Unison - File Storage Prototype

Advanced Database Systems

Final Report - 12/6/2018

Alfred Shaker
Betis Baheri
Paul Bryant

Abstract

Using GET or PUT Transferring data between HPC systems has a bottleneck over network and number of connection that one system can make. In order to solve this problem we need to use parallel PUT and GET to transfer information. On top of that we need to store the information not only to keep track of transfers, also, to keep the metadata in internal database. Rebuilding the metadata information is another problem, by using hash functions we can solve this problem. Executing programs on existing systems using normal user privileges is achieved by using MPI and running normal Restful API in python. Project focus is to use maximum and optimal resources of management system which will be act as a proxy server and maximize the network throughput using multiple connection.

Architecture

The design of the system can be separated into three distinct subsections: front-end (RESTful API), back-end storage (metadata database), and find parallel execution. For the purposes of supporting the front-end requirements we have chosen to utilize the Flask-RESTful Python extension (https://flask-restful.readthedocs.io/en/latest/index.html) that allows for creating and supporting flexible RESTful API implementations. This allows for clear definition or resource routing and argument parsing but is coupled with the necessary logic to support each request also in Python. Connections to cloud hosting applications are enabled through the officially supported methods (GCloud -> google-cloud-storage & AWS -> boto3).

The back-end is minimal at this time but aims to support tracking object identification, desired parallel/local storage location, MD5 hash (for potential verification), and required details in order to identify cloud location (bucket). In order to support flexible and quick prototyping this project required it was decided to leverage SQLlite as it meet the currently minimal performance needs but also well supported Python system level packages.

For parallelizing the put and get functions, we used MPI for Python which allows us to focus on writing our functions and optimizing those instead of worrying about the specifics of writing parallel code. We use the parallelization for splitting the file into equal chunks and adding it to the database, having each node take a chunk and setting up a direct connection to the database and saving the chunks under one folder named after the original file’s MD5 hash. When we want to get the information and join the file again, the same process is done in parallel but backwards, where we take each chunk and pass it to a node and have them combine each other’s data, then we verify with the MD5 hash that the data has been restored correctly.

Due to the shared knowledge and understanding among the team Python was utilized. In addition a standardized installation process was defined utilizing Pythons support setuptools, this allows for the potential of a single-line installation script coupled with a clearly support command line interface (Python’s argeparse package) for launching.

Parallel PUT/GET

Put and Get are standard functions for interacting with information in a database using a REST api. Put takes the files and adds them to specific locations in the database, while Get is used to retrieve that information. For our project, we used MPI to parallelize the process of adding and retrieving data from the database. We do this by first splitting up the file to be added so that each chunk is equal in size. We generate the MD5 hash of the original file and save that so that we can reference it later for when we retrieve the files, as each chunk and the repository where the file chunks are stored in the database use that same MD5 hash as the unique identifier for the file. The MD5 is also used to verify that the chunks have been combined successfully when a Get request is passed and that there is no corruption in the data. By splitting the files into equal size chunks, we are able to add and retrieve the files to and from the database in parallel using MPI and by assigning the appropriate amount of nodes and cores to the operation. MPI allows us to just execute these tasks in parallel with specified parameters without having to worry about the finer details of parallel programming, as with another parallel architectures like CUDA for example.

REST API

Clients does not have to use any special administrative privileges to transfer files between HPCs. In order to update the information projects need to have a update metadata function to update the information about stored data. Using flask API in python will be make it possible to keep our project simple and PUT or GET without adding an extra overhead to the system. Additional support has been implemented in some contexts in orders to allows for POST (registering new data with the metadata database) and DELETE (remove said registration).

System Analysis

Unison Parallel GET and PUT is based on creating multiple connection on N number of nodes in arbitrary HPC system, the scope of this project is to transfer files and keep the metadata information in one management system regardless of HPC systems, by using the same method as GET and PUT in python flask which is a restful API we can create an individual program and run them in MPI with different connection to target system. MPI is a manager for running programs in parallel and not only it uses the management system resources in a full scale also, it will be creating individual instances to target system with different connection to maximize the network throughput. In order to achieve this functionality, first we get an object id from user and then we can list the available client nodes on HPC system, depending on size of file we can create multiple chunks and split that into N number. After this process program will be gather the list of IP addresses of each node on the target system, and run the MPI on each individual destination. The information of each transfer will be stored in internal database on our system to not only keep track of transfers also, to rebuild the metadata in case of data loss. Each transfer will be hashed by MD5 function to ensure the integrity and validation.

Future Statistical Analysis

Internal database structure can help not only to rebuild but also, to reuse the same number of connections on known systems. The system is capable of storing information about each transfer which can be used in analysis of HPC transfers and build a statistical model. Later on models can be used to utilize the number of connections and resource usage to ensure the optimal performance. Performance of each transfer can be improved using information stored in internal database system. Using different implementation in MPI we can support new HPC systems. MPI is capable of running on GPU as well, to use more resources if there are available.

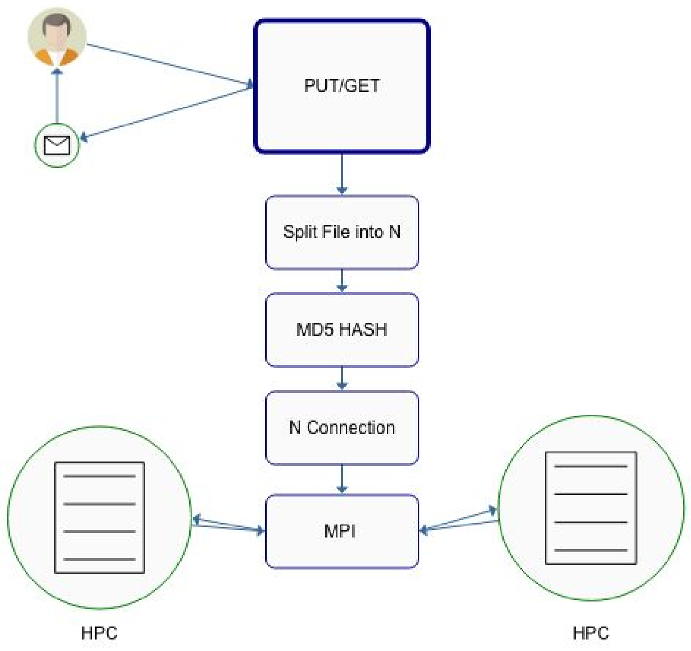
Source Code

Source code for the application can be found: https://github.com/betisb/AdvDB-Prototype.git

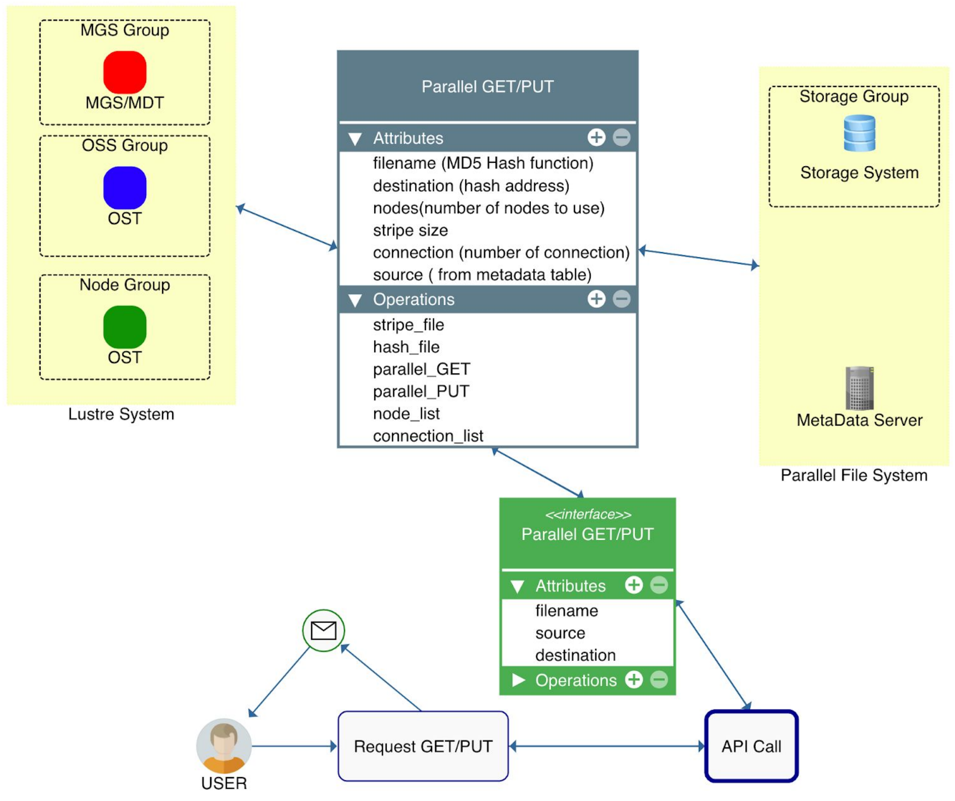
The repository has a detailed README.md file that aims to detail the installation process along with brief demonstrations of the codes functionality. In addition to this, the source code itself has been documented with the goal of allowing additional work to be continued by members of the team, so please be aware of the increased level of inline/method documentation and minor stylistic differences.

Diagrams

Overall Process:



Detailed Classes:


MGS Group
MGS/MDT
OSS Group
OST
Node Group
OST
Lustre System
Parallel GET/PUT
Attributes
filename (MD5 Hash function)
destination (hash address)
nodes(number of nodes to use)
stripe size
connection (number of connection)
source (from metadata table)
Operations
stripe_file
hash_file
parallel_GET
parallel_PUT
node_list
connection_list
Storage Group
Storage System
MetaData Server
Parallel File System
<<interface>>
Parallel GET/PUT
Attributes
filename
source
destination
Operations
USER
Request GET/PUT
API Call