exec() function\(^1\), identifying the object class, the method of that class, naming the object, and passing any inbound parameters for the call. Upon return any outbound parameters are returned including the calls status.

\(^1\) There are several variants of the rados_exec() call, including versions that are associated with a read and write. There are also async versions of each call.
B.1.6 Example Application Deployment

Skyhook is a middleware application that provides a Ceph backend to PostgreSQL. It enables PostgreSQL table stores by sharding tables on row boundaries, placing N rows of a table into a single object. If N = 1000 and an example table had a total of 100,000 rows, Skyhook would create 100 objects that would be distributed to potentially 100 OSDs/Devices in Ceph.

Figure Error! No text of specified style in document. 5 – Skyhook Example 1

Skyhook has defined its own object class that is preconfigured on each Ceph OSD node. This object class implements methods that evaluate SQL queries. When PostgreSQL receives a user submitted SQL query, Skyhook submits that query to each of the 100 objects of the table in parallel by calling the RADOS exec() function. Passing the query to the SQL evaluation method, each object independently evaluates the query for its rows, returning the results back to skyhook.
Not only does this allow the DB to scale the performance and size independent of device capacities, but it also implements parallel execution.
B.2 Linux Container Programmable CSS

This illustrative example is of a Computational Storage Drive (CSD) based on the NVMe™ over PCIe® specification, that consists of one typical LBA based NVM storage device, multiple programmable applications processors co-located on the same controller capable of running Linux OS capable of running containerized application for searching data.

Specific assumptions for this illustrative example include:

a) A server running a modern Linux kernel and user-space distribution. If CPU specific items are discussed, we will assume an Intel Xeon processor;
b) A single-ported single host system (i.e., no consideration for multi-port NVMe devices;
c) No use of virtualization;
d) No Peer-to-Peer capabilities or namespaces are used; and
e) Leverages existing PCIe and NVMe methods around security.

Figure B.2.1 – The Computational Storage architectural diagram from Section X.Y

B.2.1 Theory of Operation

In this section, we will walk through the Theory of Operation (see section 4) and apply it in a specific manner to this illustrative example. The three main phases of operation are:

a) Discovery;
b) Configuration; and
c) Usage.

In this illustrative example, it is assumed that some updates to the existing NVMe Linux driver are made. The driver is required to be capable of recognizing and configuring a Computational Storage drive.

B.2.2 Discovery

NVMe/PCIe already has a very robust controller discovery and creation process. A PCIe device (i.e. an NVMe™ controller) is able to be discovered by the host at power-up time via PCIe bus
enumeration or at run-time via a hot-plug event and bus rescan. A modern Linux operating system is able to detect PCIe devices via both the methods mentioned above, allowing discovery of a new NVMe controller to be done at any time in a running system.

Once an NVMe controller has been detected, the NVMe Computational Storage driver can be used to discover the capabilities of the controller via the NVMe Identify Admin command. The procedure is:

1. PCIe enumeration discovers the NVMe controller of the CSD;
2. The Computational Storage NVMe Linux driver binds to this PCIe device; and
3. The driver discovers the capabilities of the Computational Storage Drive:
   a. Programmable Computational Storage Service; and
   b. Linux-capable applications processor(s) available in the drive;

![Figure B.2.2 – CSx discovery process](image_url)

**B.2.3 Configuration**

Linux and Docker are be pre-installed on the CSD in a previous instantiation of the drive or by a vendor at the factory. An Admin command causes the application processor(s) to boot into a Linux Environment. The software that supports Docker is loaded automatically as part of Linux start-up process.

Once Linux and Docker have booted, the Computational Storage NVMe driver is notified so that the specific compute workload can be downloaded. The workload is deployed to the device using an embedded packet process (e.g., TCP/IP, NVMe custom commands, or using the ‘copy' command).
B.2.4 Usage

The host downloads the compute function to the configured programmable CSD. The steps are as follows:

1. A deployable container is created on or copied to the host application and CPU;
2. The container is copied to the applications processor(s) on the CSD over the NVMe interface. This process may utilize a tunnelling effort, non-transparent, from the host to drive over the NVMe protocol link. In some cases, an external interface may be required, but is not shown in this example;
3. The container is launched by the internal CSP cores within the CSD, which begins the execution of that application process on data stored locally on the device.
Figure B.2.4 – CSx usage
B.2.5 Example of Application Deployment

The open source application called Openalpr (Open source Automatic License Plate Recognition) may be deployed using the Docker market place.

OpenALPR is an open source Automatic License Plate Recognition library written in C++ with bindings in C#, Java, Node.js, Go, and Python. The library analyzes images and video streams to identify license plates. The output is the text representation of any license plate characters. Openalpr provides a set of shared libraries but also makes use of a few other shared open source libraries.

In the Docker market place, there is a Docker image based on the “Vendor OS Choice” that contains the Openalpr shared libraries, its command-line utility application, and all the required shared libraries (e.g., OpenCV, python, and java)

In this specific example, the user would:

1) ssh from the host to the device using the default IP/username/password provided by the device vendor. (e.g., user@server:~# ssh csd@ipaddress)

2) Build a Docker image (i.e., user@server:~# docker build -t openalpr https://github.com/openalpr/openalpr.git)

3) Download test image (i.e., user@server:~# wget http://plates.openalpr.com/h786poj.jpg)

4) Run alpr on image (i.e., user@server:~# docker run -it --rm -v $(pwd):/data:ro openalpr -c eu h786poj.jpg)
The output of this example is:

```
user@server:~# /openalpr$ alpr ./h786poj.jpg

plate0: top 10 results -- Processing Time = 58.1879ms.
- PE3R2X  confidence: 88.9371
- PE32X   confidence: 78.1385
- PE3R2   confidence: 77.5444
- PE3R2Y  confidence: 76.1448
```
B.3 Data Deduplication Fixed CSS

B.3.1 Overview

Data deduplication is a technique used to reduce the amount of data stored on a storage device. If compression results in the removal of repeating bytes or streams within a chunk or segment of storage, then data deduplication results in the removal of matching chunks or segments of storage.

**Simplified Data Deduplication Process**

Data Object / Stream

**Chunking:** Data is split into chunks.

**Processing:** Chunks are identified and compared.

**Consolidation:** Duplicate chunks are re-referenced so multiple pointers reference a single unique chunk. Redundant chunks are then released.

![Figure B.3.1 – Simplified Data Deduplication Process](image)

Figure B.3.1 describes a simplified data deduplication process where a given data object or stream is given to the deduplication process. The data object is split into chunks where the chunks can then be identified and compared. A location repository of pointers referencing the unique chunks is created and any duplicate chunks are released from the data object so that the resulting storage is reduced.

There are two generic ways of performing data deduplication on a device. The first way is post process data deduplication where the data that is being deduplicated already resides on the device and is processed at a scheduled time and duration. The second way is inline data deduplication where the data that is being deduplicated is immediately processed for deduplication so that only unique data is stored on the device.
Figure B.3. 2 Post Process Data Deduplication

Figure B.3. 2 describes post process data deduplication. This process does not interfere with the initial ingestion of the data object or stream and allows the system to schedule a time and duration for the deduplication process. Once the data object or stream is stored, the data is chunked where the chunk metadata describes the size, references and location of the data. A repository is also created to map the location of each chunk with respect to the way the data was initially stored. The chunks are compared and if chunks are identical, the chunk metadata of the first instance of the chunk is updated with a reference count. The repository pointer for the identical chunk is also updated to the first instance of the chunk. Once the data object or stream has had all the chunks compared, the identical chunks are freed so that only a single instance of the chunk is stored on the device. Finally, the chunks are consolidated within the device.
Inline Data Deduplication

1. Data object or stream is received by the device.
2. A chunk repository is created and the initial chunk is stored with metadata and a reference to the chunk.
3. Subsequent chunks are compared to existing chunks, and if not identical, the chunk is stored with metadata and a reference to the chunk.
4. When an identical chunk is identified, the chunk metadata is updated with a reference to identify the location of the duplicate chunk.
5. The process continues until the data object or stream is fully read.

Figure B.3. 3 Inline Data Deduplication

Figure B.3.3 describes inline data deduplication. This process interferes with the ingestion of the initial data object or stream being saved to the device. As the data object or stream is being written to the device, chunks are compared and written only once if identical. This allows for less storage space to be used on the initial ingestion of the data object or stream. The first time the data is stored into the device, a chunk repository is created to describe the location of the chunks within the data object or stream. The initial chunk is stored with chunk metadata that describes size, location and references to the chunk. Subsequent chunks are then compared with existing chunks, and if unique, it is stored as a unique chunk. If it’s identical, the repository is updated to reference the single instance of the chunk and the number of references in the chunk metadata is updated. The process continues until the data object or stream is complete.

These descriptions of data deduplication are all simplified to explain generic ways to perform data deduplication. The CSx that will implement data deduplication will likely have more intricate proprietary ways of performing the data deduplication.

B.3.2 Theory of Operation

In order to successfully implement data deduplication as an FCSS, a CSx must be able to communicate the ability to perform data deduplication as an FCSS. Once the ability is determined, the FCSS then needs to be successfully configured. The data object or data stream can then be processed by the CSx using the data deduplication FCSS while allowing monitoring of the progress of that operation.
This illustrative example will attempt to provide the initial framework of the following steps needed to successfully allow for a CSx to perform data deduplication as an FCSS.

B.3.3 Discovery

In order to determine if a CSx supports the Data deduplication FCSS, the CSx needs to first be discovered as a CSx. If the CSx is capable of a Data deduplication FCSS, then the CSx needs to also indicate if the Data deduplication FCSS allows configuration. If the CSx allows for certain configuration parameters to the Data deduplication FCSS, then the configurable parameters need to be shared. The following is a possible list of configurable parameters:

1. Supported Chunk Sizes – The acceptable size values to chunk the data object into comparable chunks. Typically, these are large, but they could also be variable.
2. Scheduling – The schedule and duration of post process data deduplication.
3. Failover – The action to take if data deduplication is interrupted (e.g., discarding, resuming from last good write, restarting).
4. Monitoring – The type of data to collect during data deduplication (e.g., the current data deduplication space savings, the I/O rate of the data deduplication operation, the size of the data processed, the size of the data remaining to be processed, the percentage complete).
5. Inline or Post Process Deduplication
6. Type of method to perform chunk comparison – Three common types are hashing, binary comparison and delta differencing.
7. Hashing Algorithm – Type of hashing algorithms allowed to identify unique chunks.
8. Data deduplication Analysis – Methods to determine the likely savings gained by performing data deduplication on a data object or stream.
9. Operational Interruption – Whether or not the data deduplication operation is able to be interrupted for purpose of either suspending, abandoning or resuming the request.

Note that discovery may also return the default values of the configurable parameters as well as capabilities like operational interruption that the Data deduplication FCSS supports.

B.3.4 Configuration

Once a CSx that supports the Data deduplication FCSS is discovered, the Data deduplication FCSS can be configured if allowed. It’s possible that the Data deduplication FCSS will have default values that the user may not want to override, and configuration is not necessary. If the user wants to configure the Data deduplication FCSS, then the parameters returned by discovery need to be set and sent to the Data deduplication FCSS before the data deduplication operation is performed.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeduplicationSupport</td>
<td>Mandatory</td>
<td>Specifies whether Deduplication is to be configured a Fixed Computational Storage</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DeduplicationType</td>
<td>Optional</td>
<td>The type of deduplication to perform. This can be either inline or post process deduplication</td>
</tr>
<tr>
<td>DeduplicationSchedule</td>
<td>Conditional</td>
<td>Valid when post process deduplication is specified. This would be the time, frequency and duration when the deduplication would be performed.</td>
</tr>
<tr>
<td>DeduplicationChunk</td>
<td>Optional</td>
<td>If set, the size of the chunks to perform deduplication comparisons.</td>
</tr>
<tr>
<td>DeduplicationFailover</td>
<td>Optional</td>
<td>If set, the action to take when data deduplication is interrupted. Possible actions are discarding, resume from last good write, restart, etc.</td>
</tr>
<tr>
<td>DeduplicationMonitoring</td>
<td>Optional</td>
<td>If set, type of data to collect during data deduplication. Possible types of data to collect are current data deduplication space savings, the I/O rate of the data deduplication operation, the size of the data processed, the size of the data remaining to be processed, the percentage complete, etc.</td>
</tr>
<tr>
<td>DeduplicationComparison</td>
<td>Optional</td>
<td>If set, type of method to perform chunk comparison. Three common types are hashing, binary comparison and delta differencing.</td>
</tr>
<tr>
<td>DeduplicationHash</td>
<td>Optional</td>
<td>If set, type of hashing algorithm to identify unique chunks.</td>
</tr>
<tr>
<td>DeduplicationSavings</td>
<td>Optional</td>
<td>If set, perform an analysis on data being received to determine the amount space saved.</td>
</tr>
<tr>
<td>DeduplicationOperation</td>
<td>Optional</td>
<td>If set, allow for the interruption of the data deduplication service by request with option to either suspend, abandoned or resume the data deduplication operation.</td>
</tr>
</tbody>
</table>

Figure B.3. 4 Configuration Parameters
Figure B.3.4 lists the type of parameters that could be configured for the Data deduplication FCSS prior to sending a data object or stream to the CSx.

B.3.5 Operation

The data deduplication FCSS will begin when the data object or stream is sent. The data object or stream may have already been processed by another FCSS or be processed for input into another FCSS. For example, data deduplication is inefficient on encrypted data, so the Encryption FCSS should first decrypt the data before sending it to the data deduplication FCSS. Additionally, once the data has been deduplicated, the data is able to then be sent to the compression FCSS for additional storage savings.

The operation of the data deduplication FCSS may be interrupted if allowed. Otherwise, the user will need to wait for the operation to complete or fail to determine the next course of action.

B.3.6 Monitoring

As the data object is being processed by the data deduplication FCSS, the user can get status and statistics on the process. The following are possible status and statistics to monitor:

1. Current data deduplication space savings
2. The I/O rate of the data deduplication operation
3. Current amount of data processed by the data deduplication operation
4. Current amount of data remaining to be processed by the data deduplication operation
5. The percentage of completion of the data deduplication operation
6. Success or failure of the operation
7. Existing state of the operation such as paused, interrupted, resumed, etc.

Based on the status and statistics, the user can then determine if the operation needs to be paused, abandoned or resumed.

B.3.7 FCSS Data Deduplication Example
1. It is necessary to create a volume (see Figure B.3.5) to perform data deduplication.

2. Configure data duplication to be enabled on the created volume by sending a configuration parameter to the data deduplication FCSS. This operation is illustrated in Figure B.3.6.

3. Verify that data deduplication is enabled and ready for the created volume by retrieving status. The status indicates that the configurable parameters have been set. This operation is illustrated by Figure B.3.7.
4. Turn off the schedule of the post process data deduplication on the volume by sending the data deduplication FCSS a request that data deduplication not be scheduled to occur on the volume. This is shown in figure B.3.8.

5. Mount the volume to a server and copy files from existing server directories into the new directory as shown in Error! Reference source not found.. This writes the files to the newly created volume and since there is no deduplication scheduled, the data is not
6. Examine the volume for the storage consumed and space using existing server tools as shown in Figure B.3. 10. No space savings have been made since deduplication has yet to occur.

7. Send a request to the data deduplication FCSS, to schedule the data deduplication immediately on the volume as shown in Figure B.3. 11. The data deduplication then
occurs on the volume.

Figure B.3. 11 Schedule deduplication

8. Monitor the progress of deduplication by sending the data deduplication FCSS a status request. The data deduplication FCSS returns the amount of space that has been deduplicated and the percentage complete. This is shown in Figure B.3. 12 Monitor Progress.

Figure B.3. 12 Monitor Progress

9. When the deduplication is complete, check the space savings by sending a status request to the data deduplication FCSS as shown in Figure B.3. 13 verify space savings. The space savings depends on the amount of data that was deduplicated on
the volume.

Figure B.3. 13 verify space savings
B.4 A Programmable Computational Storage Service on a NVM Express and OpenCL based Computational Storage Drive Illustrative Example

B.4.1 PCSS Example

A PCIe OpenCL-based Programmable Computational Storage Drive (CSD) that consists of an NVMe Controller, an OpenCL accelerator with a PCIe Peer-to-Peer BAR, presenting as a programmable CSS.

Some other specific assumptions we will make about this illustrative example include:
- An application class processor running a modern Linux kernel and user-space distribution.
- A single host system. i.e. no consideration for multi-port NVMe devices.
- No virtualization in this example.
- We will leverage existing PCIe and NVMe methods around security.
B.4.2 Computational Storage Drive

Figure B.4.1 describes the elements comprising a CSD. There is a Computational Storage Processor (CSP) combined with a SSD. The two communicate through a shared PCIe peer-to-peer bar. This example would also apply in a case with two discrete devices on the same PCIe fabric: a CSP w/ shared P2P bar and a separate SSD.

This example does not explicitly address multiple CSDs in one system, but the architecture, including the software drivers for OpenCL and NVMe all support multiple devices in one system.
B.4.3 Theory of Operation

This section walks through the Theory of Operation (Section 4) and applies it in a specific fashion to this illustrative example. The three main phases are:

- Discovery
- Configuration
- Usage

In this illustrative example, there are no changes required to the NVMe standard. The NVMe controller and namespaces are discovered, configured and used as per the NVMe standard.

Discovery, configuration and use of the CSS are performed using the OpenCL API.

B.4.4 Discovery

NVM Express (NVMe) over PCI Express (PCIe) and OpenCL already have very robust discovery and creation processes.

PCIe devices (i.e. an NVMe controller) can be discovered by the host at power-up time via PCIe bus enumeration or at run-time via a hot-plug event and bus rescan. A modern Linux operating system is capable of detecting PCIe devices via both the methods mentioned above meaning discovery of a new NVMe controller and OpenCL CSP can be done at any time in a running system. The OS takes care of binding the relevant drivers to the discovered devices. In this case the NVMe driver and the OpenCL Runtime driver.

B.4.4.1 NVMe Function Discovery

Once an NVMe controller has been detected the NVMe driver can be used to discover the capabilities of said controller via the NVMe Identify Admin command. NVMe has the concept of namespaces, which refers to discrete resources behind an NVMe controller and one controller is able to support many namespaces as well as being able to (optionally) create and delete namespaces (via namespace management commands). Therefore:

1. PCIe enumeration is able to be used to discover the NVMe controller.
2. The Linux NVMe driver binds to this PCIe device and controller capabilities and namespaces are discovered using standard NVMe Identify Admin commands. In our example the driver will discover the NVMe-based CSD supports one namespace:
   a. Namespace 1: Conventional LBA based storage namespace.

Once the NVMe driver has discovered the namespace exists it can perform an Identify Admin command against it.

At this point the information about the namespaces on this NVMe-based CSD can be displayed in admin tools like nvme-cli (assuming said tools are updated). As an example the output of nvme-cli list for such a device might look like:
B.4.4.2 OpenCL CSP Function Discovery

The first two stages of discovery for the CSP mirror those for the NVMe controller:

1. PCIe enumeration is used to discover the OpenCL CSP.
2. The Linux OpenCL Runtime driver binds to this PCIe device.

The OpenCL Runtime driver implements platform-specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

Example API functions that facilitate discovery:

- `clGetPlatformIDs(..)
- `clGetPlatformInfo(…)
- `clGetDeviceIDs(…)
- `clGetDeviceInfo(…)

B.4.5 Configuration – Explicit Mode

OpenCL provides a rich set of APIs to configure the CSP. These include mechanism to partition a device, create contexts, download kernels, allocate memory, and more.

B.4.6 Usage – Storage Direct

Direct access to the storage is provided via the traditional NVMe stack.

B.4.7 Usage – Explicit Mode Computational Storage

In explicit mode one operation requires at least two commands: a NVMe IO command and an OpenCL compute command. As an example consider a kernel that reads data from the storage media, performs a transformation, and returns the result to the host:

1. Host issues a NVMe read IO command against the storage namespace. This command specifies where to read the data using any NVMe supported command format and provides a target write address within the P2P bar of the CSD. This command completes
when the output has been placed in the CSD P2P memory. As per any NVMe command this results in a Completion Queue Entry (CQE) and (optionally) a PCIe interrupt.

2. Host issues an OpenCL kernel invocation command to the CSP function of the CSD. Prior to issues this command it is assumed that the desired kernel has been programmed on the CSP as part of the configuration stage. The kernel reads data from the CSD P2P memory and writes results back to the P2P memory.

3. (optional) The host can then issue a command to read the result from the P2P memory.

Note that other, more complicated, IO operations are possible. For example, the result from step 2 could be written back to the storage using an additional NVMe write command.

B.4.8 Configuration – Transparent Mode

It is possible to support transparent kernels in this model as well. In this case OpenCL is used to download kernels that are inserted inline in the NVMe controller datapath. These kernels could optionally be applied on all namespaces or only on specified namespaces.

B.4.9 Usage – Computational Storage Transparent Mode

In this transparent mode the host issues standard NVMe IO Commands (like read and write) against the namespace with a computational storage kernel mapped in.
B.5 FCSS Data Compression Example

B.5.1 Overview

Data compression aims to reduce the amount of data stored on a storage device. By reducing the amount of data being physically written to or read from storage media (e.g., flash memory chips), data compression can also improve the storage device IO performance and lifetime span, in addition to storage cost saving.

Different from off-loading compression/decompression through a specific API, Data Compression FCSS carries out compression/decompression on the IO data path, transparently from the host (i.e., host simply issues normal IO write/read requests to a CSD with a Data Compression FCSS, without calling any other specific API). In order to materialize the storage cost reduction, Data Compression FCSS must expose a logical storage space that is larger than its internal physical storage space. However, due to the runtime variation of data compressibility, it is possible that the physical storage space will be used up before the logical storage space is used up. Moreover, it is desirable for users to know how well different files or objects are compressed by Data Compression FCSS. Therefore, Data Compression FCSS must provide adequate reporting mechanism and observability in terms of data compression.

B.5.2 Theory of Operation

In order to successfully implement data compression as an FCSS that operates on all write data, a CSx must be able to communicate the ability to perform data compression as an FCSS. Once the ability is determined, the FCSS should allow the host to monitor and query the effect of data compression. This illustrative example will attempt to provide the initial framework of the following steps needed to successfully allow for a CSx to perform data compression as an FCSS.

B.5.3 Discovery

In order to determine if a CSx supports the Data Compression FCSS, the CSx must first be discovered as a CSx. If the CSx is a Data Compression FCSS, then the CSx must also indicate if the Data Compression FCSS allows configuration. If the CSx allows for configuration parameters to the Data Compression FCSS, then the configurable parameters must be shared. The following is a possible list of configurable parameters:

1. Supported compression block sizes – The acceptable values of compression data block size. The block size can be set globally for the entire Data Compression FCSS, be set separately for each logical storage space region of the Data Compression FCSS, or even be set for each individual write request sent to the Data Compression FCSS.
2. Supported maximum logical storage space – The maximum size of the logical storage space that can be exposed by the Data Compression FCSS.
3. Monitoring – The type of information to collect during runtime operation. Such information can include the runtime physical storage capacity usage of the entire Data
Compression FCSS, and the runtime physical storage capacity usage of any given logical storage space.

Note that discovery may also return the default values of the configurable parameters.

B.5.4 Configuration

Once the configurable parameters of a CSx that supports the Data Compression FCSS are discovered, the Data Compression FCSS can be configured if allowed. It is possible that the Data Compression FCSS will have default values that the user may not want to override, and configuration is not necessary. If the user wants to configure the Data Compression FCSS, then the parameters returned by discovery need to be set and sent to the Data Compression FCSS before the data compression operation is performed.

B.5.5 Monitoring

As the data is being written to the Data Compression FCSS, the user can get statistics on the compression. The following are possible statistics to monitor:

1. Current physical storage space usage of the entire FCSS
2. Current physical storage space usage of any logical storage space region of FCSS
3. Lifetime data compression ratio of all the data written to FCSS so far
4. Data compression ratio over a specified amount of data that has been written to the FCSS
B.6 FCSS Data Filter Example

B.6.1 Overview

As one important category of operations in data analysis, data filtering aims to filter out the data that are not needed by a query. In conventional practice, a CPU (or GPU) is responsible for data filtering, which requires transferring all of the raw data from the storage device into the CPU (or GPU) memory. The unique feature of Data Filter FCSS is to push down the data filtering operation from the CPU (or GPU) to a storage device. It can offload the data filtering operation from the CPU (or GPU), leading to higher system performance, and less host resource contention in terms of CPU/GPU cycles, memory capacity and bandwidth, and I/O bus bandwidth. The following example illustrates using data filter FCSS to carry out in-storage data filtering, as shown in Figure 1. A table with four columns is stored in the Data Filter FCSS that receives a request “SELECT ID where State=CA” that seeks the IDs of all the table entries in which State equals to CA. As illustrated in the figure, the Data Filter FCSS fetch all the table entries from the storage media, extracts each table entry, and checks whether the 4th column in the entry equals to CA. After scanning the entire table, the Data Filter FCSS returns the IDs of all the matching entries.

![Data Filter FCSS Diagram](image)

Figure B.6. 1 An example to illustrate the function of data filter FCSS
B.6.2 Theory of Operation

In order to successfully implement data filter as an FCSS, a CSx must be able to communicate the ability to perform data filter as an FCSS. Once the ability is determined, the FCSS should allow the host to query the supported filter functions and data schema. This illustrative example provides the initial framework of the following steps needed to successfully allow for a CSx to perform data filter as an FCSS.

B.6.3 Discovery

The steps in discovery are:

a) Discover if the Data Filter FCSS allows configuration.
   a. If the Data Filter FCSS allows configuration, then discover the configurable parameters. Discovery may also return the default values of the configurable parameters.
   b. The following is list of possible parameters:
      i. Supported data formats (e.g. Parquet/ORC, JSON, XML formats). In order to perform data filtering, Data Filter FCSS must be able to understand the data format specified by the host.
      ii. Supported data types (e.g., ASCII string, integer)
      iii. Supported filtering operations (e.g. >,<,=)
      iv. Failover – The action to take (e.g. rollback to host-based data filter) if data filter is interrupted.

b) Discover operational attributes of the Data Filter FCSS
   a. Monitoring Capabilities – The type of process information that can be collected during data filtering. Such possible information includes the I/O rate of the data filtering operation, the size of the data processed, the size of the data remaining to be processed, the percentage complete, etc.
   b. Operational Interruption Capability – Whether or not the data filtering operation can be interrupted for purpose of either suspending, abandoning, or resuming the request.

B.6.4 Configuration

Once a CSx that supports the Data Filter FCSS is discovered, the Data Filter FCSS can be configured if allowed. It is possible that the Data Filter FCSS will have default values that the user may not want to override, and configuration is not necessary. If the user wants to configure the Data Filter FCSS, then the parameters returned by discovery need to be set and sent to the Data Filter FCSS before the data filtering operation is performed.

B.6.5 Operation

To utilize data filter FCSS to carry out in-storage data filtering, host passes enough information about the data filter operation to the data filter FCSS. Accordingly, data filter FCSS performs in-
storage data filtering and returns the results back to the host. The information about the data filter operation may include:

1. The address of to-be-processed data
2. The data format (e.g., MySQL, Parquet) and schema (e.g., the number of columns in the table, and data type of each column)
3. The specific filter operation to be performed
4. The host memory address for the returned data

B.6.6 Monitoring

As the data object is being processed by the data filter FCSS, the user can get status and statistics on the process. The following are possible status and statistics to monitor (not all items are required; not all items are unique to this FCSS):

8. The I/O rate of the data filter operation
9. Current amount of data processed by the data filtering operation
10. Current amount of data remaining to be processed by the data filtering operation
11. The percentage of completion of the data filtering operation
12. Success or failure of the operation
13. Existing state of the operation such as paused, interrupted, resumed, etc.

Based on the status and statistics, the user can then determine if the operation needs to be paused, abandoned or resumed.
B.7 Scatter Gather FCSS

B.7.1 Overview

Many computational storage use cases require offloading and coordinating data movement, including data flows between multiple CSSes. Offloading data movement has long-standing precedents in computer architectures, specifically, Direct Memory Access (DMA).

Use cases for DMA in computational storage include:

a. Host to CSx: Gathering a series of blocks of data from host memory, which are potentially scattered across host memory space, to be transferred to a CSx (this use case is already implemented for NVMe-based CSx via PRP/SGLs);
b. CSx to Host: Scattering a series of blocks of data from a CSx to be transferred into host host memory, where destination addresses are potentially scattered across host memory space (this use case is already implemented for NVMe-based CSx);
c. Peer-to-Peer: Scatter-Gather between CSxes without having to traverse host memory (this use case is partially implemented for some NVMe-based CSx);
d. Fanout: Scatter-Gather to multiple CSxes; and
e. Pipelining: Managed flows between three or more devices, including flow control.

All of these functions need to be provided such that they can be performed to and across multiple supported fabrics.

In addition to these data movement functions, there is also a need to provide a standardized:

a. representation for applications to specify desired data movements;
b. representation for application to specify and compose data pipelines;
c. mechanism to handle private and internal busses and devices;
d. mechanism to handle flow control and back-pressure;
e. mechanism to handle buffer management for transformational processes where the size of output data changes from the size of input data; and
f. mechanism to handle errors and failure recovery.

In addition to host-managed DMA, there is a need for a FCSS to provide and coordinate DMA between CSxes.

B.7.1.1 Representation of a Data Movement Request (DMR)

For an application to express desired data movement in an interoperable way, a standardized representation of the desired data movement is required. This illustrative example proposes use of CBOR as a compact binary representation of a JSON-based data structure that describes a desired data movement request (DMR).

For readability, this document will use the corresponding JSON representation of the CBOR data structure.
Once constructed by an application, a DMR can be translated into a fabric-specific data movement operations, decomposed into topology-specific flows, and offloaded to DMA-capable CSxes.

```json
{
  "ep": {
    <One or more endpoint definitions>
  },
  "flows": [
    <One or more flow definitions>
  ]
}
```

Figure B.7. 1 Top-level DMR structure

B.7.1.2 Endpoints

Endpoint definitions specifies the entities involved in the data movement request. These can include host memory, fabric-accessible CSx memory, storage devices, internal (non-fabric accessible CSx memory), etc.

```json
"ep": {
  "1": {
    "etype": "NVMe"  <Entity Type, e.g. NVMe Namespace, Host Memory, etc. The Entity Type indicates which fields will be present>
    "ncuid": "144D...."  <NVMe Controller Universal ID>
    "nnuid": "20000..."  <NVMe Namespace Universal ID>
    "nntyp": "lba"  <NVMe Namespace Type>
  }
}
```

Figure B.7. 2 Example endpoint definition structure for an NVMe endpoint

This example endpoint uniquely specifies an endpoint named “1”, which is an NVMe namespace on a given NVMe controller, including the type of the namespace. Other endpoints include host memory (“mem”), which allows for a base address to be specified.

Endpoints also specify the addressing mode. The following address types are anticipated:

- **Memory**: A contiguous range of memory addresses space
- **Block**: A contiguous range of fixed-length blocks of data within an address space
- **Key**: The contents of a given key in a key/value space

Endpoints within internal devices and busses can also be specified when using a scatter-gather CSS to offload data movement.

B.7.1.3 Flows

Flow definitions specifies what data should be moved between endpoints.
Source and destination locations are expressed as <endpoint, start-address, length> triples. The address is dependent on the endpoint type, and when an endpoint defines a default block size, the length can be omitted if the default block length is to be used.

B.7.1.4 Example DMRs

The following example DMRs illustrates some of the use cases:

```json
{
  "ep": {
    "1": { "eptyp": "mem", "mbadr": "00002020FFF00000" },
    "2": { "eptyp": "NVMe", "ncuid": "2366......", "nnuid": "ACDE48234567ABCD", "nntyp": "lba" },
    "3": { "eptyp": "NVMe", "ncuid": "53F2......", "nnuid": "00A0BF398F8912AA", "nntyp": "lba" },
    "4": { "eptyp": "NVMe", "ncuid": "2354......", "nnuid": "00A0BF3923DS9823", "nntyp": "lba" }
  },
  "flows": [ {
    "src": [ ["1", "0", "2048"], ["1", "6144", "2048"], ["1", "2048", "2048"], ["1", "8192", "2048"], ["1", "4096", "2048"], ["1", "10240", "2048"] ],
    "dst": [ ["2", "4214"], ["3", "4214"], ["4", "4214"] ]
  } ]
}
```

Figure B.7. 3 Flow definition structure

Figure B.7. 4 Scatter DMR
This example scatters data from a BAR to three NVMe namespaces, as illustrated in figure B.7.5:

```json
{
  "ep": {
    "1": {
      "eptyp": "mem",
      "mbadr": "00002020FFFF0000"
    },
    "2": {
      "eptyp": "css",
      "cssID": "273846333"
    },
    "3": {
      "eptyp": "css",
      "cssID": "981234784"
    },
    "4": {
      "eptyp": "NVMe",
      "ncuid": "2366.....",
      "nnuid": "ACDE48234567ABCD",
      "nntyp": "lba"
    },
    "5": {
      "eptyp": "NVMe",
      "ncuid": "53F2.....",
      "nnuid": "00A0BF398F912AA",
      "nntyp": "lba"
    },
    "6": {
      "eptyp": "NVMe",
      "ncuid": "2354.....",
      "nnuid": "00A0BF3923DS9823",
      "nntyp": "lba"
    }
  },
  "flows": [
    {
      "src": [
        ["1", "0", "4096000"]
      ],
      "dst": [
        ["2"
      ]
    },
    {
      "src": [
        ["2"
      ],
      "dst": [
        ["3"
      ]
    },
    {
      "src": [
        ["3"
      ],
      "dst": [
        ["4", "4214", "2048000"],
        ["5", "4214", "2048000"],
        ["6", "4214", "2048000"
      ]
    }
  ]
}
```

Figure B.7. 5 Example Scatter DMR

Figure B.7. 6 Pipeline DMR
This example shows a pipeline where data is streamed from host memory to a first CSS (compression), streamed to a second CSS (erasure coding), then scattered across three NVMe namespaces, as illustrated in the below diagram:

![Diagram of pipeline](image)

**Figure B.7. 7 Example Pipeline DMR**

**B.7.2 Scatter-Gather CSS Theory of Operation**

While the above examples can all be coordinated by the host, it is often advantageous to offload data movement to a CSx. This is implemented by a Scatter-Gather CSS (SG CSS).

This illustrative example will attempt to provide the initial framework of a FCSS that implements scatter-gather.

**B.7.2.1 Discovery**

In order to determine if a CSx provides a SG FCSS, the CSS must first be discovered. The discovery process will also return configuration and status information as specified in the normative sections of this specification.

At this time no configurable parameters are anticipated.

Note that discovery may also return the default values of the configurable parameters as well as a list of private endpoints that the Scatter-Gather FCSS can address.

**B.7.2.2 Configuration**

Once a Scatter-Gather FCSS is discovered, it can be configured. At this time no configurable parameters are anticipated.
B.7.2.3 Operation

The Scatter-Gather FCSS will begin processing a DMR when the DMR CBOR is sent to the CSS’s NVMe input queue, or written into PCIe memory (if it is not NVMe-based). When the DMR is received, it is validated and executed, moving data as requested.

The operation of the Scatter-Gather FCSS may be interrupted if allowed. Otherwise, the user will need to wait for the operation to complete or fail to determine the next course of action.

B.7.2.4 Monitoring

As the DMR is being processed, the user can get status on the data movement. The following are possible status to monitor:

a. Flow progress; and
b. Error status.

Based on the status, the user can then determine if the operation needs to be paused, abandoned or resumed.