RAMAIAH
Institute of Technology

**Department of Computer
Science & Engineering**

A Project Report on

ARRAY CONTEXT EXTENSION FOR PYDAOS

Submitted in partial fulfillment of the requirements for the award of the degree of

Bachelor of Engineering in Computer Science & Engineering

By

M Vinay	1MS21CS068
Manoj V L	1MS21CS072
Bharath Vamsi D	1MS21CS046
Malavika Dileep	1MS21CS069
Aishwarya G	1MS21CS047

Under the guidance of

Mrs. Chandrika Prasad
Assistant professor, Dept of CSE

M S RAMAIAH INSTITUTE OF TECHNOLOGY
(Autonomous Institute, Affiliated to VTU)
BANGALORE-560054
www.msrit.edu
2024

M. S. RAMAIAH INSTITUTE OF TECHNOLOGY, BANGALORE – 560 054
(Autonomous Institute, Affiliated to VTU)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

Certified that the project work titled "Array Context Extension for Pydaos" carried out by– 1MS21CS068 M Vinay, 1MS21CS072 Manoj V L, 1MS21CS046 Bharath Vamsi D ,1MS21CS069 Malavika Dileep and 1MS21CS047 Aishwarya G are bonafide students of M. S. Ramaiah Institute of Technology, Bengaluru, in partial fulfillment of the course Mini Project CSP67 during the term March- June 2024. The project report has been approved as it satisfies the academic requirements for the aforesaid course. To the best of our understanding, the work submitted in this report is the original work of students.

Project Guide   **Head of the Department**

Mrs. Chandrika Prasad   **Dr R China Appala Naidu**

External Examiners

Name of the Examiners: **Signature with Date**

1.
2.

DECLARATION

We, hereby, declare that the entire work embodied in this mini project report has been carried out by us at M. S. Ramaiah Institute of Technology, Bengaluru, under the supervision of **Chandrika Prasad, Assistant Professor**, Department of CSE. This report has not been submitted in part or full for the award of any diploma or degree of this or to any other university.

Signature   Signature

M Vinay   Manoj VL

1MS21CS068   1MS20CS072

Signature   Signature

Bharath Vamsi D   Malavika Dileep

1MS21CS046   1MS21CS069

Signature

Aishwarya G

1MS21CS047

ACKNOWLEDGEMENT

ABSTRACT

This project aims to enhance the efficiency of data handling within the DAOS (Distributed Asynchronous Object Storage) system by leveraging the pyDAOS interface. DAOS is a high-performance storage system designed for scalable, distributed data management. Our approach focuses on three key areas: optimizing large I/O operations through chunking, automating resource selection for optimal performance, and simplifying system administration via automatic server discovery and configuration.

Firstly, we conduct a high-level study on the functionality and performance of pyDAOS when used with a key-value store, analyzing how pyDAOS interacts with the DAOS KV store to identify optimization opportunities. Based on these insights, we design and implement an abstracted API that simplifies large I/O read/write operations, making it easier for developers to handle extensive data transfers efficiently.

To enhance resource allocation, we automate the selection of pools and containers within DAOS. This involves developing mechanisms to dynamically choose the most suitable pools based on the number of targets and available free space, ensuring balanced load distribution and optimal resource utilization.

The project includes several Python scripts tailored for different tasks: from basic file operations and bulk uploads to advanced chunking strategies and metadata management. These tools collectively contribute to a more efficient and user-friendly DAOS environment, capable of handling large-scale data operations with improved performance and reduced administrative overhead.

Our results demonstrate that by combining chunking techniques with automated resource management and server discovery, we can significantly improve the efficiency, scalability, and ease of use of DAOS for large I/O operations, making it a robust solution for modern data-intensive applications.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 General Introduction

The introduction present here has been given by the HPE company. It is stated as follows: "Distributed Asynchronous Object Storage (DAOS) is a modern storage solution designed for scalability and high performance. Pydaos, built for Python, facilitates seamless interaction with DAOS using a Key-Value store mechanism". Our project aims to enhance pyDAOS by implementing efficient data handling techniques, focusing on chunking for large I/O operations, automating resource selection for optimal performance, and simplifying system administration through automatic server discovery and configuration.

1.2 Problem Statement

Design and implement the array context extension in pydaos

1.3 Objectives of the Project

- Perform a high level study on working of pydaos with KV store.
- Design and implement an abstracted API for large IO read/write operations.
- Automate pool and container selection for optimized resource allocation.
- Simplify system administration with automatic server discovery, configuration, and resource selection.

1.4 Project deliverables

- **Design Document:**

    - Detailed documentation outlining the design specifications, architecture, and functionalities of the KV store for pydaos, focusing on addressing the specific challenges and requirements outlined in the project objectives.

- **Implementation within pydaos:**

    - Develop Python code to implement the functionalities required within the pydaos framework, aligning closely with the design document and project objectives.

- **Integration with pydaos:**

  - Integrate the developed functionalities seamlessly with pydaos to enhance array handling capabilities as per the project's objectives.

- **Performance Assessment Report:**

  - Conduct a comprehensive performance assessment of pydaos with the integrated functionalities. The report will include benchmarks, analysis, and optimization recommendations tailored to meet the project's performance improvement goals.

- **Code Repository Management:**

  - Check the codebase into the designated HPE GitHub repository for version control, collaboration, and ensuring code integrity throughout the development process.

- **Demonstration and Presentation:**

  - Prepare and deliver a demonstration showcasing the pydaos interface with the integrated functionalities. The demonstration will highlight the functionality, benefits, and practical application, aligning with the project's core objectives.+

1.5   Current Scope

- Design and implement an abstracted API for write and reads targeting large I/O sizes.
- Develop algorithms to chunk I/O requests larger than 1MB into 1MB sizes.
- Create a metadata management system to track chunking information for each I/O request.
- Integrate the API with the Key-Value (KV) store within the DAOS system.
- Conduct performance testing to evaluate the efficiency of the abstracted API.

1.6   Future Scope

- Conduct a comprehensive study on the working principles and architecture of pydaos integrated with a Key-Value (KV) store and the proposed Array context extension.

- Design and implement the Array context extension within the pydaos framework, focusing on efficient array handling and data management.

- Develop algorithms and methods to integrate the Array context extension seamlessly with existing pydaos functionalities.

- Demonstrate the functionality of the pydaos interface with the Array context through provided AI/ML programs, showcasing its capabilities in handling arrays.

- Perform an in-depth assessment of I/O performance, evaluating the impact of the Array context extension on data retrieval and storage operations.

2. PROJECT ORGANISATION

2.1   Software Process Models

The software process model chosen for the implementation of the described solution, Enabling Efficient Data Handling in DAOS: A pyDAOS Approach with Chunking, Automated Resource Selection, and Server Discovery, aligns well with the Waterfall model.

The Agile model adopts Iterative development. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable and can be completed within a couple of weeks only. At a time one iteration is planned, developed, and deployed to the customers. Long-term plans are not made.

Rationale for Choosing the Agile Model:

- **Iterative and Incremental Development:** The Agile model's iterative and incremental approach is well-suited for dynamic projects where requirements evolve or are not fully defined initially. It allows for flexibility and adaptability, enabling continuous improvement and refinement throughout the development lifecycle.

- **Frequent Deliverables**: Agile methodology focuses on delivering working software in short iterations, known as sprints. This results in frequent deliverables that can be reviewed and validated by stakeholders, promoting early feedback and collaboration.

- **Adaptability to Change**: Agile practices such as daily stand-ups, sprint planning, and retrospectives facilitate quick adaptation to changing requirements, priorities, or emerging technologies. This agility ensures that the project stays aligned with evolving business needs.

- **Continuous Integration and Testing**: Agile emphasizes continuous integration and testing, enabling early detection of issues and faster resolution. This proactive approach enhances product quality and reduces risks associated with last-minute integration problems.

- **Customer Collaboration:** Agile encourages ongoing collaboration with customers or stakeholders through regular feedback cycles and demos. This ensures that the delivered product meets user expectations and adds value from the end-user perspective.

- **Emphasis on Working Software:** The primary focus of Agile is on delivering working software increments, prioritizing tangible results over extensive documentation. This approach leads to quicker time-to-market and allows for incremental improvements based on real-world usage and feedback.

2.2   Roles and Responsibilities:

The project team is structured with distinct roles: Requirements Analyst conducts a study on pydaos with KV store, Systems Architect designs the working of KV store, Software Developer implements it adhering to coding standards, Project Manager oversees progress and stakeholder feedback, while Quality Assurance ensures software quality and reliability through rigorous testing. Detailed description is given in Table 2.1

Table 2.1: Roles and Responsibilities

Role	Responsibilities
Requirements Analyst (All)	- Conduct a high-level study on pydaos with KV store. - Define specific requirements, set performance criteria, and document findings.
Systems Architect (All)	- Create a detailed design document for the KV store integration within pydaos. - Outline architecture, data structures, algorithms, and integration points. - Obtain stakeholder approval for the design plan.
Software Developer (All)	- Implement the KV store integration within pydaos based on the approved design. - Follow coding standards, and ensure version control. - Collaborate with the team for seamless implementation.

Project Manager (All)	- Demonstrate the pydaos interface with the integrated KV store capabilities.
	- Gather stakeholder feedback, conduct performance assessment, and document results.
	- Oversee project progress, ensure alignment with objectives, and address any challenges.
Quality Assurance (All)	- Test the KV store integration for functionality, compatibility, and performance within pydaos.
	- Identifying and reporting bugs or issues.
	- Ensuring overall software quality and reliability.

3. LITERATURE SURVEY

3.1   Introduction

High-performance computing (HPC) has revolutionized numerous fields, from scientific research to financial modeling, by providing immense computational power to tackle complex problems efficiently. At the heart of HPC systems lies the storage infrastructure, which plays a critical role in storing, managing, and retrieving vast amounts of data generated by computational tasks. As HPC applications continue to evolve, the demand for storage solutions capable of handling massive datasets and supporting high-speed data access becomes increasingly paramount. This literature survey delves into various storage technologies and approaches utilized in HPC environments, aiming to provide insights into their characteristics, advantages, and challenges. By examining the latest research findings, industry reports, and technological advancements, this survey seeks to elucidate the current state-of-the-art in HPC storage and identify future directions for innovation and optimization.

3.2   Related Works with the citation of the References:

A file system organizes data and manages how it is stored and retrieved on a storage device. File systems are essential for HPC environments where the efficient organization of large datasets is crucial. TechTarget provides a comprehensive definition of file systems, highlighting their role in data management, access methods, and structure [1]. File systems such as Lustre and GPFS (IBM Spectrum Scale) are widely used in HPC for their ability to handle large-scale data with high performance.

HPC involves the use of supercomputers and parallel processing techniques to solve complex computational problems. According to NetApp, HPC storage must accommodate high data throughput and low latency to support the intensive workloads typical in scientific research and simulations [2].

Storage Types

1. **File Storage**: File storage is the traditional method of storing data in a hierarchical structure. It is suitable for a wide range of applications and provides good performance for sequential data access [3].

2. **Block Storage**: Block storage divides data into fixed-sized blocks and manages them individually. This type of storage is known for its high performance and flexibility, making it ideal for databases and transactional systems .

3. **Object Storage**: Object storage manages data as objects, each containing the data itself, metadata, and a unique identifier. This approach is highly scalable and suitable for unstructured data, such as multimedia files and large datasets [5].

Pure Storage explains the critical role of HPC storage in ensuring high performance, scalability, and reliability. It discusses how storage solutions must evolve to meet the demands of modern HPC workloads, emphasizing the importance of parallel file systems and high-throughput storage architectures [4].

Red Hat conducted extensive research on the advantages of block storage solutions in HPC environments. Their findings emphasized the flexibility and performance benefits of block storage, particularly for applications requiring high-speed data access and low-latency I/O operations. By leveraging block storage technologies, organizations can optimize data processing workflows, improve resource utilization, and achieve higher levels of computational efficiency in HPC clusters.

In addition to block storage, object storage solutions have emerged as viable alternatives for HPC workloads. NetApp [6] explored the scalability and suitability of object storage architectures for managing large volumes of unstructured data in HPC environments. Object storage offers inherent scalability, fault tolerance, and ease of management, making it well-suited for storing and accessing diverse data types in distributed computing environments. Their research highlighted the potential of object storage to address the growing storage demands of modern HPC applications while providing cost-effective solutions for long-term data retention and archival.

The Distributed Asynchronous Object Storage (DAOS) project, as outlined in its architecture overview [7], represents a significant advancement in HPC storage technology. Developed by

Intel in collaboration with industry partners, DAOS is designed to address the scalability and performance challenges of traditional storage systems in HPC environments. Its architecture emphasizes modularity, scalability, and efficiency, making it well-suited for exascale computing and data-intensive scientific applications.

Intel [8] provided a comprehensive solution brief on high-performance storage for HPC, highlighting the key considerations and best practices for designing and deploying storage solutions in HPC environments. Their document emphasized the importance of understanding workload characteristics, storage requirements, and performance metrics to tailor storage architectures to specific HPC applications. Additionally, they discussed the benefits of distributed storage architectures and the role of emerging technologies, such as non-volatile memory and persistent memory, in improving storage performance and efficiency in HPC clusters.

Moreover, the integration of DAOS with Python through PyDAOS, as demonstrated in Intel's case study [9], showcases its versatility and ease of use for developers. By providing Python bindings and APIs, PyDAOS enables developers to leverage the capabilities of DAOS for building scalable and efficient data-intensive applications in Python.

Overall, the research and industry literature on HPC storage systems reflect a dynamic and rapidly evolving landscape. As the demand for computational resources and data-intensive applications continues to grow, optimizing storage infrastructure becomes increasingly critical for unlocking the full potential of high-performance computing. By leveraging advanced storage technologies, optimization techniques, and collaborative research efforts, organizations can address the evolving challenges of HPC storage and pave the way for future innovations in computational science and engineering.

3.3   Conclusion of Survey

In conclusion, this literature survey has provided comprehensive insights into the diverse and evolving landscape of high-performance computing (HPC) storage systems. By examining an array of storage technologies and approaches, including file systems, block storage, object storage, and advanced solutions like the Distributed Asynchronous Object Storage (DAOS), it has underscored the pivotal role of robust storage infrastructure in supporting and accelerating

modern computational workloads. The in-depth analysis of these technologies reveals how each type addresses different aspects of performance, scalability, and reliability, catering to the varied demands of scientific research, engineering simulations, and other data-intensive computing tasks. Through the survey, it is evident that optimizing storage solutions is not merely a support function but a fundamental component in the architecture of HPC systems, directly influencing the efficiency and success of computational endeavors.

Furthermore, the survey has illuminated emerging trends, challenges, and opportunities within the HPC storage domain. Notable trends include the increasing adoption of hybrid storage solutions that combine the strengths of different storage types, the integration of artificial intelligence and machine learning to enhance storage management, and the growing importance of data security and resilience. The challenges, such as managing the exponential growth of data, ensuring low-latency access, and maintaining cost-effectiveness, present significant opportunities for innovation and advancement. The insights gained from this survey pave the way for future research aimed at overcoming these challenges and harnessing new technologies to further optimize HPC storage. As the field of HPC continues to progress, the continual evolution and enhancement of storage solutions will be crucial in enabling breakthroughs and sustaining the momentum in scientific discovery and technological development.

4. PROJECT MANAGEMENT PLAN

4.1   Schedule of the Project

The Gantt chart as shown in fig 4.1 depicts the progress of a mini project named HPE-CTY 2024. It illustrates the schedule for various project tasks including research, development, integration, testing, and documentation. The timeline spans from March 10th, 2024 to June 5th, 2024**.**

From March 10th, 2024 to April 5th, 2024 to understand the problem statement thoroughly. Development follows, taking up six weeks from April 5th to May 10th. Integration begins partially during development, starting on April 26th and continuing for four weeks until May 23rd. Testing commences on May 16th and lasts for two weeks, encompassing unit testing, system testing, performance evaluation, and integration issue monitoring. Finally, comprehensive project documentation takes place over the last two weeks, concluding on June 5th, 2024. This structured approach ensures a well-defined workflow for the HPE-CTY 2024 mini-project.
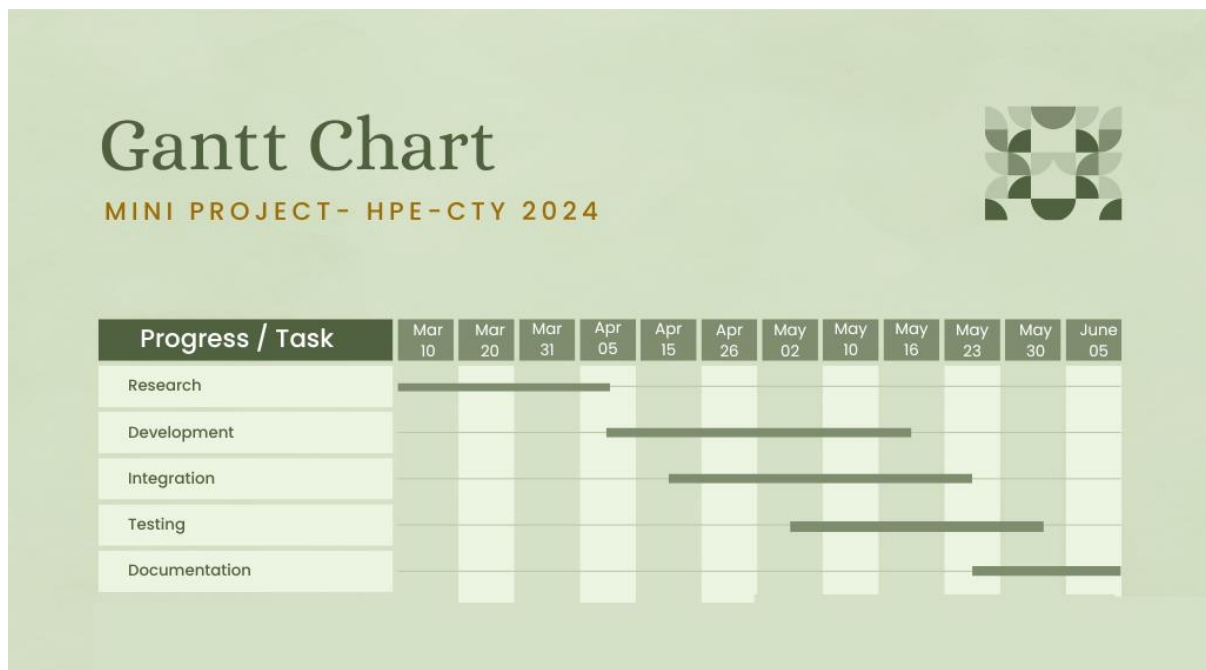
Fig. 4.1: Gantt Chart for the Project Management Plan

4.2   Risk Identification

The following categories of risks can safely be assumed to be non-existent:

- **Technical Risks:**

  - Inadequate Understanding of pyDAOS and DAOS: Insufficient understanding of pyDAOS and its interaction with the DAOS system may lead to incorrect implementation and project delays.
  - Integration Complexity: Complex integration requirements between pyDAOS and the DAOS KV store could result in integration issues and software instability, impacting system performance.
  - Performance Bottlenecks: Potential performance bottlenecks in the implemented functionalities may cause reduced I/O efficiency and system slowdowns, affecting the overall project success.

- **Documentation and Knowledge Risks:**

  - Lack of Comprehensive Documentation: Inadequate documentation for design, implementation, and usage of the enhanced pyDAOS functionalities may lead to maintenance challenges and knowledge gaps within the team.
  - Testing Coverage: Insufficient testing coverage for the implemented enhancements could result in undetected bugs and performance issues during deployment, affecting the system's reliability.

- **Codebase Management Risks:**

  - Codebase Errors and Version Control: Without proper version control systems and code reviews, codebase errors, functionality breakdowns, and version conflicts may occur, impacting the stability and maintainability of the project.

- **Resource Risks:**

  - Skill Set Availability: Lack of skilled developers or resources familiar with pyDAOS, DAOS, Python programming may impact the project's progress and the quality of implementation.
  - Resource Allocation: Challenges in dynamically selecting and managing resources within the DAOS system may affect performance optimization and scalability.

- **Business Risks:**

- Changes in Project Scope: Mid-development changes in project scope or requirements may lead to project delays and increased workload, impacting project timelines and deliverables.
- Budget Constraints: Budget constraints or unexpected costs related to software development tools, resources, or training may affect project timelines, resource allocation, and overall project quality.

5. SOFTWARE REQUIREMENT SPECIFICATIONS

5.1 Purpose

The purpose of this project is to enhance the efficiency and usability of the Distributed Asynchronous Object Storage (DAOS) system by leveraging the pyDAOS interface. By conducting a high-level study on pyDAOS with its Key-Value (KV) store the project aims to identify optimal methods for data storage and retrieval. It seeks to design and implement an abstracted API that simplifies large I/O operations, automate the selection of pools and containers for optimized resource allocation, and streamline system administration through automatic server discovery, configuration, and resource selection. These improvements are intended to make DAOS more powerful, scalable, and user-friendly for managing large-scale data storage and retrieval operations.

5.2 Project Scope

The scope of this project encompasses a comprehensive study of pyDAOS, focusing on its Key-Value (KV) store, to identify best practices for efficient data handling. It includes the design and implementation of a high-level, abstracted API that simplifies large I/O operations, ensuring scalability and ease of integration. The project also involves developing automation tools for pool and container selection, optimizing resource allocation, and creating mechanisms for automatic server discovery and configuration to facilitate seamless system administration. Additionally, it covers extensive testing, along with the creation of detailed documentation to support users and administrators in effectively utilizing the enhanced DAOS system.

5.3 Overall Description

5.3.1 Product Perspective :

In an effort to enhance the user experience and streamline data management within the Distributed Asynchronous Object Storage (DAOS) system, a Python-based approach utilizing pyDAOS was implemented. This project successfully achieved its objectives through the development of a high-level Python API. Said API effectively abstracts the complexities of the underlying DAOS API, thereby facilitating streamlined data read/write operations. Furthermore, functionalities were designed to automate pool and

container selection, ensuring optimized resource allocation based on specific data requirements. Finally, the project explored the automation of server discovery, configuration, and resource selection tasks, with the potential to significantly simplify system administration.

5.3.2   Product Functions:

The software is designed to perform the following functions:

- **Abstracted Python API for Data Management:** A high-level Python API was developed, effectively abstracting the complexities of the underlying DAOS API. This API empowers users to perform large-scale data read/write operations without requiring in-depth knowledge of DAOS internals.
- **Automated Resource Allocation for Data Storage:** Pool and container selection for data storage within DAOS was automated. Data characteristics were analyzed to identify the most suitable resources based on specific requirements, such as performance demands, capacity limitations, and data replication needs. This automation ensures optimized utilization of DAOS storage resources.
- **Automated DAOS Server Discovery and Resource Assessment:** The project implemented the automation of discovering active DAOS servers on a network. This process retrieves and analyzes their engine configurations, including the number of targets (storage units), CPUs, and threads. This information is crucial for optimizing data placement and resource allocation within the DAOS system.

5.3.3   User Characteristics

Users of the software include developers, data scientists, and system administrators familiar with DAOS, Python programming. They have a technical background and expertise in high-performance storage systems and data management.

5.3.4   Operating Environment

The software operates in a virtualized environment using CentOS with DAOS installed. It requires access to DAOS storage infrastructure and the ability to run Python programs for development and testing.

5.3.5 Assumptions and Dependencies

It is assumed that a DAOS system is already deployed and operational on the network. Additionally, the project relies on the availability of functional Python libraries like pyDAOS and yaml for interacting with the DAOS API and parsing configuration files. SSH access with appropriate credentials is necessary to connect to and retrieve information from the DAOS servers. Finally, for the automated server discovery and resource assessment to function, the DAOS servers should be discoverable on the network and configured to allow remote access for data retrieval. Additionally, the project relies on GitHub for code management and version control.

5.4 External Interfaces

5.4.2 User Interfaces:

Command-Line Interface:

The Large IO API provides a text-based CLI for user interaction. Users can execute commands with arguments to perform write and read operations for large files. The CLI is designed for user-friendliness, featuring clear and concise commands with descriptive arguments for specifying data paths, pool names, and container names. Comprehensive help documentation is available within the CLI or as a separate document, guiding users on available commands, arguments, and usage examples.

Additionally, some outputs and associated metadata generated by these operations are provided in JSON format. This structured data format facilitates easy exchange and parsing by other tools or applications for further processing or analysis.

5.4.3 Hardware Interfaces:

The Large IO API is designed to function seamlessly with various hardware configurations commonly used in HPC environments. This ensures compatibility with a range of storage solutions to cater to diverse user needs.

NVMe SSDs (Non-Volatile Memory Express Solid-State Drives): The API leverages high-performance NVMe SSDs to handle large data transfers efficiently. Their speed

makes them ideal for data-intensive tasks in scientific simulations and machine learning.

SCM (Storage Class Memory): The API also supports SCM, a persistent memory technology offering faster read/write speeds than traditional SSDs. This can benefit applications requiring frequent access to large datasets

5.4.4 Software Interfaces:

The Large IO API integrates seamlessly with the following software components:

DAOS Storage System: The API interacts with the DAOS storage system to efficiently store and retrieve large datasets. DAOS, a scalable and high-performance storage solution, is well-suited for HPC environments.

pyDAOS Library: This library provides a Python interface for interacting with DAOS. By integrating with pyDAOS, the Large IO API allows Python programs to leverage DAOS functionalities for data management tasks.

YAML Library: The YAML library was utilized to parse DAOS server configuration files, likely stored in YAML format on remote servers. This retrieval of information is essential for functionalities like automated resource allocation based on server capabilities.

SSH: In the event that the project implemented the automated server discovery and resource selection, SSH was used to establish secure remote access to DAOS servers. This access allows the program to connect, retrieve configuration data, and potentially manage server settings.

5.4.5 Communication Interfaces:

DAOS Communication Protocols:

The Large IO API relies on the existing communication protocols established within the DAOS system for data transfer. These protocols govern how data is transferred between clients (user programs) and storage targets (storage devices) within the distributed DAOS storage environment. They ensure efficient and reliable data movement across the network, facilitating high-performance data transfers.

5.5 System Features

5.5.1 Functional Requirements

Functional requirements play a pivotal role in defining the core capabilities and features of the kv store within the pydaos system. These requirements encompass essential functionalities such as data insertion and retrieval, dual storage support, integration with KV store, data manipulation operations, error handling, performance optimization, concurrency control, and configuration options, ensuring a robust and versatile framework for data management in high-performance computing environments.

Functional Requirements

The following functional requirements were established to deliver the Large IO API functionalities outlined in the project objectives.

1. Handling Large Data Sizes:

- Requirement ID : FR001

- Description: The Large IO API must effectively support read and write operations for data exceeding a predefined size threshold. This functionality allows users to manage large datasets efficiently.

- Acceptance Criteria:

    o The API successfully performs read and write operations on large data files.

    o Performance benchmarks demonstrate efficient handling of large data transfers.

2. Object Handling through Serialization:

- Requirement ID: FR002

- Description: The Large IO API must enable users to manage data objects through a serialization mechanism. Serialization allows the API to handle various data formats and facilitates data exchange between the user's environment and the DAOS system.

- Acceptance Criteria:

  o The API offers methods for serialization and deserialization of data objects.

  o The API supports serialization of user-defined data structures.

3. Pool and Container Detection

- Requirement ID : FR003

- Description: The Large IO API must possess the capability to discover existing DAOS storage pools and containers. This functionality is essential for identifying available storage resources within the DAOS system. (FR04 relies on FR03 to identify suitable storage locations)

- Acceptance Criteria:

  o The API successfully detects existing DAOS pools and containers.

  o The API provides methods to retrieve information about detected pools and containers (e.g., names, properties).

4. Optimal Pool and Container Selection:

- Requirement ID: FR004

- Description: The Large IO API must automate the selection of optimal DAOS pools and containers for data storage operations. This selection should consider factors like performance requirements, capacity constraints, and data replication needs. To achieve optimal selection, the API relies on the ability to discover available pools and containers (FR03).

- Acceptance Criteria:

  o The API analyzes data characteristics (size, performance requirements, replication needs) and storage system resources (identified through FR03) to select suitable pools and containers for each operation.

o The selection process optimizes resource utilization based on defined criteria (performance, capacity, replication).

5. Local Metadata Management:

- Requirement ID: FR005

- Description: The Large IO API must maintain metadata associated with each data object within the local system. This metadata can include details like object names, timestamps, and potentially retrieval references. This information can be crucial for managing and tracking data objects locally.

- Acceptance Criteria:

    o The API offers methods to store and manage metadata related to data objects within the local system.

    o The stored metadata accurately reflects information about the corresponding data objects.

6. Distributed Monitoring of Resources:

- Requirement ID: FR006

- Description: The Large IO API (or a separate monitoring tool) should be able to monitor the status of targets, threads, and CPUs across distributed DAOS server nodes. This information is valuable for resource management and performance optimization. While not part of the core API functionality, monitoring capabilities can be beneficial for understanding system health.

- Acceptance Criteria:

    o The system provides mechanisms to monitor resource utilization (targets, threads, CPUs) on DAOS servers.

    o The monitoring functionality offers insights into the overall health and performance of the storage system (may require additional development).

5.5.2 Non-Functional Requirements

Efficient data management and seamless user experiences are vital for the success of any system. This document outlines key non-functional requirements essential for ensuring optimal system performance and reliability.

Performance:

- Requirement ID: NFR001
- The platform should demonstrate efficient resource utilization, optimizing CPU, memory, and storage usage to maximize performance.
- Resource consumption should be monitored and managed to prevent resource exhaustion and ensure consistent performance under varying workloads.

Reliability:

- Requirement ID: NFR002
- The system should have a high level of reliability, with a target of at least 99.99% uptime over a given period.
- It should be resilient to hardware failures, with mechanisms in place for automatic failover and recovery**.**

Scalability:

- Requirement ID: NFR003
- The system should be scalable both vertically and horizontally, allowing it to handle increasing volumes of data and user requests without degradation in performance.
- It should support dynamic scaling to accommodate fluctuations in workload demand**.**

Security:

- Requirement ID: NFR004
- Data transmission and storage should be encrypted using industry-standard encryption algorithms to ensure data confidentiality and integrity.
- Access to sensitive data should be restricted based on role-based access control (RBAC), with mechanisms for authentication and authorization**.**

Availability:

- Requirement ID: NFR005

- The system should be highly available, with redundant components and failover mechanisms to minimize downtime.
- Maintenance tasks, such as software updates and patches, should be performed with minimal disruption to service availability.

Maintainability:

- Requirement ID: NFR006
- The system should be easy to maintain, with clear documentation and modular architecture that facilitates code maintenance and updates..
- Monitoring and logging functionalities should be in place to track system performance and diagnose issues effectively.

Data Integrity :
- Requirement ID: NFR007
- The system should guarantee data accuracy and consistency throughout storage and retrieval processes. This includes adhering to ACID properties (Atomicity, Consistency, Isolation, Durability) .
- Atomicity: Each data operation is treated as an indivisible unit. Either all changes within the operation succeed, or none of them are applied. This prevents partial updates or inconsistencies in the data.
- Consistency: The data adheres to predefined rules and constraints before and after each operation. This ensures the data remains in a valid state.
- Isolation: Concurrent data access operations are isolated from each other, preventing conflicts and ensuring data integrity.
- Durability: Once a data write operation is confirmed successful, the changes are persisted to the storage medium and survive system failures.

Compatibility:
- Requirement ID: NFR008
- The system should be compatible with a range of operating systems, browsers, and devices to ensure broad accessibility for users.

- It should adhere to industry standards and protocols to enable interoperability with other systems and tools.

Usability:

- Requirement ID: NFR009

- The system should have an intuitive user interface with clear navigation and user-friendly features to enhance usability.

- It should provide adequate feedback and guidance to users to facilitate smooth interaction and minimize user errors.

By adhering to stringent performance, reliability, and security standards, the system can deliver exceptional user experiences, foster trust, and maintain operational excellence in dynamic environments.

5.5.3 Use case description

A use case description captures how a user interacts with a system to achieve a specific goal. It details the steps involved, including user actions, system responses, and potential variations like error handling.

Tables 5.1 to 5.5 detail the interactions between a user (or a Python program acting on the user's behalf) and the DAOS storage system. Each table focuses on a specific action within the system.

Table 5.1: Handling Large Data through Chunking

Use Case 1	Content
Actor	DAOS Client, DAOS Server
Description	Perform read/write operations on large datasets
Stimulus	User initiates a read or write operation specifying a data file exceeding the predefined size threshold, when the data exceeds the threshold, it chunks the data.
Response	- The API successfully reads/writes the entire data file. - Performance metrics indicate efficient handling of large data transfers.
Comments	- The specific size threshold is configurable based on system resources. - This use case validates the API's ability to manage large datasets effectively.

Table 5.2: Handle Storing of Objects

Use Case 2	Content
Actor	DAOS Client, DAOS Server
Description	Store or retrieve a user-defined data object
Stimulus	User attempts to store a data object using the API
Response	- The API serializes the data object into a format suitable for storage within the DAOS system. - The API stores the serialized object in the designated pool and container.
Comments	- This use case demonstrates the API's ability to handle various data formats through serialization. - The API should support user-defined data structures for serialization.

Table 5.3: Monitor the server to get list of all pools and containers

Use Case 3	Content
Actor	DAOS Client, DAOS Server
Description	Identify available storage resources within the DAOS system
Stimulus	The API initiates the discovery process upon user request or during initialization.
Response	- The API successfully discovers existing DAOS pools and containers on the storage system. - The API retrieves information about the discovered pools and containers (names, properties).
Comments	- This use case is crucial for the API to understand the available storage landscape within DAOS. - The retrieved information is used for subsequent functionalities like optimal pool/container selection (FR004).

Table 5.4: Enquire pool to select optimal pool and container

Use Case 4	Content
Actor	DAOS Server, DAOS Client
Description	Select suitable storage locations for data objects
Stimulus	The API receives a request to store or retrieve a data object along with its characteristics (size, performance requirements).
Response	- The API analyzes data characteristics and discovered storage resources (from FR003). - The API selects the optimal pool and container based on defined criteria (performance, capacity, replication).
Comments	- This use case demonstrates the automated selection of storage resources for optimal data placement and utilization. - The API relies on FR003 to identify available pools and containers.

Table 5.5: Maintain Metadata for each key

Use Case 5	Content
Actor	DAOS Server, DAOS Client
Description	Manage metadata associated with data objects for each key
Stimulus	User interacts with the API to store or retrieve a data object.
Response	- The API stores or retrieves the associated metadata alongside the data object within the local system. - The metadata can include details like object names, timestamps, and potentially retrieval references, about which container and pool the key has been stored in.
Comments	- This use case ensures proper tracking and management of data objects locally. - The stored metadata accurately reflects information about the corresponding data objects.

Table 5.6: Monitor all server nodes

Use Case 6	Content
Actor	DAOS Server, DAOS Client
Description	Continuously monitor availability and resource utilization (CPU, threads) of DAOS servers across the storage system.
Stimulus	Periodic monitoring or event-triggered.
Response	- Check server status for DAOS service activity. - Retrieve configuration & extract engine resources (CPU, threads) if server active. - Store info for active servers (name, resources).
Comments	Provides insights into compute capacity for optimized data placement and resource management.

5.5.4 Use Case Diagram

Use case diagram as shown in fig 5.1 illustrates the interaction between a user and a DAOS system. It depicts how the user leverages a Python program to manage storage within DAOS.

The diagram focuses on the functionalities delivered by the system from the user's perspective, promoting a user-centric understanding. It emphasizes the key actions and interactions involved, providing a high-level overview of the system's capabilities.
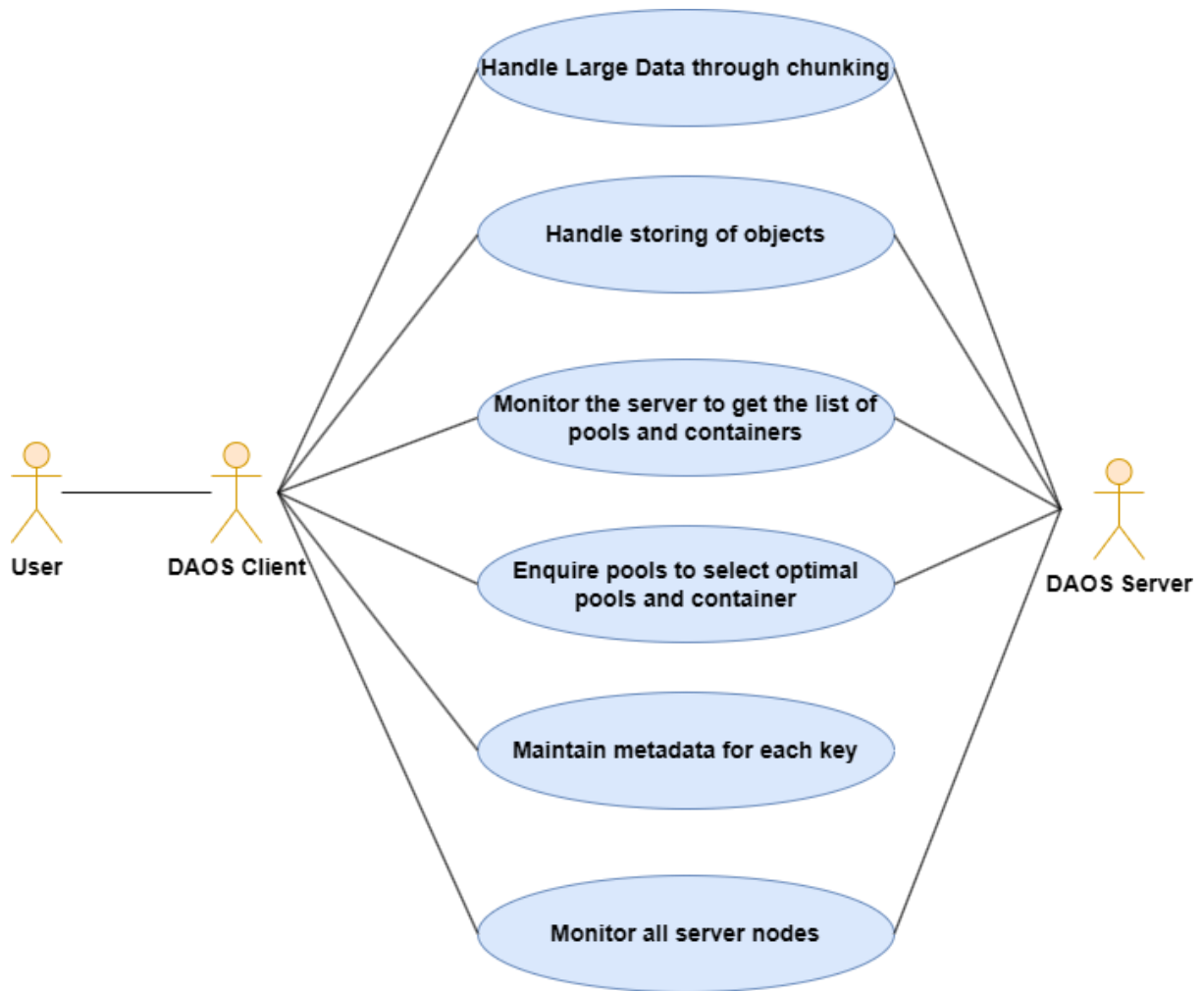
Fig. 5.1: Use Case Diagram

6. DESIGN

6.1 Introduction

The ever-growing demands of high-performance computing (HPC) necessitate innovative approaches to data storage and retrieval. When dealing with large data objects, traditional methods can become bottlenecks, hindering computational efficiency. This document proposes an elegant solution: an abstracted Large IO API for the pyDAOS framework. This API empowers users with a powerful tool to tackle large data transfers seamlessly, leveraging the strengths of pyDAOS, NVMe/SCM technologies, and intelligent data placement strategies.

HPC environments often grapple with the challenge of managing and manipulating massive datasets. Traditional methods of handling large data transfers can lead to inefficiency, primarily due to:

- Limited IOPS (Input/Output Operations Per Second): Conventional storage solutions may struggle with the high volume of IOs associated with large data transfers, impacting overall performance.
- Suboptimal Resource Utilization: Transferring large data objects as a single unit can strain hardware resources, potentially leading to bottlenecks.

Proposed Solution:

This Large IO API addresses these challenges head-on by introducing the concept of chunking. The API intercepts write requests targeting large data objects (> 1MB). It then intelligently splits the data into smaller, manageable chunks, optimizing the transfer process. But the benefits go beyond mere chunking:

- Seamless Integration: The API seamlessly integrates with the existing KV store within pyDAOS. This ensures compatibility with existing workflows and leverages the strengths of DAOS for distributed storage.
- NVMe/SCM Optimization**:** The design is specifically tailored to benefit from the high-performance capabilities of NVMe SSDs and Storage Class Memory (SCM).

These technologies excel at handling large volumes of small IOs, making them ideal for chunked data transfers.

- High-Performance Computing: By reducing the burden on storage systems and optimizing resource utilization, the Large IO API paves the way for faster data access and manipulation, critical for accelerating HPC workloads.
- Intelligent Data Placement: To further optimize performance and scalability, the API analyzes available storage pools and selects the one with the most targets (storage devices) for writing chunked data. This distributes the write load across multiple devices, enhancing overall throughput.

A crucial aspect of the Large IO API lies in its metadata management strategy. For each large data object:

- Chunking Metadata: The API stores information about the chunking process, including the original data size, number of chunks, and individual chunk sizes. This metadata facilitates efficient data reassembly during read operations.
- Pool Mapping: The API stores a mapping that identifies the specific storage pool where each chunked data object resides. This information is vital for efficiently locating and retrieving data across potentially distributed storage.

By storing this comprehensive metadata, the API unlocks significant advantages:

- Efficient Data Retrieval: During read operations, the API can leverage the stored metadata to retrieve the correct chunks from their designated storage pool. This targeted approach streamlines data retrieval and minimizes unnecessary IOs.
- Scalability and Flexibility: The pool mapping metadata empowers the system to handle potential changes in storage pool configurations. If a pool experiences performance degradation or becomes unavailable, the API can locate and retrieve data from alternative pools with suitable capacity.

6.2 Architectural Design

This Object-oriented programming (OOP) is a programming paradigm that revolves around objects. An object encapsulates data (attributes) and the procedures that operate

on that data (methods). OOP promotes code modularity, reusability, and maintainability through key principles:

- Encapsulation: Data is bundled together with the methods that operate on it, promoting data protection and controlled access.
- Inheritance: New classes can inherit properties and functionalities from existing classes, facilitating code reuse and extension.
- Polymorphism**:** Different classes can implement the same method with varying behavior, providing flexibility for handling diverse situations.

Benefits of OOP for the Large IO API:

1. Encapsulation:
   - The Large IO API class encapsulates the logic for chunking, storage management, and data retrieval. This promotes code clarity and organization. For example, the _chunk_data(data: bytes) method hides the internal logic of splitting data into chunks, making the write(data: bytes, path: str) method cleaner to use.

2. Reusability:
   - The use of classes like KVStore (abstract) and potentially concrete implementations for specific storage backends (e.g., pyDAOS KVStore) promotes reusability. The core functionalities of the API can remain the same, while the underlying data persistence mechanism can be adapted through different KVStore implementations.

3. Maintainability:
   - By separating functionalities into well-defined classes, the code becomes easier to understand, modify, and debug. For instance, if the chunking logic needs modification, changes can be isolated within the _chunk_data(data: bytes) method without affecting other parts of the API.

4. Inheritance (Potential):
   - The design allows for future extension through inheritance. Subclasses could potentially inherit functionalities from the LargeIOAPI class and introduce specialized behaviors for handling different data types or storage systems.

AdminManager

-servers: List<Server>

-configurations: List<Configuration>

+discoverServers(): List[Server]

+getServerInfo(server: Server): ServerInfo

ServerMonitor

-serverStatus: ServerStatus

+getPools(): List[Pool]

+getContainers(pool: Pool): List[Container]

+monitorServers(): ServerStatus

PoolContainerSelector

-poolMetrics: Dict<Pool, Metrics(targets, empty space) >

-containerMetrics: Dict<Container, Metrics(targets, empty space)>

+selectOptimalPool(): Pool

+selectOptimalContainer(pool: Pool): Container

DataChunkHandler

-chunkSize: int

-dataChunks: List<DataChunk>

+splitData(data: byte[]): List<DataChunk>

+mergeChunks(chunks: List<DataChunk>): byte[]

+mergeChunks(chunks: List<DataChunk>): byte[]

+insertChunk(key: Key, chunk: DataChunk)

+retrieveChunk(key: Key): DataChunk

+deleteChunk(key: Key)

+listChunks(): List[Key]

+bulkInsertChunks(chunks: Dict[Key, DataChunk])

+bulkRetrieveChunks(keys: List[Key]): Dict[Key, DataChunk]

ObjectStorageHandler

-storage: Dict<Key, bytes>

+storeObject(key: Key, obj: list/tuple/dictionary/set)

+retrieveObject(key: Key): list/tuple/dictionary/set

+deleteObject(key: Key)

+serialize(obj: list/tuple/dictionary/set): bytes

+deserialize(data: bytes): list/tuple/dictionary/set

MetadataManager

-metadata: Dict<Key, Metadata(pool, container,chunk_size)>

+storeMetadata(key: Key, metadata: Metadata)

+getMetadata(key: Key): Metadata

+updateMetadata(key: Key, metadata: Metadata)

Fig 6.1: Class Diagram

This class diagram as show in fig 6.1 illustrates a system for managing data storage and server resources. The diagram consists of six primary classes: DataChunkHandler, MetadataManager, ObjectStorageHandler, PoolContainerSelector, ServMonitor, and AdminManager. Below is a detailed description of each class and their interactions:

1. **DataChunkHandler**:
   - **Attributes**:
     - chunkSize: int
     - dataChunks: List<DataChunk>
   - **Methods**:
     - splitData(data: bytes) -> List<DataChunk>: Splits data into smaller chunks.
     - mergeChunks(chunks: List<DataChunk>) -> bytes: Merges data chunks back into the original data.
     - addChunk(chunk: DataChunk): Adds a new data chunk.
     - removeChunk(chunk: DataChunk): Removes a specified data chunk.
     - retrieveChunk(key: Key) -> DataChunk: Retrieves a data chunk by key.
     - updateChunk(key: Key, chunk: DataChunk): Updates a specific data chunk by key.
     - listAllChunks() -> List<DataChunk>: Lists all data chunks.

2. **MetadataManager**:
   - **Attributes**:
     - metadataPool: Dictionary<Key, Metadata>
   - **Methods**:
     - getMetadata(key: Key) -> Metadata: Retrieves metadata by key.
     - setMetadata(key: Key, metadata: Metadata): Sets metadata for a specified key.
     - deleteMetadata(key: Key): Deletes metadata by key.
     - updateMetadata(key: Key, metadata: Metadata): Updates metadata for a specified key.
     - mergeMetadata(key: Key, metadata: Metadata): Merges metadata for a specified key.

3. **ObjectStorageHandler**:
   - **Attributes**:
     - storage: Dictionary<Key, Object>
   - **Methods**:
     - storeObject(key: Key, obj: List[Tuple[Dictionary, Set]]): Stores an object in the storage.
     - retrieveObject(key: Key) -> List[Tuple[Dictionary, Set]]: Retrieves an object by key.
     - deleteObject(key: Key): Deletes an object by key.
     - serialize(obj: List[Tuple[Dictionary, Set]]) -> bytes: Serializes the object.
     - deserialize(data: bytes) -> List[Tuple[Dictionary, Set]]: Deserializes the data into an object.

4. **PoolContainerSelector**:
   - **Attributes**:
     - poolMetrics: Dictionary<Pool, Metrics(empty_space)>
     - containerMetrics: Dictionary<Container, Metrics(empty_space)>
   - **Methods**:
     - selectOptimalPool(pools: List[Pool], obj: Object) -> Pool: Selects the optimal pool for storing an object.
     - selectOptimalContainer(pool: Pool, obj: Object) -> Container: Selects the optimal container within a pool.
     - getPoolMetrics(pool: Pool) -> Metrics: Retrieves metrics for a specified pool.
     - getContainerMetrics(container: Container) -> Metrics: Retrieves metrics for a specified container.

5. **ServMonitor**:
   - **Attributes**:
     - serverStatus: List[ServerStatus]
   - **Methods**:
     - monitorServers() -> List[ServerStatus]: Monitors and returns the status of all servers.

6. **AdminManager**:
   - **Attributes**:

- servers: List[Server]
- configurations: List[Configuration]

- o **Methods**:
  - getServers() -> List[Server]: Retrieves a list of all servers.
  - addServer(server: Server): Adds a new server.
  - removeServer(server: Server): Removes a specified server.
  - updateServer(server: Server, config: Configuration): Updates the configuration of a specified server.
  - getServerInfo(server: Server) -> ServerInfo: Retrieves information about a specified server.

- **DataChunkHandler** interacts with **MetadataManager** to manage metadata for data chunks and with **ObjectStorageHandler** to handle storage and retrieval of objects split into data chunks.

- **ObjectStorageHandler** relies on **PoolContainerSelector** to choose the optimal pool and container for storage.

- **ServMonitor** provides server status information to **AdminManager**, which manages the servers and their configurations.

6.3   User Interface Design

The Large IO API is a Python script designed to facilitate data transfer between local file systems and DAOS, a high-performance distributed storage system. This user manual details the command-line interface (CLI) for interacting with the Large IO API. The CLI provides a user-friendly way to manage large data objects within a DAOS environment.

Prerequisites

- A CentOS virtual machine set up in VirtualBox with DAOS successfully installed is assumed. Basic familiarity with creating and managing DAOS pools and containers within the virtual machine environment is required. Instructions for setting up DAOS can be found in the official DAOS documentation.

- Python 3 and the pyDAOS library must be installed on the virtual machine. The pyDAOS library provides Python bindings for interacting with the DAOS system.
- Basic knowledge of using the command line (terminal) is necessary for interaction with the Large IO API script.

User Interface

A text-based CLI is utilized by the Large IO API for user interaction. Specific commands and arguments are used to execute the script and perform various data transfer operations.

Script Name

The script is assumed to be named large_io_api.py. This name can be replaced with the actual script name used in your implementation.

Available Commands

The following commands are available for data transfer and management tasks:

- write (data_path, pool_name, container_name): Writes a local file to the DAOS system.
    - data_path: Path to the local file on the system.
    - pool_name: Name of the DAOS pool to store the data.
    - container_name: Name of the DAOS container within the pool to store the data.
- read (container_name, data_path): Reads data from a DAOS container to a local file.
    - container_name: Name of the DAOS container containing the data.
    - data_path: Path where the retrieved data will be written as a local file.
- list_pools: Lists all available DAOS pools on the system.
- list_containers (pool_name): Lists all containers within a specified DAOS pool.
- help: Displays help information for all available commands and their options.

6.4   Low Level Design

Sequence Diagram:

The sequence diagram as depicted in fig 6.2 for managing DAOS servers begins with the client initiating the DAOS server start process, which involves launching DAOS servers on multiple nodes. The SSH Manager then connects to each node via SSH to interact with the DAOS servers, checking their status. If a server is running, the SSH Manager opens the configuration file to extract details such as the number of engines, and for each engine, the number of targets, CPUs, and threads. This extracted information is stored in a metadata file by the ServerConfiguration Manager, which then uses this information to select an appropriate server based on the specified criteria.

Following the server selection, the client creates pools and containers on the chosen server. The DAOS server then provides a list of available tools, and the ServerConfiguration Manager selects the optimal pool based on the number of targets and available free space. It also selects a container using a round-robin approach. Data is then inserted into the selected pool and container by the client. Throughout this process, metadata for each data insertion is stored in a file to maintain an accurate record of the operations and configurations used.
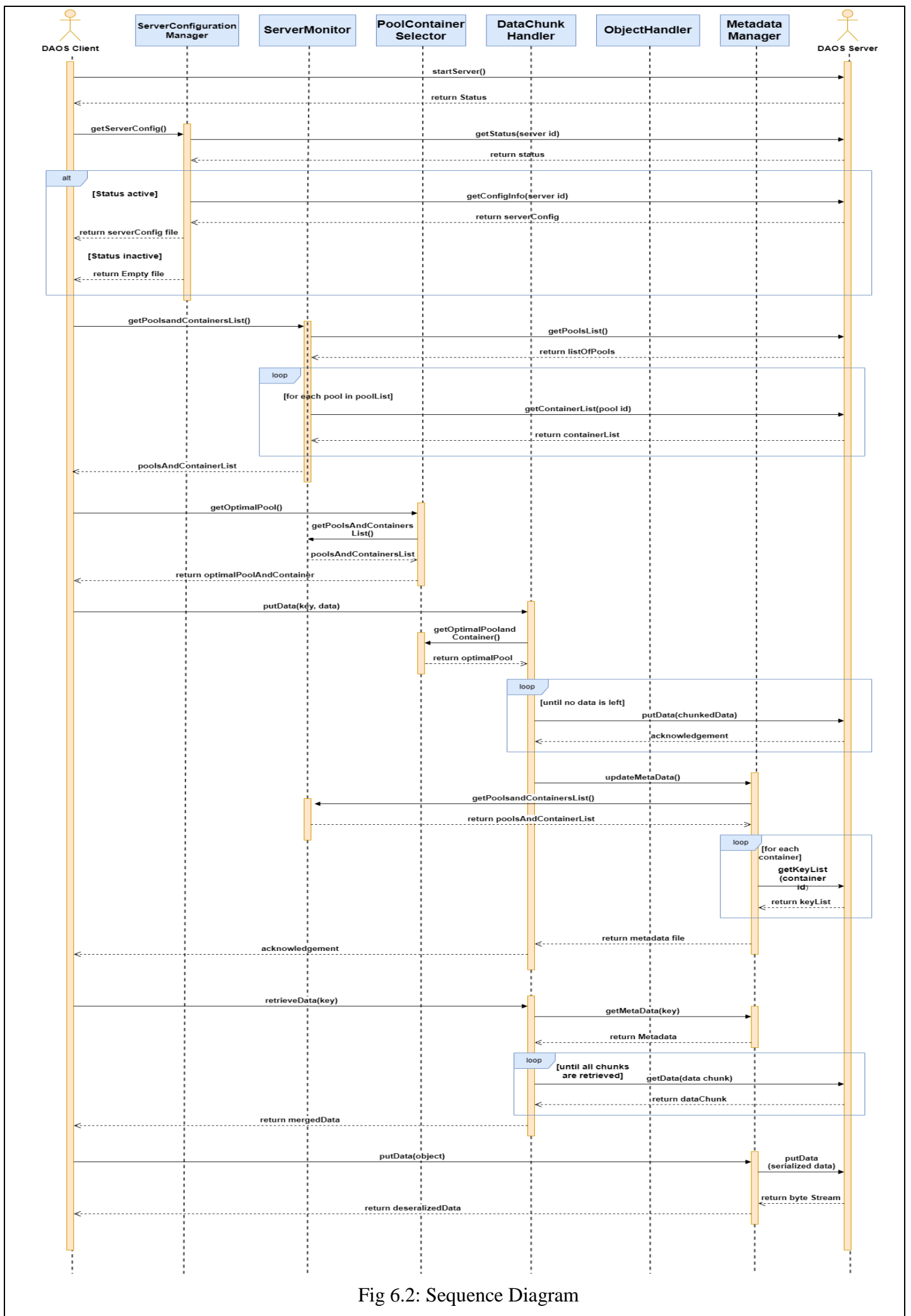
Fig 6.2: Sequence Diagram

State Diagrams:

State diagrams are a type of UML diagram that represent the various states an object can be in, as well as the transitions between those states. They are used to model the dynamic behavior of a system, helping to visualize how an object responds to different events over time. Fig 6.3 to 6.8 depicts state diagram for various objects in the system and tables 6.1 to 6.12 describes state and stimuli in the state diagram

Fig 6.3: State Diagram for DataChunkHandler
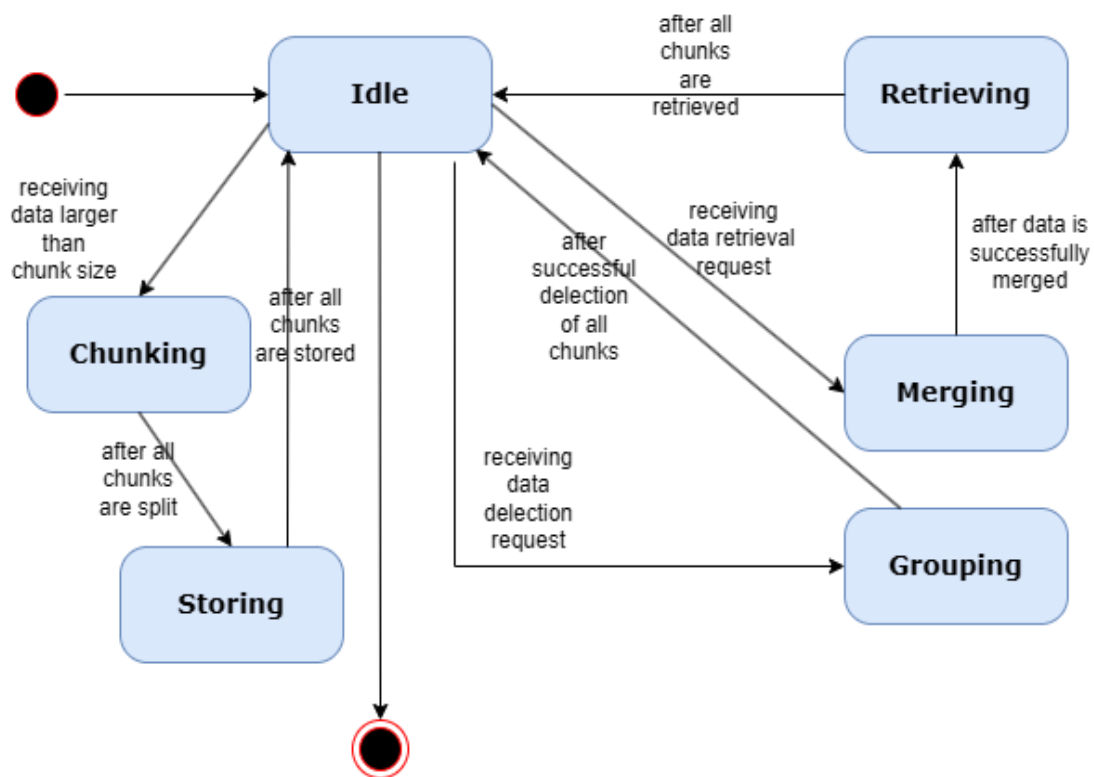
Table 6.1: Description of the States in the State Diagram for DataChunkHandler

State	Description
Idle	The initial state of the system.
Chunking	The state where data is split into chunks.
Storing	The state where data chunks are stored.
Retrieving	The state where data is being retrieved.
Merging	The state where retrieved data chunks are merged.
Grouping	The state where data is grouped.

Table 6.2: Description of the Stimuli in the State Diagram for DataChunkHandler

Stimulus	Description
request to create a container	Initiates the process of creating a container.
after all chunks are split	Triggers the transition from the chunking state to receiving data state.
receiving data request	Initiates receiving data chunks.
after successful deletion of all chunks	Triggers the transition from the retrieving state to the merging state.

Fig 6.4: State Diagram for ObjectHandler
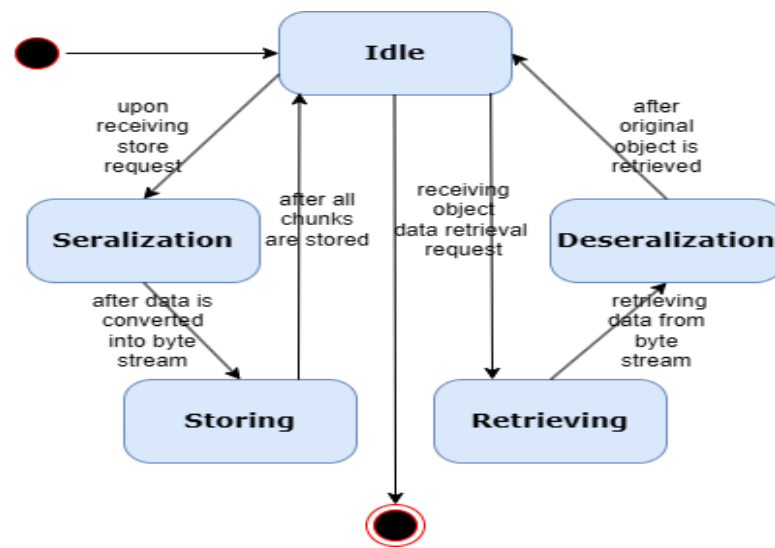
Table 6.3: Description of the States in the State Diagram for ObjectHandler

State	Description
Idle	The system is idle, waiting for a data retrieval request.
Retrieving	The system is in the process of retrieving data from a byte stream.
Serialization	The system is converting the data from its original format into a byte stream.
Deserialization	The system is converting the data from a byte stream into its original format.
Storing	The system is storing the data after conversion into a byte stream.

Table 6.4: Description of Stimulus applied to the States in the State Diagram for ObjectHandler

Stimulus	Description
Data retrieval request	The system receives a request to retrieve data.
After data is converted into byte stream	The data has been converted from its original format into a byte stream.
After receiving original object	The system has received the original data object to be stored.
After all chunks are stored	All chunks of data have been stored.
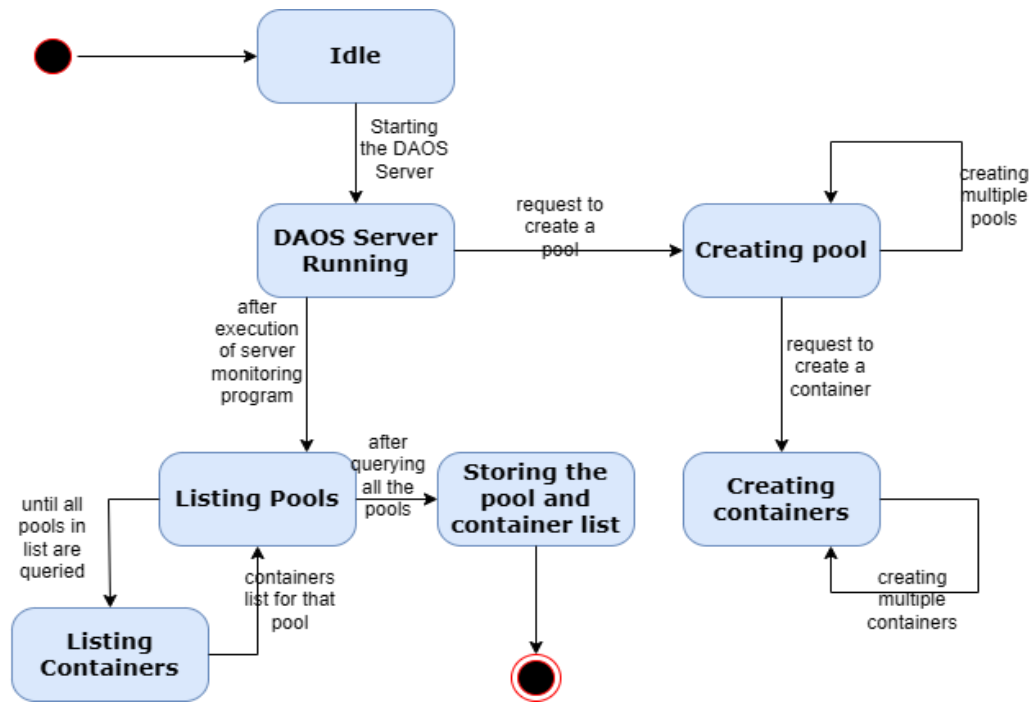
Fig 6.5: State Diagram for ServerMonitor

Table 6.5: Description of the States in the State Diagram for ServerMonitor

State	Description
Idle	The system's initial state, waiting for user input.
DAOS Server Running	The state where DAOS server is up and running
Creating Pool	The state where a single pool is being created
Creating containers	The state where a single container is being created
Listing Pools	The process of retrieving a list of available pools.
Pool Selection	The stage where the optimal pool for the operation is chosen.
Listing Containers	Each pool is queried to get containers list
Storing the pool and container list	State where obtained pool and container list is stored in a file

Table 6.6: Description of Stimulus applied to the States in the State Diagram for DataChunkHandler

Stimulus	Description
request to create a container	Initiates the container creation process.
after execution of server monitor program	Triggers the system to query for pools and containers
request to create a pool	Initiates the process of creating a pool.
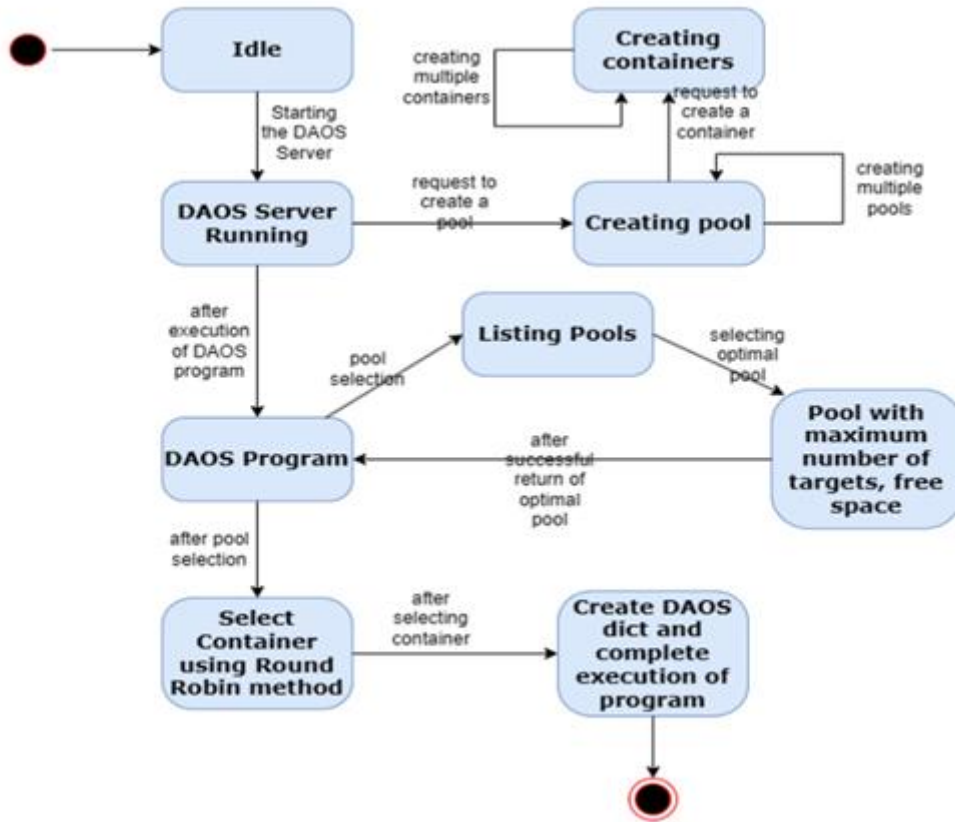
Fig 6.6: State Diagram for PoolContainerSelector

Table 6.7: Description of the States in the State Diagram for PoolContainerSelector

State	Description
Idle	The initial state of the system.
Starting the DAOS Server	The state where the DAOS server is being started.
Creating	The state where a single container is being created.
Creating multiple containers	The state where multiple containers are being created.
Creating pool	The state where a pool is being created.
Running	The state where the DAOS Server is running.
Listing Pools	The state where the list of pools are being fetched.
Pool Selection	The state where the optimal pool is being selected.
Selecting container using Round Robin method	The state where the container is selected using the Round Robin method.
Create DAOS dict and complete execution of program	The state where the DAOS dictionary is created and the program execution is completed.

Table 6.8: Description of Stimulus applied to the States in the State Diagram for PoolContainerSelector

State	Description
Idle	The initial state of the system, waiting for user input.
Creating pool	The state where a pool is being established.
Creating multiple containers	The state where multiple containers are being created simultaneously.
Running DAOS Server	The state where the DAOS Server is up and running.
Insert Data	The state where data is being inserted.
Retrieve Data	The state where data is being retrieved.
Delete Data	The state where data is being deleted.
Synchronize	The state where the metadata is synchronized.

Fig 6.7: State Diagram for MetaDataManager

Table 6.9: Description of the States in the State Diagram for MetaDataManager

State	Description
Idle	The initial state of the system, waiting for user input.
Creating pool	The state where a pool is being established.
Creating multiple containers	The state where multiple containers are being created simultaneously.
Running DAOS Server	The state where the DAOS Server is up and running.
Insert Data	The state where data is being inserted.
Retrieve Data	The state where data is being retrieved.
Delete Data	The state where data is being deleted.
Synchronize	The state where the metadata is synchronized.

Table 6.10: Description of Stimulus applied to the States in the State Diagram for MetaDataManager

Stimulus	Description
request to create a pool	Initiates the process of creating a pool.
request to create containers	Triggers the creation of multiple containers within an existing pool.
request to insert data	Initiates the data insertion process within the running DAOS Server.
request to fetch data	Triggers the retrieval of data from the DAOS Server.
request to delete data	Initiates the data deletion process within the DAOS Server.
after all pools and containers are parsed	Signals the completion of metadata parsing and triggers the synchronization process.
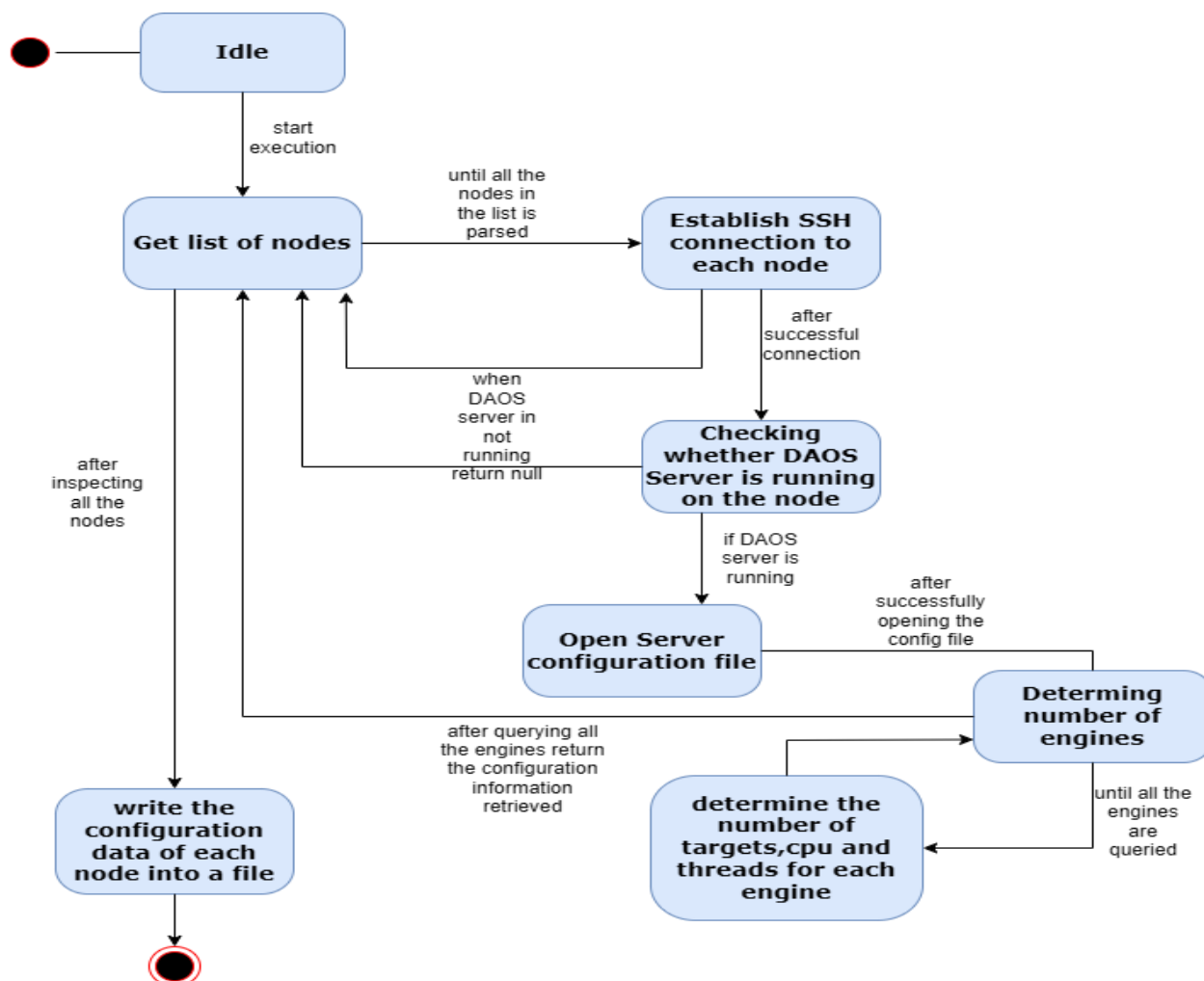
Fig 6.8: State Diagram for ServerConfigurationManager

Table 6.11: Description of the States in the State Diagram for ServerConfigurationManager

State	Description
Idle	The system is in an idle state, waiting for a start signal.
Start	The system has received a start signal and is beginning the process of retrieving information from the DAOS servers.
Get list of nodes	The system is retrieving a list of DAOS servers to be queried.
Checking whether DAOS Server is running on the node	The system is checking if a DAOS server is running on the specified node.
After successful connection	The system has successfully connected to a DAOS server.

Open Server configuration file	The system is opening the configuration file for the connected DAOS server.
Determining number of engines	The system is determining the number of engines from the DAOS server configuration file.
After querying all the engines return the configuration information retrieved	The system has retrieved the configuration information from all the engines and is ready to return it.
Write the configuration data of each node into a file	The system is writing the retrieved configuration data for each DAOS server to a file.

Table 6.12: Description of Stimulus applied to the States in the State Diagram for ServerConfigurationManager

Stimulus	Description
Start	The system receives a signal to start the process of retrieving information from the DAOS servers.
Until all the nodes in the list is parsed	This loop continues until the system has retrieved information from all the DAOS servers in the list.
When DAOS server is not running	The system encounters a DAOS server that is not running.
After successful connection	The system has successfully connected to a DAOS server.
After successfully opening the configuration file	The system has successfully opened the configuration file for the connected DAOS server.
Until all the engines are queried	This loop continues until the system has retrieved information from all the engines on the DAOS server.

6.5 Conclusion

The design section meticulously details the blueprint for the Large IO API project, emphasizing an object-oriented approach. The architectural design leverages the power of object-oriented programming, as visualized by a class diagram. This diagram depicts the system's classes, their attributes, and the methods they possess. This object-oriented foundation fosters modularity, code reusability, and maintainability, promoting a well-structured system.

Complementing the architecture is a user-friendly command-line interface (CLI) detailed in the user interface design. Users can seamlessly interact with the Large IO

API and manage data transfer operations through available commands, arguments, and expected behaviors.

The low-level design delves into the intricate details using sequence diagrams. These diagrams depict the interactions between objects within the system, illustrating how objects collaborate to achieve data transfer tasks. This provides a granular view of the internal logic and control flow, serving as a valuable guide for developers during the implementation phase.

By meticulously considering these elements – the object-oriented architecture, user-friendly CLI, and granular low-level design with object interactions – the Large IO API establishes a robust and user-friendly framework for efficient data transfer between local file systems and the DAOS storage system. This user manual, complemented by the comprehensive object-oriented design documented here, empowers users to leverage the Large IO API's capabilities and effectively manage large data objects within their DAOS environment. The object-oriented approach fosters a well-structured, maintainable, and efficient system, paving the way for successful project implementation.

7. IMPLEMENTATION

7.1 Tools

The project showcases the versatility of working with the DAOS (Distributed Asynchronous Object Storage) system. This comprehensive setup leverages a range of tools and technologies, including a CentOS virtual machine, DAOS RPM installation, DAOS Python API, PYDAOS pool and container management, and the use of the Pickle library for serializing and deserializing Python objects. The project also includes functionality for measuring the timing of various DAOS operations, ensuring the efficient and effective utilization of the system, and takes advantage of high-performance SSD and NVMe storage media to optimize performance.

- **CentOS Operating System**:Running the PYDAOS-Programs project on a CentOS virtual machine provides a stable and compatible environment for working with the DAOS system.

- **Virtual Machine (VirtualBox):**Using a virtual machine, such as VirtualBox, allows you to easily set up and manage the development environment, isolating the DAOS system from your primary operating system.

- **DAOS RPM Installation**:Installing the DAOS RPM package is the recommended approach to set up the DAOS system on your CentOS virtual machine, ensuring compatibility and access to the necessary components.

- **DAOS Python API** :The DAOS Pydaos API is a crucial component that enables you to interact with the DAOS storage system programmatically, allowing for file uploads, object storage, and metadata management.

- **PYDAOS Pool and Container Management**:The PYDAOS-Programs project likely includes utilities and examples for creating, managing, and interacting with DAOS pools and containers, which are fundamental to the DAOS storage hierarchy.

- **Pickle Library**:The use of the Pickle library allows you to serialize and deserialize Python objects, enabling the storage and retrieval of custom data structures within the DAOS directory.

- **Timing Measurements:**Measuring the time taken for various DAOS operations, such as file uploads and container retrievals, helps you assess the performance and efficiency of the DAOS system in your specific setup.

- **SSD and NVMe Storage**:Utilizing high-performance storage media, such as SSDs and NVMe drives, can significantly improve the overall performance and responsiveness of the DAOS system, especially for file-intensive operations.

  This combination of tools and processes provides a solid foundation for your work with the DAOS system, allowing you to explore its capabilities, measure its performance, and develop robust applications that leverage the distributed and asynchronous nature of the DAOS storage solution.

7.2 Technologies

This project explores the intersection of Python and DAOS (Distributed Asynchronous Object Storage) to create a powerful toolkit for managing data in a high-performance and scalable manner. DAOS, known for its fault tolerance and speed, is ideal for data-intensive workloads, while Python provides a user-friendly and versatile programming language. By combining these strengths, PYDAOS-Programs empowers users to interact with DAOS storage efficiently.

1. **DAOS (Distributed Asynchronous Object Storage)**: DAOS is an open-source, software-defined, object storage system designed for high-performance, fault-tolerant, and scalable storage. It is optimized for distributed, data-intensive workloads, such as HPC, AI, and analytics.

2. **Python**: Python is the primary programming language used in this project to interact with the DAOS storage system.

3. **File Uploading and Chunking**: The programs demonstrate the ability to upload files to the DAOS storage system in smaller chunks, which can potentially improve the overall upload performance. This involves techniques like:

   - Splitting files into customizable chunk sizes

   - Uploading the chunks asynchronously

   - Reassembling the file on the server-side

4. **Serialization and Deserialization**: The programs showcase the ability to store and retrieve Python objects (such as classes and dictionaries) in the DAOS storage

system by serializing and deserializing them. This allows for the persistence and retrieval of complex data structures.

5. **Pool and Container Management**: The programs demonstrate the ability to automatically find, allocate, and manage pools and containers within the DAOS storage system. Pools are top-level storage units, while containers are analogous to directories or namespaces within the pools.

6. **File Comparison**: One program provides the functionality to compare the contents of an original file and the retrieved file from the DAOS storage system to ensure data integrity. This can be useful for verifying the correctness of the upload and download process.

7. **Metadata Management**: The programs mention the use of a metadata file to track the stored data, indicating the ability to maintain additional information about the stored objects and files.

8. **Asynchronous Programming**: Some of the programs utilize asynchronous programming techniques, such as concurrent file uploads, to optimize the performance of the storage operations.

   Performance Evaluation: The programs include features to measure and compare the upload times for different chunk sizes, which can help in determining the optimal configuration for file uploads.

7.3 Overall view of the project in terms of implementation

The programs provide a comprehensive set of tools and examples for interacting with the DAOS storage system, making it a valuable resource for DAOS developers and users. The inclusion of utility programs and a focus on reusable and efficient DAOS-related functionality suggest a well-designed and practical approach to working with the DAOS storage system.

1. **Handle Large Data through Chunking**:
Implementation:
   - This module uses the daos_python library to interact with the DAOS storage system.

- The read_and_upload_file() function is the main entry point, accepting the file path and chunk size as input.
- The function opens the file in binary mode and reads it in chunks, uploading each chunk to the DAOS storage.
- The container.write_obj() function from the DAOS API is used to upload the chunks.
- Error handling is implemented to capture and log any issues that may occur during the upload process.
- The function returns a status indicator (success/failure) and a list of any errors encountered.

```python
def upload_file():
    key = input("Enter new key: ")
    file_path = input("Enter path to file: ")

    if os.path.exists(file_path):
        chunk_dict = {}
        try:
            with open(file_path, "rb") as f:
                chunk_count = 0
                while True:
                    data = f.read(CHUNK_SIZE)
                    if not data:
                        break
                    chunk_key = f"{key}chunk{chunk_count}"
                    chunk_dict[chunk_key] = data
                    chunk_count += 1
            # Measure time only for the bput operation
            bput_start_time = time.time()
            daos_dict.bput(chunk_dict)
            bput_end_time = time.time()
            upload_time = bput_end_time - bput_start_time

            print(f"File uploaded in {chunk_count} chunks successfully. Time taken: {upload_time} seconds")
```

```
        except Exception as e:
            print(f"Error uploading file: {e}")
    else:
        print("File not found.")
```

2. Handle Storing of Objects:

Implementation:

- This module uses the pickle module for efficient serialization and deserialization of Python objects.
- The store_object() function is the main entry point, accepting the Python object as input.
- The function serializes the object using pickle.dumps() and determines the optimal storage location using the Pool and Container Management module.
- The serialized object is then uploaded to the DAOS storage using the File Uploading and Chunking module.
- A unique object ID is generated for the stored object, and the object's metadata is recorded in the local metadata store.
- The function returns the successful storage status and the object's unique ID.
- The local metadata store can be implemented using a simple key-value store, such as a Python dictionary or a dedicated metadata management system.

```
my_object = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

# Serialize the object using pickle
serialized_object = pickle.dumps(my_object)
# Store the serialized object in the DAOS container
try:
    # If the object already exists, delete it first
    daos_dict.pop(key)
except DObjNotFound:
    pass
```

```
# Store the serialized object
daos_dict.put(key, serialized_object)
print("Python object saved successfully.")


key = "my_python_object"
# Retrieve the serialized object from the DAOS container
serialized_object = daos_dict.get(key)
# Deserialize the object using pickle
my_object = pickle.loads(serialized_object)
# Now you can use the Python object as needed
print("Name:", my_object["name"])
print("Age:", my_object["age"])
print("City:", my_object["city"])
```

3. Monitor the Server to Get List of Pools and Containers:

Implementation:

- This module uses the daos_python library to interact with the DAOS system.
- The get_optimal_pool_and_container() function is the main entry point, which orchestrates the pool and container management process.
- The function first connects to the DAOS system using the daos.connect() function.
- It then retrieves the list of available pools using the daos.pool.list() function and queries the properties of each pool using the daos.pool.query() function.
- The function identifies the pool with the maximum number of targets and creates or opens a container within that pool.
- Finally, the function synchronizes the local metadata file to ensure that the DAOS storage information is properly recorded.

```
# Listing all pools
    output_bytes = subprocess.check_output(["dmg", "pool", "list"])
    output_string = output_bytes.decode("utf-8")


# Run the command to get the number of targets in the pool
```

```
        query_output_bytes = subprocess.check_output(["dmg", "pool", "query",
pool_name])
        query_output_string = query_output_bytes.decode("utf-8")
        # Extract free space and unit from query output using regex
        free_space_match        =        re.search(r'Free:\s*([\d..]+)\s*(GB|MB)?',
query_output_string)


    # Run the command to list containers in the pool with the maximum number of
targets
        output_bytes = subprocess.check_output(["daos", "cont", "list", pool_name])
        output_string = output_bytes.decode("utf-8")
        lines = output_string.strip().split("\n")[2:]
```

4. Select the Optimistic Pool and Container

Implementation:

- This functionality is implemented within the Automated Chunking module, which combines the functionality of the File Uploading and Chunking and Pool and Container Management modules.
- The upload_file() function in the Automated Chunking module is responsible for selecting the optimal pool and container.
- It uses the get_optimal_pool_and_container() function from the Pool and Container Management module to retrieve the list of available pools and their properties.
- The function then selects the pool with the maximum number of targets and allocates a new container or uses the next available container in a round-robin fashion.
- The selected pool and container information is then used by the File Uploading and Chunking module to upload the file.

```
# Get a list of pools
    output_bytes = subprocess.check_output(["dmg", "pool", "list"])
    output_string = output_bytes.decode("utf-8")
    lines = output_string.strip().split("\n")[2:]


# Iterate over each line and extract container name
```

```
    for line in lines:

       container_name = line.split()[1]

       containers_in_pool.append(container_name)

    # Return the list of container names in the pool with the maximum number of
targets

    return pool_name, containers_in_pool
```

5. Maintain Metadata of Each Key in the Local System:

Implementation:

- The Automated Chunking module and the Object Serialization and Storage module are responsible for maintaining the local metadata.
- The Automated Chunking module updates the local metadata after a successful file upload, recording the file's storage location and other relevant information.
- The Object Serialization and Storage module maintains the metadata of stored objects, including their unique identifiers, storage locations, and serialized sizes.
- The local metadata store can be implemented using a simple key-value store, such as a Python dictionary or a dedicated metadata management system.

```
# Append key, pool, container, and chunk size information to the list

    for key_prefix, chunk_size_mb in unique_keys:

       keys_info.append({

          "key": key_prefix,

          "chunk_size": chunk_size_mb,

          "pool": pool_name,

          "container": container_name

       })
```

6.Simplify System Administration with Automatic Server Discovery:

Implementation:

- This functionality can be implemented in a separate module, such as DAOS Server Discovery.
- The DAOS Server Discovery module would be responsible for the following:
  - Providing a method to scan the network or a predefined set of servers to discover available DAOS servers.

- Maintaining a cache of the discovered DAOS server information, which can be accessed by other modules.
- Implementing a configuration mechanism for users to specify the DAOS server details manually.
- Integrating with a centralized server discovery or service registry solution (if available) to automatically obtain the DAOS server information.

```
for server, credentials in servers_credentials.items():
    try:
        ip_address = credentials['ip_address']
        username = credentials['username']
        password = credentials['password']

        # Construct SSH command with password and sudo
        ssh_command = f'sshpass -p "{password}" ssh -o
StrictHostKeyChecking=no {username}@{ip_address} "echo {password} | sudo -
S systemctl status daos_server"'

        result = subprocess.run(ssh_command, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.PIPE, universal_newlines=True)

        # Check if "Active: active (running)" is in the output
        if "Active: active (running)" in result.stdout:

            # Read daos_server.yml
            ssh_command = f'sshpass -p "{password}" ssh -o
StrictHostKeyChecking=no {username}@{ip_address} "cat
/etc/daos/daos_server.yml"'
            output = subprocess.check_output(ssh_command, shell=True)
            yaml_data_remote = yaml.safe_load(output.decode('utf-8'))
            engine_info_remote = extract_engine_info(yaml_data_remote)
            active_servers[server] = engine_info_remote
    except subprocess.CalledProcessError as e:
```

```
            print(f"Error    accessing    server    {server}    ({ip_address}):
{e.stderr.decode('utf-8')}")
```

Each module is designed to be self-contained and reusable, allowing users to incorporate the desired functionalities into their own DAOS-based projects. The modular structure also facilitates future expansion and enhancement of the repository, as new features and utilities can be added without disrupting the existing codebase.

7.4   Explanation of Algorithm and how it is been implemented

Algorithm:

This project Implements Python programs that explore various aspects of working with the DAOS (Distributed Asynchronous Object Storage) storage system. DAOS is a high-performance, scalable, and fault-tolerant distributed storage solution designed for modern data-centric workloads and architectures. The programs to demonstrate how to interact with DAOS, including file uploading, object serialization, pool and container management.

1. **Handle Large Data through Chunking**:

Algorithm:

1. Accept the file path to be uploaded and the desired chunk size (or use a default value).
2. Open the file in binary mode.
3. Initialize a progress tracker variable to keep track of the upload progress.
4. While there is data left to be read from the file:
    - Read a chunk of data from the file using the specified chunk size.
    - Upload the chunk to the DAOS storage using the DAOS Python API:
        - Create a unique object ID for the chunk.
        - Use the DAOS container.write_obj() function to upload the chunk.
        - Handle any errors that occur during the upload and log them.
    - Update the progress tracker variable.
5. Close the file.
6. Return the successful upload status and any error information.

2. Handle Storing of Objects:

Algorithm:

1. Accept the Python object to be stored.

2. Serialize the object using the pickle module.

3. Determine the storage location (pool and container) for the serialized object:

    - Use the Pool and Container Management module to select the optimal pool and container.

4. Upload the serialized object to the DAOS storage using the File Uploading and Chunking module.

5. Generate a unique object ID for the stored object.

6. Store the object's metadata (e.g., object ID, storage location, serialized size) in the local metadata store.

7. Return the successful storage status and the object's unique ID.

3. Monitor the Server to Get List of Pools and Containers:

Algorithm:

1. Use the DAOS Python API to retrieve the list of available DAOS pools:

    - Call the daos.connect() function to establish a connection to the DAOS system.

    - Use the daos.pool.list() function to retrieve the list of available pools.

2. For each pool, retrieve the pool's properties (e.g., number of targets, utilization):

    - Call the daos.pool.query() function to get the pool's properties.

3. Identify the pool with the maximum number of targets, as this is likely the most optimal pool for storage.

4. Within the selected pool, create a new container or use an existing one for the data storage:

    - Use the daos.container.create() function to create a new container.

    - Alternatively, use the daos.container.open() function to open an existing container.

5. Synchronize the local metadata file to ensure consistency between the DAOS storage and the local file system:

- Write the pool and container information to the local metadata file.

4. **Select the Optimistic Pool and Container**:

Algorithm:

1. Use the Pool and Container Management module to retrieve the list of available DAOS pools and their properties.
2. Select the pool with the maximum number of targets as the optimal pool for storage.
3. Within the selected pool, allocate a new container or use an existing one in a round-robin fashion:
   - Maintain a pointer to the last used container.
   - When a new container is needed, either create a new one or use the next available container in the pool.

5. **Maintain Metadata of Each Key in the Local System**:

Algorithm:

1. Create a local metadata store to keep track of the stored objects and their storage details.
2. After a successful file upload or object storage:
   - Generate a unique identifier for the stored data.
   - Retrieve the storage location (pool and container) information.
   - Store the object's metadata (unique identifier, storage location, serialized size, etc.) in the local metadata store.

6. **Simplify System Administration with Automatic Server Discovery**:

Algorithm:

1. Periodically scan the network or a predefined set of servers to discover available DAOS servers.
2. Maintain a cache of the discovered DAOS server information, including their IP addresses, port numbers, and any other relevant details.
3. Provide a configuration mechanism for users to specify the DAOS server details, which can be used by other modules to interact with the DAOS system.
4. Integrate with a centralized server discovery or service registry solution (if available) to automatically obtain the DAOS server information.

Other modules, such as the Automated Chunking and Object Serialization and Storage modules, can use the DAOS Server Discovery module to obtain the necessary DAOS server information, simplifying the overall system administration.

7.5   Conclusion

The PYDAOS project emerges as a powerful bridge between the user-friendly world of Python and the high-performance realm of DAOS storage. This project empowers developers by offering a comprehensive toolkit for interacting with DAOS using Python. Key functionalities include efficient file uploading with chunking for large datasets, object storage with serialization for complex data structures, and pool and container management for organized data placement. Additionally, data integrity verification ensures data consistency, while asynchronous programming techniques optimize performance by enabling concurrent operations.

This well-structured and practical approach simplifies DAOS utilization for developers. The project's modular design allows for easy integration of functionalities into existing projects, while the use of Python makes it accessible for developers with varying skillsets. Furthermore, PYDAOS-Programs leverages high-performance storage media like SSDs and NVMe drives to maximize performance, especially for data-intensive operations. This combination empowers users to manage data efficiently in various fields like high-performance computing, artificial intelligence, and data analytics.

8. TESTING

8.1   Introduction

Testing is a critical component of software development, ensuring that applications function as intended and meet specified requirements. The project leverages DAOS (Distributed Asynchronous Object Storage) with Python to handle large data, object serialization, pool and container management, and metadata synchronization. Effective testing for this project encompasses unit testing, integration testing, functionality testing, and performance testing as shown in table 8.1.

Unit Testing

Unit Testing involves testing individual components or modules of the software to ensure they work correctly in isolation. In this project, unit tests would validate functions like file chunking, object serialization, pool management, and metadata synchronization.

- **Isolated Tests:** Test each function independently.

- **Mocking:** Use mock objects to simulate DAOS API interactions.

- **Assertions:** Verify outputs against expected results, such as successful chunk uploads or correct serialization.

Integration Testing

Integration Testing checks the interactions between different modules to ensure they work together seamlessly. For this project, integration tests would verify the combined functionality of file uploading, object storage, and metadata management.

- **Module Interactions:** Ensure file chunking and serialization work together.

- **Data Flow:** Test the complete data flow from file input to DAOS storage.

- **End-to-End Scenarios:** Simulate real-world use cases, such as uploading and retrieving a file.

Functionality Testing

Functionality Testing ensures the software behaves according to its specifications and intended use. This involves testing the overall functionality of DAOS operations implemented in the project.

- **Feature Verification:** Test all implemented features like file uploads, object storage, and pool management.

- **Error Handling:** Verify the system's response to erroneous inputs or failed operations.

- **Performance:** Measure the time taken for DAOS operations and check if it meets performance benchmarks.

Usability Testing

Usability Testing evaluates the user interface and user experience to ensure the software is intuitive and easy to use. Although the project is primarily backend-focused, usability tests are still relevant for any user-facing scripts or command-line interfaces.

- **User Interface:** Assess the clarity and usability of command-line prompts and outputs.

- **Documentation:** Ensure that documentation and user guides are clear and helpful.

- **Simulated Scenrios :** Simulate usage scenarios to identify any usability issues, focusing on ease of interaction with CLI and program instructions.

Performance Testing

Performance Testing evaluates the system's stability and efficiency under various conditions. This testing is crucial to ensure the DAOS-integrated system performs well under expected and peak loads.

- **Load Testing:** Assess how the system performs under heavy load conditions.

- **Stress Testing:** Determine the system's behavior under extreme conditions, such as maximum file upload sizes.

- **Scalability Testing:** Evaluate the system's ability to scale up, handling increased data volumes and concurrent operations.

- **Throughput and Latency:** Measure the speed (throughput) and response time (latency) of DAOS operations.

The PYDAOS project demonstrates the powerful integration of Python with DAOS for efficient and scalable data management. Testing plays a vital role in ensuring the robustness and reliability of the system. By implementing comprehensive unit, integration, functionality, and performance tests, developers can ensure that the project not only meets its technical requirements but also performs efficiently under various

conditions. The modular design and use of high-performance storage media further enhance the project's practicality and efficiency, making it a valuable tool for various data-intensive applications.

Table 8.1: Table for Different Test Results

	Unit Testing	Integration Testing	Functional Testing	Usability Testing
DataChunkHandler	✓	-	✓	✓
ObjectHandler	✓	-	✓	✓
ServerMonitor	✓	✓	✓	✓
PoolContainerSelector	✓	✓	✓	✓
MetadataManager	✓	✓	✓	✓
ServerConfiguration Manager	✓	-	✓	-

8.2 Test Cases

The table 8.2 contains test cases for handling data chunking operations. It includes splitting data into chunks, merging chunks back into the original data, inserting data into storage, retrieving data from storage, deleting data, and listing all keys in storage, all these test cases pass. But the system fails to handle data when split into chunks of larger size. The system also fails to handle duplicate keys, deletes the old data without throwing any error.

Table 8.2: Test Cases for DataChunkHandler

Test Case ID	Description	Input	Expected Output	Actual Output	Pass/Fail
TC1.1	Split data into chunks	2048 bytes	2 chunks each of 1024 bytes	2 chunks	Pass
TC1.2	Merge chunks back	2 chunks		2048 bytes	Pass

	into original data		Original data of 2048 bytes		
TC1.3	Insert of very small data into storage	key: "file1", data: 2 bytes	Data stored in chunks, key in storage	Data Stored, Number of chunks:1	Pass
TC1.4	Insert of very large data into storage	key: "file1", data: 100MB	Data stored in chunks, key in storage	Data Stored Number of chunks: 100	Pass
TC1.5	Setting chunk size to very large value	key: "file1", data: 100MB chunk size:50MB	Data stored in chunks, key in storage	DER_NOMEM error.	Fail
TC1.6	Setting chunk size to very large value	key: "file1", data: 100MB chunk size:1KB	Data stored in chunks, key in storage	Runtime Error	Fail
TC1.7	Retrieve data from storage	key: "file1"	Original data of 2048 bytes	2048 bytes Retrieved object matches with original	Pass
TC1.8	Delete data from storage	key: "file1"	Data and key removed from storage	Data and key removed	Pass
TC1.9	Insert duplicate key	Key:"file1"	Key already exists	Original data is replaced by new data	Fail
TC1.10	Handle non-existent key	key: "nonexistent"	KeyError	KeyError	Pass

This table 8.3 includes test cases for storing, retrieving the objects. It also tests the serialization and deserialization processes, the system handles these test cases successfully.

Table 8.3: Test Cases for ObjectStorageHandler

Test Case ID	Description	Input	Expected Output	Actual Output	Pass/Fail
TC2.1	Store and retrieve object	{"key": "value"}	Retrieved object matches stored object	Matches	Pass

Test Case ID	Description	Input	Expected Output	Actual Output	Pass/Fail
TC2.2	Store and retrieve different varieties of objects (list/tuple/dict/set)	Object of different category	Retrieved object matches stored object	Matches	Pass

This table Table 8.4 contains test cases for monitoring server status, retrieving lists of pools and containers, and handling scenarios with empty lists. It ensures the system correctly fetches and displays server status and related information.

Table 8.4: Test cases for ServerMonitor

Test Case ID	Description	Input	Expected Output	Actual Output	Pass/Fail
TC3.1	Get list of pools and containers	N/A	List of pools and containers in json format	Json file containing pools and containers list	Pass
TC3.2	Running in system with no pools and containers	N/A	File with no content	Empty file	Pass

The table 8.5 includes test cases for selecting the optimal pool and container based on metrics. It also covers handling scenarios with no available pools or containers, and updating pool metrics to reflect changes.

Table 8.5: Test cases for PoolContainerSelector

Test Case ID	Description	Input	Expected Output	Actual Output	Pass/Fail
TC4.1	Select optimal pool	N/A	Pool with the highest number of targets and largest free space	Optimal pool	Pass
TC4.2	Select optimal container	Pool id	Container selected in the order of round robin method	Optimal container	Pass

Test Case ID	Description	Input	Expected Output	Actual Output	Pass/Fail
TC4.3	Handle no available pools and containers	Empty pool list	None	None	Pass

The table 8.6 contains test cases for updating and retrieving metadata associated with keys. In all these scenarios the system works correctly

Table 8.6: Test cases for MetadataManager

Test Case ID	Description	Input	Expected Output	Actual Output	Pass/Fail
TC5.1	Update metadata	N/A	Metadata is updated correctly and retrievable	Updated And Retrievable	Pass
TC5.2	Retrieve metadata	key: "file1"	Metadata associated with the key	Metadata	Pass

The table 8.7 includes test cases for retrieving server configurations, which system handles successfully, but fails to handle when server configuration changes dynamically

Table 8.7: Test cases for **ServerConfigurationManager**

Test Case ID	Description	Input	Expected Output	Actual Output	Pass/Fail
TC6.1	Get server configurations	List of Server ID	Configuration details of the server	Config details	Pass
TC6.2	Handle servers that are not running	Server ID	Empty File	File just containing server names but no configuration information	Pass
TC6.3	Handle the changes made with server configuration	List of Server ID	Changes should be reflected	Changes are not reflected	Fail

9. RESULTS & PERFORMANCE ANALYSIS

9.1 Results

9.1.1 DAOS-backed Key-Value Store with File Management

This program implements a command-line key-value store that utilizes DAOS for persistent and high-performance storage. Users can interact with the store to manage key-value pairs as shown in fig 9.1. Key functionalities include:

- Uploading files to create new key-value pairs, where the file content becomes the value associated with the key.

- Reading the value of an existing key and saving it as a file.

- Deleting a key-value pair.

- Listing all existing keys.

- Uploading multiple key-value pairs in bulk from separate files.

The program leverages DAOS for storing the key-value pairs, ensuring data persistence and high performance. It also manages uploaded files associated with keys in a dedicated "uploads" directory.

```
[root@localhost hpecty]# python3 prog2.py
?        - Print this help
r        - Read a key
u        - Upload file for a new key
ub       - Upload files for new keys in bulk
d        - Delete key
p        - Display keys
q        - Quit

Commands:
Enter command (? for help): u
Enter new key: k1
Enter path to file: 100mb.txt
File uploaded successfully.

Commands:
Enter command (? for help): r
Enter key to read: k1
Value saved as file: uploads/k1.dat

Commands:
Enter command (? for help): p
k1

Commands:
Enter command (? for help): d
Enter key to delete: k1
Key deleted successfully.

Commands:
Enter command (? for help): p

Commands:
Enter command (? for help): ub
Enter the number of keys to insert: 2
Enter key 1: k1
Enter path to file for key k1: 100mb.txt
File uploaded for key k1 successfully.
Enter key 2: k2
Enter path to file for key k2: file1.pdf
File uploaded for key k2 successfully.

Commands:
Enter command (? for help): p
k2
k1

Commands:
Enter command (? for help): r
Enter key to read: k2
Value saved as file: uploads/k2.dat

Commands:
Enter command (? for help): ☐
```

Fig 9.1: Output for the kv store program

9.1.2 Chunking

This program extends the previous key-value store with functionalities to handle large files in chunks.

- **Chunking:** It allows users to specify a chunk size in megabytes (MB). Large files are split into chunks of this size during upload and retrieval operations.

- **Time Measurement:** The program measures the time taken for uploading and downloading data chunks using the bput and bget operations of the DAOS dictionary object.

- **Modified Read Function:** The read_key function is modified to handle chunked data. It retrieves all chunks for a key using their key names (e.g., "keychunk0", "keychunk1") and assembles them before saving the data as a file.

- **Modified Upload Function:** The upload_file function is modified to handle chunking. It reads the file in chunks and stores each chunk as a separate key-value pair in the DAOS dictionary using a key prefix followed by a chunk number (e.g., "keychunk0", "keychunk1").

- **Unique Key Listing:** The print_keys function is improved to avoid listing duplicate keys for chunked data. It extracts the key prefix (original key) and lists only unique prefixes.

Overall, this program enhances the key-value store to handle large files efficiently by splitting them into chunks for storage and retrieval in DAOS. It also measures the time taken for these operations as shown in fig 9.2 ,9.3, 9.4.

```
[root@localhost hpe-cty]# python3 chunks3.py
Enter the size of the chunks (in MB) : 10

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
q       - Quit
Enter command (? for help): u
Enter new key: key7\
Enter path to file: 200mb.txt
File uploaded in 21 chunks successfully.Time taken: 1.5637426376342773 seconds

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
q       - Quit
Enter command (? for help): r
Enter key to read: key7\
Value saved as file: uploads/key7\.dat
Value retrieved successfully. Total chunks: 21.Time taken: 1.8377153873443604 seconds
```

Fig 9.2: Uploading 200MB file as 10MB chunks

```
[root@localhost hpe-cty]# python3 chunks3.py
Enter the size of the chunks (in MB) : 25

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
q       - Quit
Enter command (? for help): u
Enter new key: key6
Enter path to file: 200mb.txt
File uploaded in 9 chunks successfully.Time taken: 1.4199693202972412 seconds

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
q       - Quit
Enter command (? for help): r
Enter key to read: key6
Value saved as file: uploads/key6.dat
Value retrieved successfully. Total chunks: 9.Time taken: 1.67698073387146 seconds
```

Fig 9.3: Uploading 200MB file as 25MB chunks

```
[root@localhost hpe-cty]# python3 chunks3.py
Enter the size of the chunks (in MB) : 50

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
q       - Quit
Enter command (? for help): u
Enter new key: key5
Enter path to file: 200mb.txt
File uploaded in 5 chunks successfully.Time taken: 1.5447664260864258 seconds

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
q       - Quit
Enter command (? for help): r
Enter key to read: key5
Value saved as file: uploads/key5.dat
Value retrieved successfully. Total chunks: 5.Time taken: 0.8875629901885986 seconds
```

Fig 9.4: Uploading 200MB file as 50MB chunks

9.1.3   Querying the Pools and Containers

This program is designed to automate the discovery and enumeration of existing Distributed Asynchronous Object Storage (DAOS) pools and their associated containers. The retrieved information is subsequently persisted in a human-readable and machine-interpretable JSON format as depicted in fig 9.5, facilitating utilization by subsequent programs for further analysis and management tasks.

Process:

1. Pool Discovery: The program initiates the process by querying the DAOS management tool (dmg) via the pool list command. This command passively retrieves a comprehensive listing of all currently active DAOS pools within the storage system.

2. Container Enumeration: For each discovered pool, a subsequent daos cont list command is executed, targeting the specific pool name. This command passively gathers a detailed enumeration of all containers currently residing within the designated pool.

3. JSON Output Generation: The program meticulously constructs a JSON data structure that encapsulates the discovered pools and their corresponding containers. This structure adheres to established JSON formatting conventions, ensuring both human readability and machine interpretation. The generated JSON object is then persisted to a designated file named "pool_metadata.json".

Benefits:

- Enhanced Efficiency: By automating the discovery process and providing a centralized JSON output, this program streamlines the workflow for subsequent programs that require access to DAOS pools and containers.

- Improved Readability and Abstraction: The utilization of JSON as the output format fosters improved human readability and facilitates programmatic interaction with the discovered resources.

- Foundation for Further Analysis: This program serves as a critical initial step, laying the groundwork for subsequent programs to perform more comprehensive analysis and management of DAOS storage resources.

In essence, this program passively gathers and presents a comprehensive inventory of DAOS pools and their contained objects, thereby establishing a foundation for further exploration and management by subsequent programs within the DAOS ecosystem.

Fig 9.5: JSON file with pools and containers

9.1.4   Optimal Poll Selection

This program is designed to automate the selection of an optimal DAOS pool for subsequent programs. The selection process prioritizes two key factors to ensure efficient resource allocation and potentially improved performance within the DAOS system.

Prioritization Criteria:

1. Number of Targets: A query is initiated using the dmg pool query command to retrieve the number of targets (storage servers) associated with each pool. Subsequently, pools with a higher number of targets are preferentially chosen. This prioritization is based on the assumption that a greater number of targets translates to increased parallelism and potentially better performance for data access and manipulation operations within the chosen pool.

2. Free Space: In the event that multiple pools possess an identical number of targets, an additional selection criterion is employed. Regular expressions are passively applied to the output generated by the dmg pool query command to extract the available free space information for each pool. The pool with the highest amount of free space is then selected. This approach ensures efficient storage utilization by allocating subsequent programs to pools with sufficient available capacity.

Selection Process:

1. A comprehensive listing of all DAOS pools is passively retrieved using the dmg pool list command.

2. For each identified pool, details including the number of targets and free space are passively extracted via the dmg pool query command.

3. An iterative process is then undertaken to examine the retrieved pool information.

4. Pools are prioritized based on the number of targets, with preference given to those exhibiting the highest target count.

5. If a scenario arises where multiple pools possess the same number of targets, a tie-breaking mechanism is implemented. This mechanism passively selects the pool with the most free space.

6. Upon identification of the optimal pool based on the aforementioned criteria, a list of containers residing within that specific pool is retrieved using the daos cont list command.

Benefits for Subsequent Programs:

By passively selecting the optimal pool based on this established strategy, the program offers several advantages for other DAOS programs that leverage its functionality:

- Streamlined Configuration: Subsequent programs are relieved of the burden associated with manual pool selection, resulting in a simplified configuration and setup process.

- Enhanced Performance Potential: By prioritizing pools with a higher number of targets, the program fosters the potential for improved performance within programs that rely heavily on data access and manipulation operations.

- Efficient Resource Utilization: Selecting pools with sufficient free space passively ensures efficient storage allocation and avoids overloading of pools with limited capacity.

9.1.5 Storing Metadata and Synchronization

This program synchronizes metadata about key-value stores implemented using DAOS into a JSON file named "metadata.json".

1. Gathering Key-Value Store Metadata:

- It iterates over each pool and container combination.

- For each container, it creates or retrieves a DAOS dictionary object named "pydaos_kvstore_dict".

- It extracts unique key prefixes (assuming chunked data) from all keys in the DAOS dictionary.

- For each unique key prefix, it checks if a key with the prefix ending in "chunk0" exists. If found, it retrieves that chunk data to determine the chunk size by measuring its length in bytes and converting it to megabytes (MB).

2. Storing Metadata as JSON:

- It builds a list of dictionaries, where each dictionary contains information about a key:

  - Key prefix (original key)

  - Chunk size (in MB, 0 if no chunk size information found)

  - Pool name

  - Container name

- Finally, it writes this list of key-value store metadata as a JSON file named "metadata.json" as shown in fig 9.6.

In summary, this program discovers DAOS pools and containers, gathers metadata about key-value stores within them (including key prefix, chunk size if applicable, pool, and container), and stores this information in a JSON file for potential future use.

Fig 9.6: Metadata JSON file

9.1.6 Optimized Selection of Pools and Chunking(built on 9.4 & 9.5)

This program provides a dynamic and user-friendly tool for managing key-value data within a DAOS system. It prioritizes resource optimization through dynamic pool selection and round-robin container allocation, while maintaining data integrity through meticulous metadata synchronization as shown in fig 9.7 and 9.8.

- **File Management:**

    o The program allows uploading files as the values for keys.

    o It splits large files into chunks of a specified size (in MB) before storing them in the DAOS container using the bput method.

    o Conversely, during read operations, it retrieves all chunks associated with a key and assembles them into the original data before presenting it to the user.

- **Dynamic Pool Selection (from 9.4):** The program leverages the pool_test.py module to discover DAOS pools. It avoids hardcoding pool details and instead relies on this module to identify the pool with the **highest number of targets**. This approach promotes potentially better performance by utilizing pools with more resources for data storage and access.

- **Round-Robin Container Selection:** When interacting with the key-value store, the program employs a round-robin strategy for selecting containers within the chosen pool. This ensures a fair distribution of data across available containers and prevents overloading any single container.

- **Metadata Synchronization (from 9.5):** Following each operation (upload, read, delete), the program meticulously synchronizes the metadata file named "pool_metadata.json". This file stores crucial information about the keys, including:

    o Key name

    o Chunk size (if applicable)

    o Name of the DAOS pool where the container resides (using dynamic selection)

    o Name of the container within the pool (selected using round-robin)

This emphasis on metadata synchronization ensures the program maintains an accurate record of key locations and associated resources, facilitating efficient data management.

```
[root@localhost manoj]# vi pool_test.py
[root@localhost manoj]# python3 auto_chunk.py

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
d       - Delete a key
q       - Quit
Enter command (? for help): u
Enter new key: k1
Enter path to file: 100MB.txt
Enter size of chunks (in MB): 25
File uploaded in Pool:pydaos1, Container:kvstore2, 5 chunks successfully. Time taken: 2.6819722652435303 seconds

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
d       - Delete a key
q       - Quit
Enter command (? for help): r
Enter key to read: k1
Value saved as file: uploads/k1.dat
Value retrieved successfully. Total chunks: 5. Time taken: 1.3976233005523682 seconds

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
d       - Delete a key
q       - Quit
Enter command (? for help): p
Key                            chunk size              Pool                    Container
--------------------------------------------------------------------------------------
k1                             25                      pydaos1                 kvstore2
```

Fig 9.7: Output for the optimal selection and auto-chunking program-part 1

```
p       - Display keys
d       - Delete a key
q       - Quit
Enter command (? for help): u
Enter new key: k1
Enter path to file: 100MB.txt
Enter size of chunks (in MB): 25
File uploaded in Pool:pydaos1, Container:kvstore2, 5 chunks successfully. Time taken: 2.6819722652435303 seconds

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
d       - Delete a key
q       - Quit
Enter command (? for help): r
Enter key to read: k1
Value saved as file: uploads/k1.dat
Value retrieved successfully. Total chunks: 5. Time taken: 1.3976233005523682 seconds

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
d       - Delete a key
q       - Quit
Enter command (? for help): p
Key                            chunk size              Pool                    Container
--------------------------------------------------------------------------------------
k1                             25                      pydaos1                 kvstore2

Commands:
?       - Print this help
r       - Read a key
u       - Upload file for a new key
p       - Display keys
d       - Delete a key
q       - Quit
```

Fig 9.8: Output for the optimal selection and auto-chunking program-part 2

9.1.7 Object Storage and Retreival

This is a suite of python programs that demonstrate the utilization of a Distributed Key-Value (DKV) store implemented with DAOS for the exchange of data and functionalities between programs. The DKV store serves as a centralized repository, enabling a paradigm shift from direct program-to-program communication to a more decoupled and scalable approach.

1. Sharing Class Definitions:

The initial programs facilitate the dissemination of class definitions across a distributed environment. A class, encapsulating both data attributes and methods, is defined within a program acting as the sender. Subsequently, the class definition is serialized, transforming it into a machine-readable byte stream suitable for storage. This serialized representation is then deposited within the DAOS DKV store.

A separate program, functioning as the receiver, retrieves the serialized class definition from the DKV store. The retrieved data is subsequently deserialized, reconstructing the original class definition within the receiver's environment. This dynamic execution of the definition effectively creates an identical class on the receiving program, enabling the exchange of custom functionalities across the distributed system.

2. Sharing Python Objects

Another set of programs exemplifies the exchange of arbitrary Python objects through the DKV store. A program acting as the sender defines a Python data structure, such as a dictionary, to encapsulate the information to be shared. This data structure is then serialized using a dedicated library, converting it into a format optimized for storage within the DKV store. The serialized data is subsequently stored within the DKV store under a pre-defined key, facilitating efficient retrieval.

A separate program, acting as the receiver, retrieves the serialized data from the DKV store using the key. The retrieved data is then deserialized using the same library employed during the serialization process, transforming it back into its original Python object representation within the receiver's environment. This mechanism enables the

exchange of various data structures and objects between distributed programs, fostering collaboration and data sharing across the system as shown in fig 9.9.
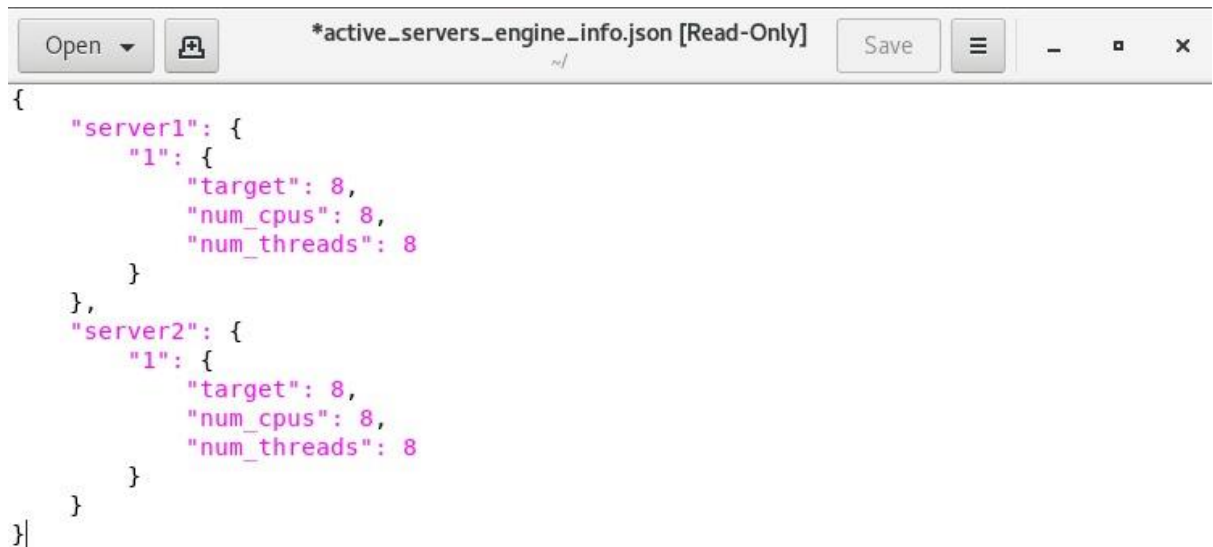
```
[root@localhost manoj]# vi sender_dict.py
[root@localhost manoj]# vi reciver_dict.py
[root@localhost manoj]# python3 sender_dict.py
Python object saved successfully.
[root@localhost manoj]# python3 reciver_dict.py
Name: John
Age: 30
City: New York
[root@localhost manoj]#
```

Fig 9.9: Object storage and retrieval

9.1.8   Monitoring Server Nodes

This Python program automates the process of collecting information about DAOS server engines running on multiple remote servers. It acts like a remote DAOS server engine information gatherer.

1. Server Credentials: The script defines a dictionary containing login credentials (IP address, username, password) for each server you want to check.

2. SSH Connection: It connects to each server using SSH with the provided credentials.

3. DAOS Service Check: It verifies if the DAOS server service is actively running on the remote server.

4. Engine Configuration Retrieval: If the service is running, the script retrieves the /etc/daos/daos_server.yml configuration file containing information about the DAOS server engines on that server.

5. Engine Info Extraction: The script parses the retrieved configuration file and extracts details about each engine, such as the number of targets it manages and the number of CPU cores and threads it utilizes.

6. Data Storage: Finally, the script stores the collected information about active servers and their engine configurations in a JSON file for easier access and analysis as depicted in fig 9.10.

```
Open  ▼    ⊞         *active_servers_engine_info.json [Read-Only]    Save   ≡   _   ▫   ✕
                              ~/

{
    "server1": {
        "1": {
            "target": 8,
            "num_cpus": 8,
            "num_threads": 8
        }
    },
    "server2": {
        "1": {
            "target": 8,
            "num_cpus": 8,
            "num_threads": 8
        }
    }
}
```

Fig 9.10: JSON file with information of server nodes and engines

9.2   Performance Analysis

Directly uploading the file

This analysis examined the direct upload performance of varying file sizes within a DAOS key-value store. The results revealed a non-linear relationship between file size and upload time as shown in fig 9.11. The upload times for larger files increased more significantly than expected. Compared to smaller files, larger files exhibited a much sharper increase in upload time, with the 50MB file taking approximately 6.6 times longer to upload than the 25MB file. This trend continued for the 100MB file, which showed a slowdown of approximately 3.5 times compared to the 50MB file. These observations suggest potential limitations in handling larger files. Two possible explanations could be network bandwidth saturation for larger uploads and a base system overhead within DAOS that applies regardless of file size.
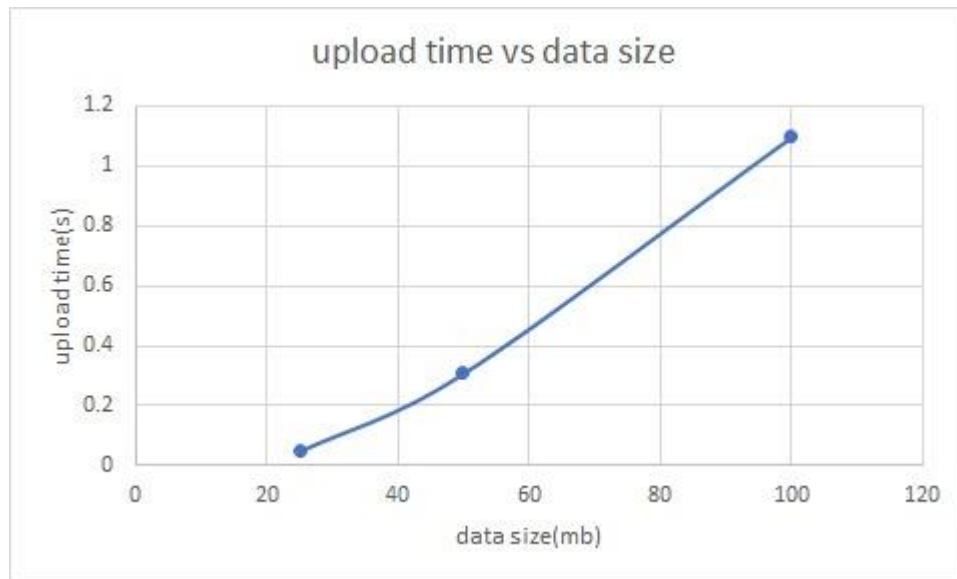
Fig 9.11: Plot of Upload time vs Data size

Uploading the file in chunks

An analysis was conducted to examine the impact of chunk size on upload times within a DAOS key-value store. A 104MB file was uploaded in varying chunk sizes (1MB, 5MB, 25MB, 50MB, and 100MB). Interestingly, the results indicated that upload times decreased as chunk size increased as depicted in fig 9.12. This observation is counterintuitive to the initial assumption that smaller chunks would be faster.

While DAOS offers advantages for large data transfers, it was observed that there appears to be an overhead associated with initiating each chunk upload within the key-value store. This overhead likely encompasses establishing connections, sending metadata, and finalizing transfers. For smaller chunks (1MB), a larger proportion of the total upload time is consumed by this overhead compared to larger chunks (100MB).

It is important to note that this analysis solely focused on upload time. Breaking the file into smaller chunks may offer advantages in other areas within DAOS:

- **Error Handling:** In the event of a transfer failure during upload, smaller chunks would allow for resuming the upload from the point of failure rather than restarting the entire file.

- **Parallel Processing:** DAOS may enable parallel uploads of multiple chunks, potentially leading to an acceleration of the overall process for smaller chunks (although network limitations could still apply.
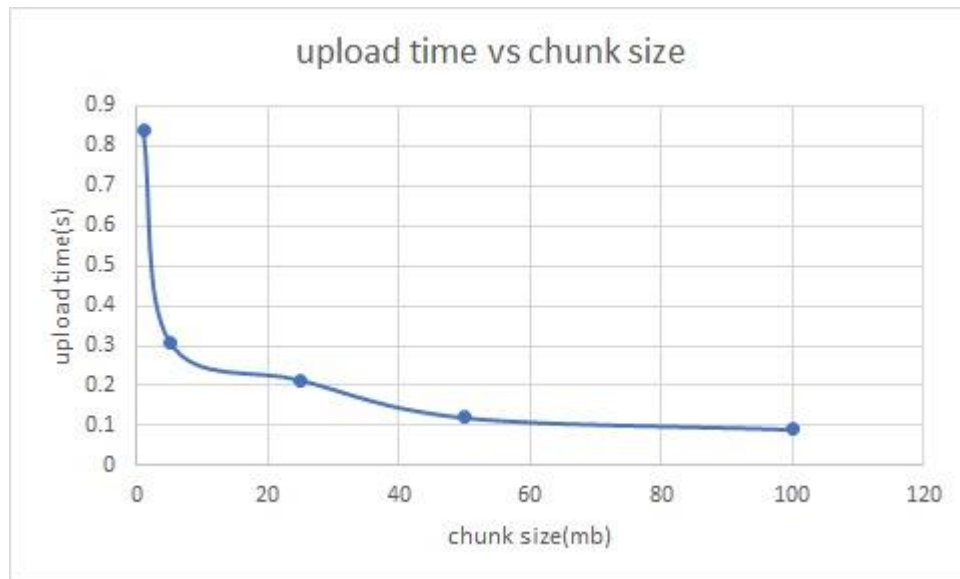
Fig 9.12: Plot of Upload time vs Chunk size

Retrieving the file in chunks

Following the analysis of upload times, a complementary investigation was conducted to examine the impact of chunk size on retrieval times within a DAOS key-value store. The same 104MB file, previously uploaded in varying chunk sizes (1MB, 5MB, 10MB, 25MB, 50MB, and 100MB), was used for retrieval testing. Interestingly, the results mirrored the upload analysis, indicating that retrieval time decreased as chunk size increased as depicted in fig 9.13. This observation reinforces the presence of an overhead associated with each chunk operation within the DAOS key-value store. Consistent with the findings from the upload analysis, this overhead likely encompasses establishing connections, sending metadata (requests for retrieval in this case), and receiving data transfers. As observed previously, with increasing chunk size, this overhead becomes a smaller proportion of the total retrieval time.

The overhead ratio is again an estimated value based on the observed retrieval times. It highlights that smaller chunks are significantly impacted by this overhead compared to larger chunks. Additionally, efficient data access patterns within DAOS for larger chunks might further contribute to faster retrieval times, similar to the potential benefits observed during uploads.

A potential difference between upload and retrieval behavior is acknowledged. Network limitations can play a more significant role in retrieval times, especially for very large chunk sizes. In scenarios with limited bandwidth, retrieving a single large chunk might not be as

efficient as retrieving multiple smaller chunks due to network congestion. This introduces a trade-off that wasn't apparent in the upload analysis.
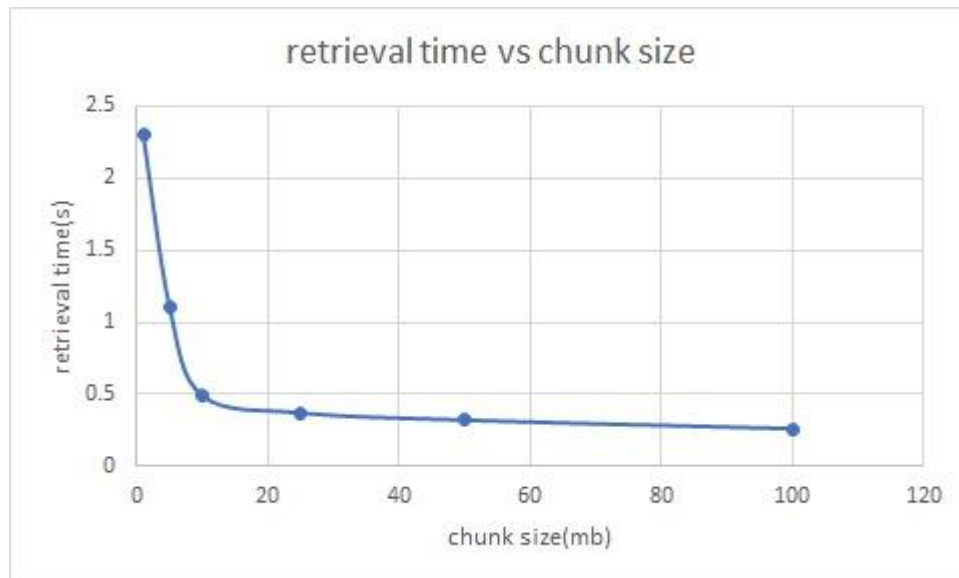

Fig 9.13: Plot of Retrieval time vs Chunk size

10. CONCLUSION & SCOPE FOR FUTURE WORK

10.1 Findings and suggestions

This project demonstrates the development of a versatile suite of Python programs for interacting with DAOS. The key findings include:

- **Efficient Key-Value Store:** The command-line key-value store provides a user-friendly interface for managing key-value pairs with DAOS for persistence. Chunking large files and automatic pool/container selection optimize storage and resource utilization.

- **Simplified DAOS Management:** Programs automate DAOS pool and container discovery, selection, and metadata management, simplifying system administration tasks.

- **Distributed Data Sharing:** The DKV store facilitates communication between programs by enabling the exchange of class definitions and Python objects, promoting collaboration and data sharing.

- **Automated Server Monitoring:** The server monitoring program automates the collection of information about DAOS server engines on remote servers, aiding system management and health checks.

10.2 Significance of the Proposed Research Work:

- **Enhanced DAOS Usability:** The developed programs offer a user-friendly and efficient way to interact with DAOS for various tasks, making it more accessible to researchers and developers.

- **Optimized Data Management:** Chunking, pool/container selection, and metadata synchronization improve data storage efficiency and resource utilization within DAOS.

- **Streamlined Distributed Programming:** The DKV store simplifies communication and data exchange between programs in a distributed DAOS environment.

- **Simplified Server Administration:** Automated DAOS pool/container management and server monitoring reduce administrative burden and provide valuable insights for system maintenance.

10.3 Limitations of this Research Work:

- **Focus on Core Functionalities:** The current implementation prioritizes core functionalities. Advanced features like security, error handling, and performance monitoring require further development.

- **Scalability Testing:** Extensive testing is needed to evaluate the scalability of the programs under high data volumes, concurrent operations, and large numbers of keys.

- **Integration with HPC Frameworks (Future Work):** The potential integration of the DAOS interface with the Array Context extension for high-performance computing applications needs further investigation.

10.4 Directions of Future Work:

- **Advanced Chunking Strategies:** Explore adaptive chunking algorithms that consider data characteristics and storage behavior for improved efficiency.

- **Integration with HPC Frameworks:** Investigate the integration of the PyDAOS interface with the Array Context extension (if implemented) into existing HPC frameworks for large-scale data handling.

- **Security Enhancements:** Implement robust authentication and authorization mechanisms to control access to DAOS resources and stored data.

- **Error Handling and Recovery:** Develop comprehensive error handling and recovery strategies to address potential issues during data transfer and manipulation operations.

- **Performance Monitoring and Optimization:** Implement mechanisms to monitor and optimize the performance of DAOS key-value store operations, including chunk size impact and resource utilization.

- **Scalability Testing:** Conduct extensive testing to evaluate the program's scalability as the number of keys, data size, and concurrent operations increase within the DAOS system.

By addressing these limitations and pursuing the proposed future work directions, this project can significantly enhance the usability, functionality, and scalability of DAOS for diverse applications involving big data, distributed systems, and high-performance computing.

REFERENCES

[1] Luke Logan, Jay Lofstead, Xian-He Sun, and Anthony Kougkas. 2023. An Evaluation of DAOS for Simulation and Deep Learning HPC Workloads. In Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '23). Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/3578353.3589542

[2] N. Manubens, T. Quintino, S. D. Smart, E. Danovaro and A. Jackson, "DAOS as HPC Storage: a View From Numerical Weather Prediction," *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, St. Petersburg, FL, USA, 2023, pp. 1029-1040, doi: 10.1109/IPDPS54959.2023.00106.

[3] Michael Hennecke. 2023. Understanding DAOS Storage Performance Scalability. In Proceedings of the HPC Asia 2023 Workshops (HPCAsia '23 Workshops). Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3581576.3581577

[4] Jackson and N. Manubens, "DAOS as HPC Storage: Exploring Interfaces," *2023 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, Santa Fe, NM, USA, 2023, pp. 8-10, doi: 10.1109/CLUSTERWorkshops61457.2023.00011

[5] F. Garcia-Carballeira, D. Camarmas-Alonso, A. Caderon-Mateos and J. Carretero, "A new Ad-Hoc parallel file system for HPC environments based on the Expand parallel file system," 2023 22nd International Symposium on Parallel and Distributed Computing (ISPDC), Bucharest, Romania, 2023, pp. 69-76, doi: 10.1109/ISPDC59212.2023.00015.

[6] Michael Hennecke, Motohiko Matsuda, and Masahiro Nakao. 2023. Evaluating DAOS Storage on ARM64 Clients. In Proceedings of the HPC Asia 2023 Workshops (HPCAsia '23 Workshops). Association for Computing Machinery, New York, NY, USA, 65–72. https://doi.org/10.1145/3581576.3581616

[7] J. Liu et al., "Evaluation of HPC Application I/O on Object Storage Systems," 2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS), Dallas, TX, USA, 2018, pp. 24-34, doi: 10.1109/PDSW-DISCS.2018.00005.

[8] Hennecke, M. (2022). Designing DAOS Storage Solutions with Lenovo ThinkSystem SR630 V2 Servers. https://lenovopress.lenovo.com/lp1421.pdf

[9] K. -J. Cho, I. Kang and J. -S. Kim, "ArkFS: A Distributed File System on Object Storage for Archiving Data in HPC Environment," *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, St. Petersburg, FL, USA, 2023, pp. 301-311, doi: 10.1109/IPDPS54959.2023.00038.

[10] N. Manubens, S. D. Smart, T. Quintino and A. Jackson, "Performance Comparison of DAOS and Lustre for Object Data Storage Approaches," *2022 IEEE/ACM International Parallel Data Systems Workshop (PDSW)*, Dallas, TX, USA, 2022, pp. 7-12, doi: 10.1109/PDSW56643.2022.00007.

[11] F. Garcia-Carballeira, D. Camarmas-Alonso, A. Caderon-Mateos and J. Carretero, "A new Ad-Hoc parallel file system for HPC environments based on the Expand parallel file system," 2023 22nd International Symposium on Parallel and Distributed Computing (ISPDC), Bucharest, Romania, 2023, pp. 69-76, doi: 10.1109/ISPDC59212.2023.00015.

[12] J. Lombardi, "ESSA 2022 Invited Speaker DAOS: Nextgen Storage Stack for HPC and AI," 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Lyon, France, 2022, pp. 1101-1101, doi: 10.1109/IPDPSW55747.2022.00180.

[13] B. Yang, Y. Zou, W. Liu and W. Xue, "An End-to-end and Adaptive I/O Optimization Tool for Modern HPC Storage Systems," 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 2022, pp. 1294-1304, doi: 10.1109/IPDPS53621.2022.00128.

[14] J. Soumagne et al., "Accelerating HDF5 I/O for Exascale Using DAOS," in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 4, pp. 903-914, 1 April 2022, doi: 10.1109/TPDS.2021.3097884.

[15] M. J. Heer, J. -E. R. Wichmann and K. Sano, "Achieving Scalable Quantum Error Correction with Union-Find on Systolic Arrays by Using Multi-Context Processing Elements," 2023 IEEE International Conference on Quantum Computing and Engineering (QCE), Bellevue, WA, USA, 2023, pp. 242-243, doi: 10.1109/QCE57702.2023.10224.

[16] Thekkath, C. A., Mann, T., & Lee, E. K. (Year). Frangipani: A Scalable Distributed File System. Systems Research Center, Digital Equipment Corporation, 130 Lytton Ave, Palo Alto, CA 94301.

[17] Wang, F., Brandt, S. A., Miller, E. L., & Long, D. D. E. (Year). OBFS: A File System for Object-based Storage Devices. Storage System Research Center, University of California, Santa Cruz, Santa Cruz, CA 95064.

[18] Lüttgau, J., Kuhn, M., Duwe, K., Alforov, Y., Betke, E., Kunkel, J., & Ludwig, T. (2018). Survey of storage systems for high-performance computing. Supercomputing Frontiers and Innovations, 5(1). Retrieved from https://centaur.reading.ac.uk/77664/

[19] Netto, M. A. S., Calheiros, R. N., Rodrigues, E. R., Cunha, R. L. F., & Buyya, R. (2018). HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. ACM Computing Surveys, 51(1), Article 8. https://doi.org/10.1145/3150224

[20] Paul, A. K., Wang, B., Rutman, N., Spitz, C., & Butt, A. R. Efficient Metadata Indexing for HPC Storage Systems. Virginia Tech & Cray Inc. Retreived from https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9139660.