

Simple Object Storage

Distributed object storage optimized for small files

Nijad Huseynov

Advisors: Stephen H. Kaisler, Jamaladdin Hasanov
SEAS GW University, SITE ADA University



3 August 2023

Outline

- Project objective
- System Design
- Data node components
- Implementation
- Features
- Evaluation
- Conclusion
- Future work

Project Objectives

Designing and implementing a distributed object storage based on the ideas from the Facebook haystack paper

Optimizing the object storage for large volumes of small files

Minimizing the metadata operations on small files, thus improving the read performance of the system

Features

The system is optimized for small objects and implemented for horizontal scalability

- ID generation for objects
- Volume manager. It is rule based periodic job that checks the statistics of the volumes from data node
- Object read and write
- Heartbeat mechanism
- Object corruption detection on read

Key ideas for optimizing for small files

In memory mapping of object id to the offset

Hashmap data structure is used.
Key is the object id and the value is offset of the object in the volume file.

Merge small files into one large file

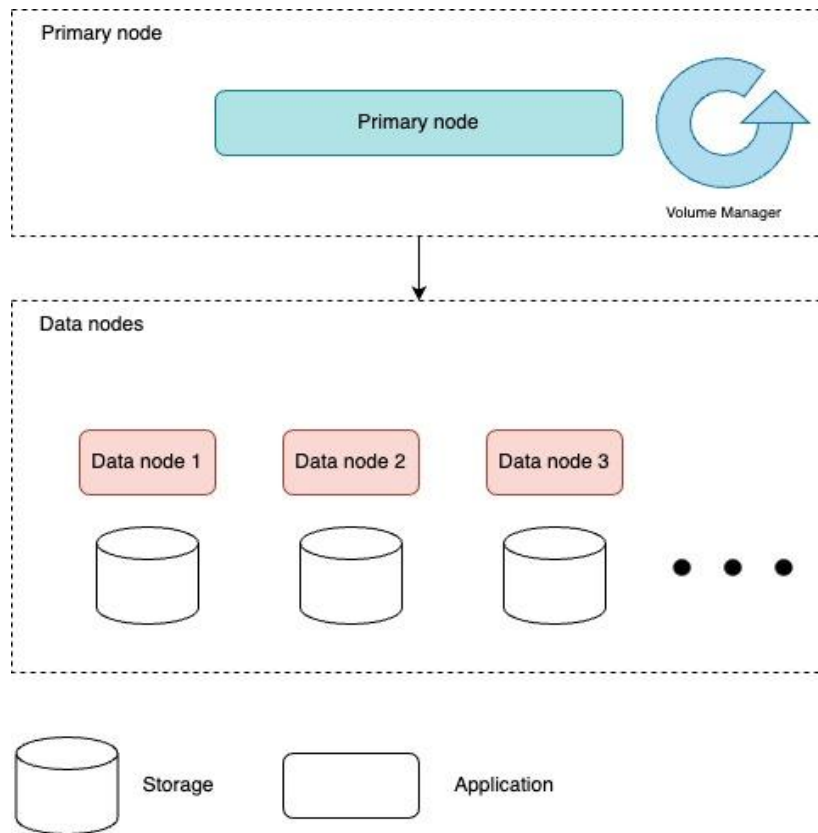
Lot of small files will have lots of metadata in local filesystem and it will be hard to cache those metadata. Thus we create one large file and append the object.

Keeping file descriptor open for each volume to reduce the disk IO requests to $O(1)$

Since we will have one large file, it will be easy to cache metadata of the file in ram, thus reducing the disk IO to 1

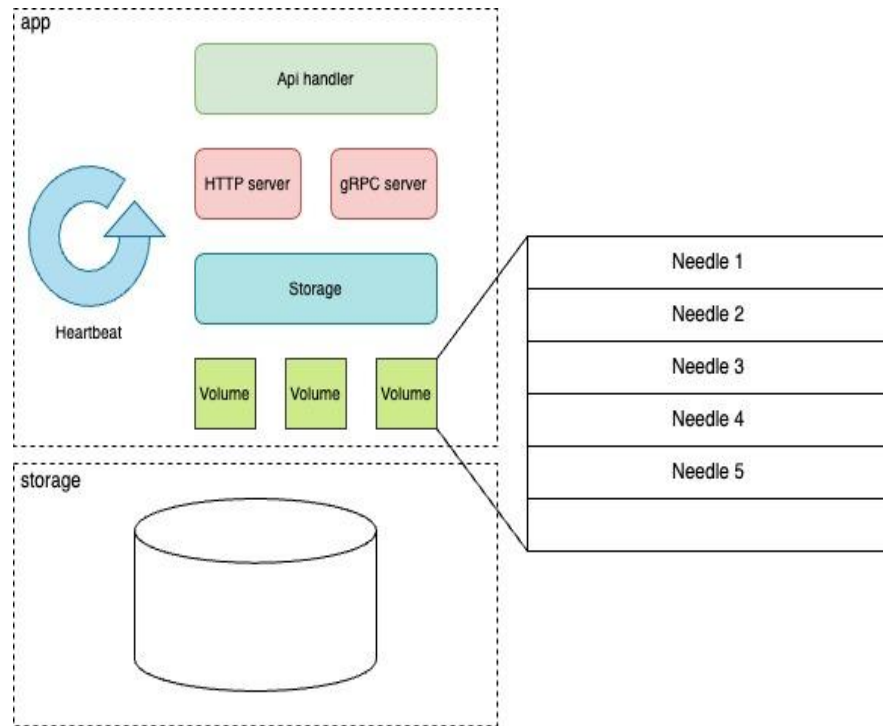
System design

- Primary node is responsible for managing volumes
 - Volume manager
 - ID generator
 - Cluster info
- Data node is responsible for storing data
 - Stores the objects in volumes as “needles”
 - Read and write apis for objects
 - Heartbeat process to primary node



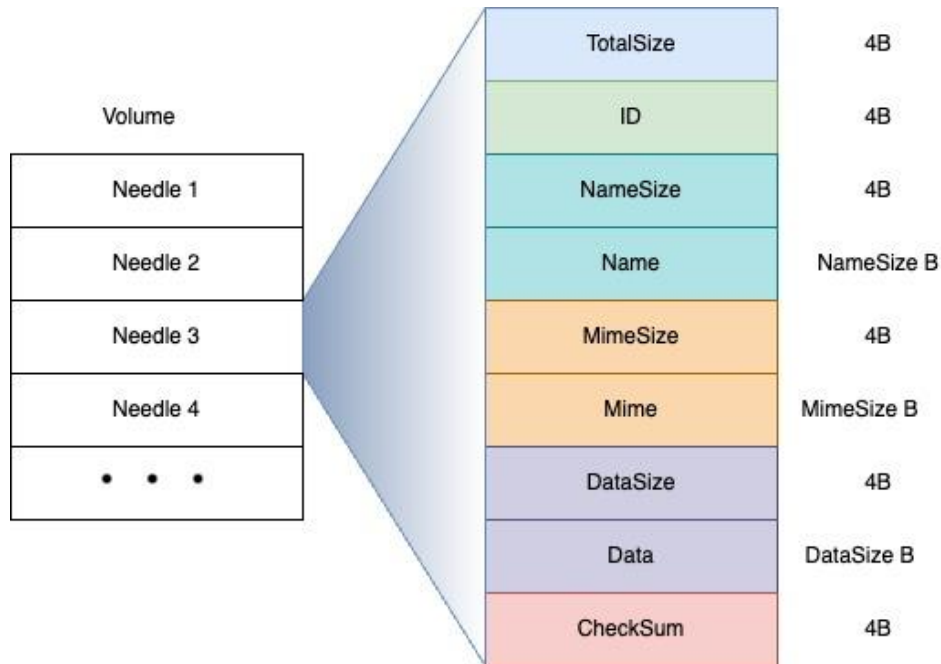
Data node components

- **Api handler** is responsible for the api calls. In this layer, there will be no logic apart from some input sanity checks. It will basically parse request params and pass it to the server layer
- **Server layer** will be responsible for the translation of the incoming requests to the storage layer like constructing the needle to the format storage layer understands and so on.
- **Storage** layer is responsible for the management of the volumes.
- **Volume** will contain some number of small objects(needles).



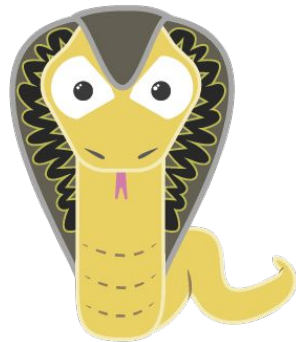
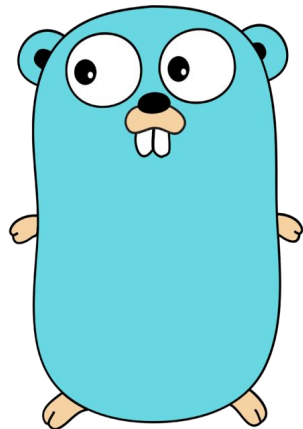
Needle - unit of data in the system

```
type Needle struct {  
    TotalSize  uint32 # Total size of Needle  
    Id         uint32 # Id of the writtent obj  
    NameSize   uint32 # Number of bytes in name  
    Name       []byte # Obj name  
    MimeSize   uint32 # Obj mime size  
    Mime       []byte # Obj mime type name  
    DataSize   uint32 # Obj content size  
    Data       []byte # Obj's content in bytes  
    Checksum   uint32 # Checksum of the obj  
}
```



Implementation

- Golang is used for the implementation
 - Simple
 - Fast
 - Concurrent
 - Portable
- Following open source golang libraries are used
 - Echo - for the http server
 - gRPC Go - for the gRPC server
 - Cobra - for the command line interface implementation
 - Logrus - for the logging



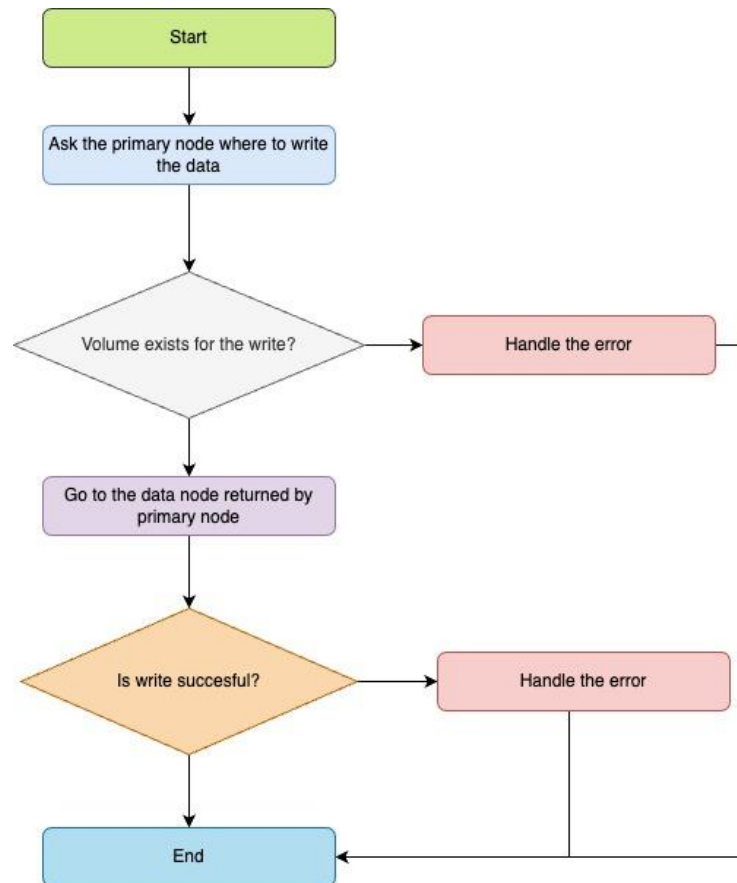
gRPC

Object write flow

Step 1. Client asks the location to write the object from the primary node

Step 2. Primary node returns the address of the data node which object can be written

Step 3. Client writes the object to the given data node.

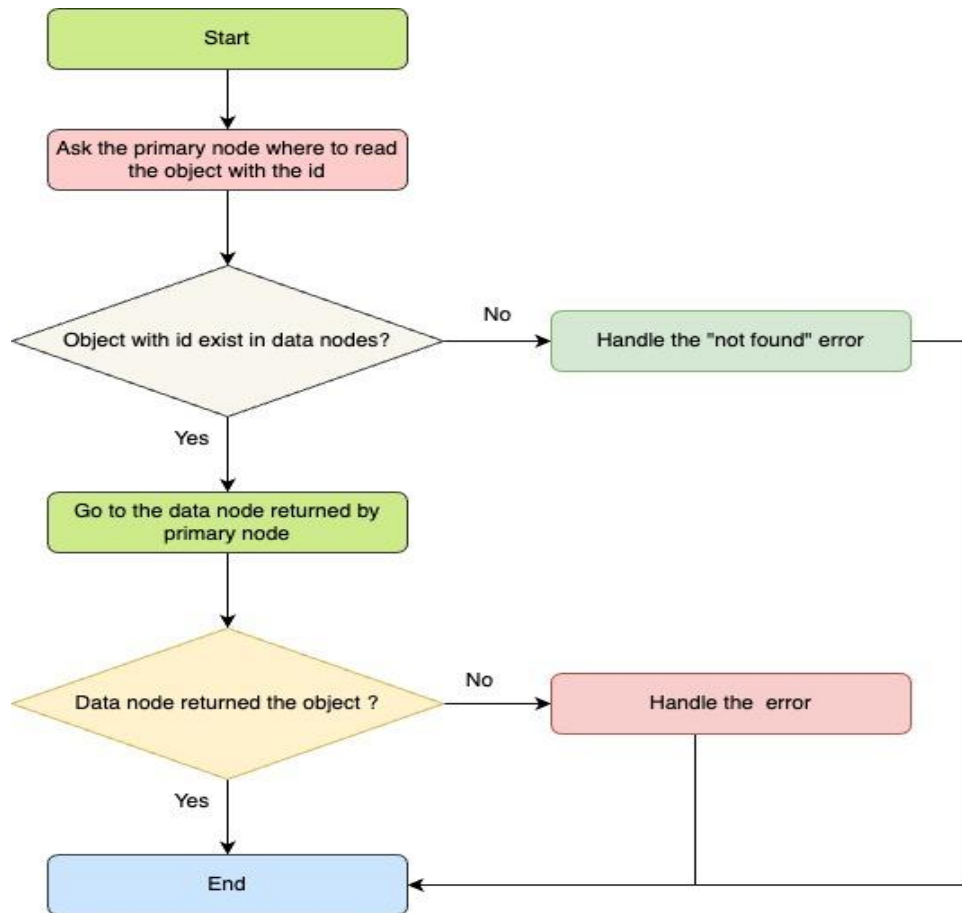


Object read flow

Step 1. Client asks the location of the object from the primary node

Step 2. Primary node returns the address of the data node which has the object

Step 3. Client requests the object from the data node.



Evaluation

- Setup
- Workloads

Conclusion

Future work

- Object deletes/updates
- Garbage collection
- Index for the object id to offset mapping
- Batched writes to take advantage of the HDDs

Backup