Annales Univ. Sci. Budapest., Sect. Comp. 56 (2024) 271–291

DISTRIBUTED STORAGE PATTERN

Jianhao Li and Viktória Zsók

(Budapest, Hungary)

Communicated by Zoltán Horváth

(Received January 9, 2024; accepted June 16, 2024)

Abstract. In this paper, the principles of a new distributed storage pattern are specified. By this architecture design we provide the system implementing this pattern with new distributed storage features. The pattern specification, implementation and extension possibilities are introduced. New distributed put, get, delete processes and their concurrent versions are tested and evaluated.
The distributed pattern of this paper is the general architecture description. Instead of having a set of rules, we provide distributed design principles which guide through architecture design. Programmers can modify or extend it based on their own needs.

1. Introduction

Nowadays, more and more useful applications are implemented by students and individual programmers. These developers often want to deploy their applications in experimental production environments to gain valuable experience. However, managing and implementing distributed storage systems can be challenging due to limited resources and the complexity of existing solutions. To address these challenges, the projects following this pattern can help them get the distributed storage service for their application using their limited resources. The project architecture can be easily understood and extended based on their needs. This pattern can also be used as the basic architecture of a distributed storage web service in any company.

Key words and phrases: Distributed systems, distributed storage.
2010 Mathematics Subject Classification: 68M14, 68Q85, 68W15.

The proposed pattern is a generalization of the distributed storage system introduced in the master thesis [5] implemented using the Go programming language [6] and the RabbitMQ [8] [12] (which supports the AMQP 0-9-1 Advanced Message Queuing Protocol [1]). This paper contains more details on the implementation and description of source code for distributed storage.

There are other Go distributed storage systems, like seaweedfs [10] and Kubo [7]. However, they are too huge and complex for individual programmers to understand and get guidelines from them that bring benefits to their own distributed system design in a short time. In contrast, our proposed pattern simplifies the process by offering clear design principles and an intuitive implementation. It enables developers to integrate distributed storage into their applications more efficiently.

2. Related work

Numerous distributed storage systems have been developed, each with unique features and architectures. This section highlights some notable projects and their relevance to our proposed pattern.

SeaweedFS [10] is a distributed object store and file system. It started as an Object Store to handle small files efficiently. Instead of managing all file metadata in a central master, the central master only manages file volumes, letting these volume servers manage files and their metadata. This feature relieves concurrency pressure from the central master and spreads file metadata into volume servers, allowing faster file access (O(1), usually just one disk read operation) [11]. It is implemented in Go and Java. However, its complexity can be a barrier for individual developers who need a more straightforward solution. For the project introduced in this paper, the number of file replications is fixed, and the files are stored in fragments.

Kubo [7] is a global, versioned, peer-to-peer filesystem. It combines good ideas from previous systems such as Git, BitTorrent, Kademlia, and SFS. It is implemented in Go [13]. Kubo provides robust distributed storage and retrieval capabilities but can be challenging to grasp for those new to distributed systems. Our pattern simplifies these concepts, enabling developers to implement distributed storage without delving into the intricate details of IPFS [14]. Moreover, the project introduced in this paper does not arrange all the nodes in a peer-to-peer architecture.

Ceph [3] is a highly reliable and scalable distributed storage system that offers object, block, and file storage in a unified system. While Ceph provides comprehensive features, it also comes with significant complexity. Our pattern offers a more accessible alternative for developers who need a lightweight solution for their distributed storage needs.

Kubernetes [2] has emerged as a powerful platform for automating the deployment, scaling, and management of containerized applications. It provides essential capabilities for managing containerized applications, including automated rollouts and rollbacks, service discovery, load balancing, and self-healing. Integrating our distributed storage pattern with Kubernetes presents an opportunity to leverage the orchestration capabilities of Kubernetes to provide robust and adaptable solutions. This combination can enhance the scalability and resilience of distributed storage systems, making it possible to manage storage resources dynamically and efficiently.

3. Pattern design principles

In this pattern of distributed storage system implementation the following principles are valid:

1. The data servers are divided into several clusters. A hard disk space limit can be set up before the data servers run. The file to be stored in this system is cut into fixed number of fragments or into fragments of fixed size so that the data servers can cooperate to store them. When we get or delete a file, the operation of the fragments should be concurrent for efficiency reasons. The `get` process merges the fragments into the original file using a method defined by the programmer.

2. The system has stream duplicate mechanism. Based on the file stream which is provided by the web server, the API server can store multiple streams with the same content, at the same time in each cluster of data servers. The system can detect a damage of file fragments of a cluster and it can recover the file according to the content of other clusters with complete file fragments.

3. The `put` process uses request and promise system. The data servers of one cluster share a RabbitMQ queue bound to a RabbitMQ exchange of type `direct`. The API server sends store requests of file fragments to clusters of data servers. Each request in the queue is handled by only one data server of a cluster. The requests are received by the data servers of a cluster in a balanced way.

After receiving a request, the data server estimates the free hard disk space to decide whether to give a promise to the related API server or to put this request back to the queue. The system should have a mechanism to handle the requests that have been put back to the queue too many times. Each promise contains a promise token and the address of this data server. The data server stores the promise made for a certain period of time. The API server that got a promise can use the token of it to make HTTP PUT request of the file fragment to the related data server. After the data server stored the file fragment, it discards the related promise. The communication is reduced or avoided (e.g.

the status report messages from the data servers) when there are no tasks to be done by the system; therefore, the system reacts on demand.

4. The `get` process uses cluster broadcast locating system. The data servers have their own private RabbitMQ queues for receiving the locate requests from the API server. In order to enable the data servers of a cluster to handle each locate requests concurrently, the queues are bound to a RabbitMQ exchange of type `direct` with the same routing key. The broadcast only happens in each cluster instead of all the data servers. Therefore, the system can also achieve workload balance between clusters during the `get` process.

4. Main features of the distributed storage

There are four types of servers in the pattern. The web server handles requests from the clients who use the distributed storage service provided by the project implementing this pattern. It uses the distributed storage service interfaces by making HTTP requests to API servers.

The API servers offer the interface of this distributed storage service. The data servers handle the file fragment store requests from the API servers and store the file fragments locally. The monitor server inspects the status of the data servers (and the API servers), records and solves some of the problems in the system. In the following, the main features of the new generic distributed storage pattern are given.

Controllable space usage. It can happen that the computer running a data server would not allow that all the hard disk space can be used for storage. Therefore, in order to regulate the space usage, a hard disk space limit must be set up before the data servers run. After a fragment store request is received, the data server checks the hard disk free space availability. If there is enough space for the fragment, the data server makes a promise to the API server that sends the request. After sending the promise, the related space is reserved before the HTTP PUT request of the fragment is received from the relevant API server.

Data safety. In this distributed storage pattern, there are several clusters of data servers. The file is stored in at least two distributed copies. The copies are stored in separate clusters of data servers. When the API server is reconstructing the file from the fragments, it may be found damaged in one of the clusters. In this case, the API server will try to reconstruct based on fragments stored at another cluster. If on the other cluster the file is not damaged, the API gets the reconstructed file and starts another goroutine to recover the file in the cluster with the damaged file. If the file is damaged in all the clusters, then the system cannot recover the file.

Cooperation in storage distribution. The data servers cooperate in a distributed manner to store a big file. They have hard disk space limits, and the file to be stored may be beyond their free spaces, i.e. the file can not be stored by a single data server. However, the file can be cut into fragments. Therefore, by this storage pattern, a file can be stored only if all the fragments can be saved in all of the free capacities offered by each cluster of data servers.

Load balance. In this pattern, a request and promise system is implemented based on a balanced queue consuming mechanism which is tested in the thesis [5]. In this request and promise system, the API server sends store requests to the data servers. All the data servers of one cluster share one RabbitMQ queue. The fragment store request sent to this cluster arrives in this queue.

The data servers of this cluster consume the fragment store requests from this queue in a balanced round-robin way because each data server has a consumer connected to this shared queue [9]. For example, suppose there are six fragments to be stored in a cluster. If there are two data servers connected in the cluster, then each data server stores three fragments. If there are three data servers connected in the cluster, each data server stores only two fragments.

Concurrent operations. The `get` and `delete` operations on the fragments are executed concurrently. Each operation is processed in a separate goroutine. The number of goroutines that run concurrently is constrained by a buffered Go channel. We only create a new goroutine when we can insert an element into that Go channel. Before the goroutine finishes, an element is removed from the buffered Go channel.

Lazy. This pattern makes the distributed storage system lazier by using a request and promise service rather than the heartbeat mechanism. It aims to reduce the messages' traffic, especially when there are no tasks for the system. "Heartbeat mechanism detects the states of nodes in cluster by periodically sending heartbeat message to other nodes and waiting for acknowledgement" [4].

The API servers do not need to maintain the information of all the data servers, and they do not need to select a data server in a balanced way to store the whole file. They just send the fragment store requests to the clusters of data servers and wait for promises until a timeout. Instead of the periodic status reports, the data servers report their status only on demand.

5. Implementation description

This section presents the project implementing this distributed storage pattern. In addition to the overview of the architecture, we also introduce the main distributed processes of this project. To make it easier to understand and make this paper more compact, the code listing only presents part of the

source code. For example, the error handling and the package import part are eliminated.

There are four types of servers in this project, as shown in Figure 1. The web server receives HTTP requests from the client and sends HTTP requests to the API server. The API servers send AMQP messages and HTTP requests to the data servers. The data servers reply to the API servers using the AMQP messages. The monitor server and the data servers communicate only using the AMQP messages.

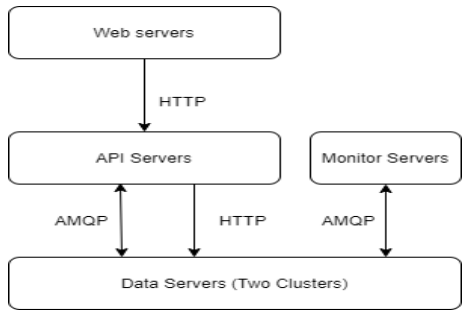


Figure 1. Distributed storage project architecture

There are two versions of distributed `put`, `get` and `delete` processes: the original versions (as in [5]) and the concurrent ones. The concurrent `put`, `get`, `delete` execute the operations of the file fragments concurrently. Since the concurrent `get` and `delete` processes showed improvement on the speed and the concurrent `put` showed identical performance (the measurements and their analysis are in Section 6), this paper presents the concurrent version of the distributed `get` and `delete`, and the original version of the distributed `put`.

5.1. Distributed `put` process

The file to be stored is in the HTTP request body. When a request comes, the request body can be read as a stream (because request body implements the interface `io.Reader`). There are two clusters of data servers. The API server duplicates the stream without memory usage. The two file streams are stored in the two clusters. Therefore, there are two distributed copies of each file in this system. The API server cuts each stream fragment by fragment. The fragment size or the number of fragments are fixed. For each fragment, the API server sends a RabbitMQ message to each cluster of data servers. All the data servers of one cluster share one queue to receive fragment store requests.

The data servers of a cluster receive the requests in a round-robin way. The data server received the request will check its hard disk free space availability. If the data server can not store the fragment, it re-queues the request. If the

data server can store the file, it sends back a promise which contains its address. Then the API server sends an HTTP PUT request of the fragment to this address. The API server also generates a JSON file of metadata and stores it in the same way as the fragments are stored. In the end, the API server replies to the web server with a file Id.

In the distributed `put` process, the incoming stream is cut into fragments. Each fragment is stored into two clusters of data servers, as shown in Figure 2. In this figure, there are only two data servers of each cluster and the stream is cut into two fragments.

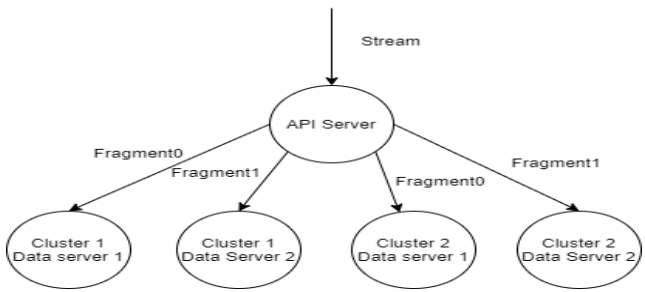


Figure 2. Duplicate stream

Now we introduce what the API server performs during a `put` process. The API server receives file store requests from the Web server and stores two copies of the file as fragments; therefore, we need to solve the duplication of the stream. During the `put` process, the API server does not load the whole file into the memory, it deals with the file streams.

The request body `r.Body` of the filestore request implemented the interface `io.Reader`, which means it can be read as a stream by using it method `Read`. This stream is duplicated and stored in both clusters. The stream duplication is done by the function `TeeReader` and `Pipe`, as Figure 3 shows.

Listing 1 shows how the API server duplicates the stream and stores them in two clusters of data servers during the `put` process. When the `TeeReader` with the name `tee` is being read, the content is also written to the second parameter `pipew`, which is a writer implementing the interface `io.Writer`. However, a reader implementing the interface `io.Reader` is needed when storing the file to both clusters. Therefore, the `io.Pipe()` is used to convert the writer to the reader. The reader `tee` and the reader `piper` should be read at the same time in different goroutines. The function `storeFile` stores the file stream in one of the clusters of data servers.

Listing 2 shows the `storeFile` function of the API server, which stores a file in a cluster. The RabbitMQ client is set up for distributed communication.

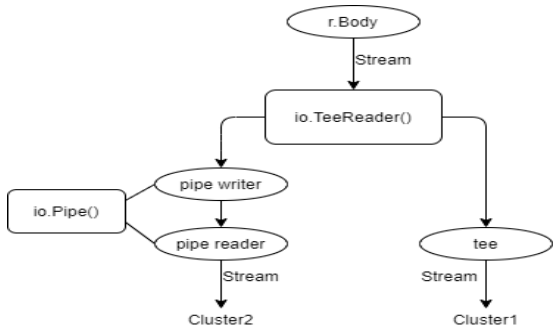


Figure 3. Duplicate stream implementation

The API server initializes the connection as `conn`, a channel for output as `cho`, and a channel for input as `chi`. The `storeRequestExchange` is declared for storing requests. A queue `q` is declared with a RabbitMQ server-generated name, which is automatically bound to the default `exchange` of type `direct` with its queue name as the routing key. This queue is used for receiving the promises from the data server and it is declared `exclusive` so that no other connections can access this queue. The consumer consumes from this queue also exclusively so that no other consumers can consume from this queue.

The API Server reads the stream fragment by fragment until the end of the file error `io.EOF`. The fragment size is set globally in the API Server. For each fragment, the API server sends a store request to a cluster of data servers by using the function `storeRequestSend`. After sending the store request, the API server waits data servers of the cluster for a promise message, which means this data server can store this fragment. The promise message includes the data server address and the promise token. The `fragmentId` is created by attaching the `fileId` to the `fragmentIndex`. Then the function `storeFragement` stores the fragment in this address.

Figure 4 depicts how the stream is stored in the `cluster1`. It includes the store request sending, the promise receiving, and the `put` request sending. In this figure, the stream is only cut into two fragments and the `cluster1` contains only two data servers. The fragment store requests are sent to the shared queue of `cluster1` where they are consumed by each data server in a balanced way. The store requests have to go through a `storeRequestExchange` before reaching the shared queue. This exchange is not shown in the diagram for simplicity.

However, when there is no queue bound to the `storeRequest-Exchange` that matches the routing key, the message sent by the API server is lost. In that case, there is no promise received by the API server. Thus, there is a timeout when receiving the promise (line 14).

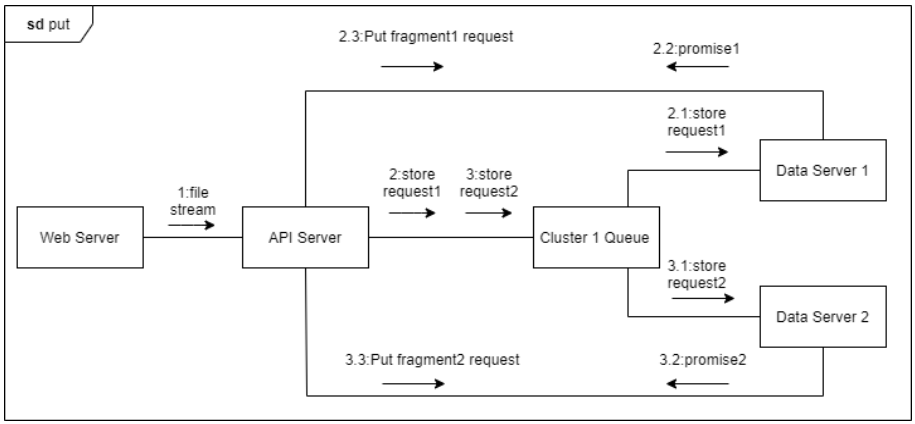


Figure 4. Distributed `put` process

As defined for the `io.Pipe`, the write operation to the pipe is blocked until reading on the pipe. There is no internal buffering. Therefore, the call of the function `storeFragment` with argument `piper` in the line 12 is synchronized with the call of the `io.CopyN` with argument `pipew` in the line 17. In the child goroutine, the store request sending and the promise receiving is made before this synchronization point. Therefore, at the very same time, there is only one request sent and waiting for the promise. There is no matching problem between store requests and responses. So, there is no need to declare a correlation id to pair the requests and responses.

If there is an error in the goroutine storing the fragment, the pipe reader `piper` is closed. It causes error in the line 17 at the pipe writer `pipew`. Then this function `storeFile` returns and the function `putHandler` returns the `http.StatusInternalServerError` error to the client. After all the fragments are stored, the metadata is stored in the same cluster. If the function `StoreFile` has stored all the fragments successfully, then it returns how many fragments were stored. The error value it returns is nil (line 31).

In this pattern, the metadata is stored locally rather than using a third-party distributed database. The metadata is a JSON file that contains the information of a file, and it is created when storing the file in one of the data servers of a cluster. The fragments of the file are stored with different generated fragment id. The fragment id is not related to the original file name. The file name and the maximum index of the fragments are stored in the metadata of the file. Those are used when the fragments need to be merged into the original file.

The function `storeRequestSend` sends to a cluster a RabbitMQ message that has a republish count header with initial value 0. The function `storeFragment` sends a HTTP PUT request to the data server to store the fragment. The `fragmentId` and `promiseToken` are included in the URL.

Now the data server operations during a `put` process are introduced. The global variable `promisedSpaceB` is the bytes count that the data server has promised to the API servers. The `promiseTable` maps the promise token to the `promiseData`. It records all the promises that this data server has made. The `promiseData` is a structure with two fields. The field `size` is the byte count that this promise has made. The field `createTime` is recording the time when this promise is created. The `mutex` is used to lock and unlock the resource shared by multiple goroutines.

Listing 3. shows the data server starting a goroutine with the function `startHandleStore-Request` to handle the store requests. It also starts another goroutine running the function `startRemove-ExpiredPromise` to remove the expired promises periodically.

Listing 4. contains the function `startHandleStoreRequest` of the data server. The data servers share one queue of store requests, and they consume the requests in a balanced way. Each store request should be handled by one of the data servers. After the data server consumed the store request, it decides to handle it or re-queue to the shared queue according to the local free space status.

Beside the RabbitMQ setting, the queue `q` is declared with the given cluster name. This queue is bound to the `storeRequestExchange` with the queue name as the routing key. All the data servers of one cluster share and consume from the same queue. In this case, all the consumers of the same queue consume the messages in a balanced, round-robin way (line 2-3).

After it has received a store request, the data server first checks if this message reached the republish count limit. If it reaches the limit, a loop message `loopMsg` is sent to the monitor server. After checking the republish count, the data server checks if it has enough space for storage. If there is enough space, then the promised bytes are atomically added to the local variable `promisedSpaceB`. The `promiseToken` is created by appending a token suffix to a generated random string. One more record is added to the `promiseTable` in mutually exclusive way. The data server sends a reply with its TCP network address to the store request sender. If there is not enough space, then the data server republishes this request for other data servers of its cluster.

When the data server checks if the storage space is enough by the function `isSpace-Enough`, both the stored bytes and the promised bytes are considered. The function `putHandler` is called to handle the request when there are HTTP PUT requests coming from the API server. The function `putHandler` gets the `promiseToken` from the URL. It checks if this `promiseToken` exists in the `promiseTable`. If this token does not exist in the table, then this function returns and shows the status code `StatusForbidden`. After checking the `promiseToken`, this function creates the file. All the contents in the request body are copied to the file. If the token exists in the map, this record is deleted

from the map `promiseTable`. The stored number of bytes is removed from the `promisedSpaceB`. In the end, the file is stored locally.

The monitor server has a goroutine running the function `startHandleLoopMsg`. It receives a message that have been re-queued too many times sent from the data server and it stores locally.

The `put` process cuts the file stream into fragments of fixed size set in the global variable of the API server. The fragment size is the same for all different files (except the last fragment). It is more convenient to manage the fragments of the same size. The fragment count for each file may be different. In this case, it is not easy to find a proper fragment size for this system. If the size of the fragment is set too small, there are too many fragments for a file and too much communication between the API server and the data server to put, get or delete a file that is distributedly stored.

In the `put` process introduced before, the file is stored in fragments of fixed size. We need to find a proper fragment size for the system. Hereinafter, another strategy for the distributed `put` process is introduced, the file is stored in fixed number of fragments. It provides a comparison to the previous `put` process. However, it still needs to find a proper fixed number for the system.

Listing 5 presents how the function `putHandler2` gets the `fileName` and the `fileSize` from the URL. In the function `storeFile2`, all the fragment sizes are calculated by the passed argument `fileSize` and the global constant `fixedFragmentCount`. The global constant `fixedFragmentCount` is the number of iterations of the for loop storing the fragments.

5.2. Distributed `get` process

The `get` request sent to the API servers contains the file Id. The API server locates and gets the metadata of this file first. Then randomly picks up a cluster to get the file. The API server locates all the fragments according to the metadata. Then the API server sends the `get` requests to the located data servers for all the fragments. Each request receives a response which contains a `reader` implementing the interface `io.Reader` from which we can read the fragment in a stream. The API server merges those readers of fragment in order into a single reader of the file. In the end, the API server copies all the contents in this reader to the client.

In the distributed `get` process all the fragments of a stream are retrieved from one of the clusters of data servers, as shown in Figure 5, which depicts only two data servers of each cluster, and the stream has two fragments.

Now we introduce what the API server performs during a `get` (see Listing 6). In the function `concGetHandler`, the `fileId` is included in the request URL. A cluster is randomly chosen to get the file. The function `concGetReaderAndMeta` is used to get a merged reader that contains the whole file and the metadata of

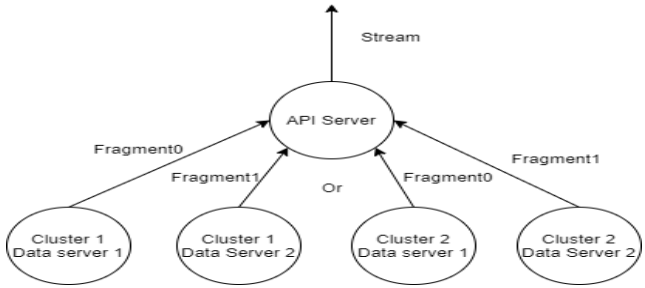


Figure 5. Distributed `get` process

it. If there is an error, another cluster is chosen and a goroutine running the function `recoverFile` is started to deal with the recovery redirection from one cluster to another cluster. The function `anotherCluster` gets another cluster's name. If an error occurs when trying to get the merged file in both clusters, this function writes a header `StatusNotFound` as response and it returns. By setting the response header `Content-Disposition` to `attachment` with the file name, the response is downloaded as a file by the browser. In the end, the file is copied from the merged reader to the response writer.

The function `recoverFile` gets the reader from the cluster with the complete file. The file in the reader is stored in the cluster with the damaged file. The function `concGetReaderAndMeta` can get all the fragments from the data servers of one of the two clusters and it returns a merged reader.

Figure 6 shows the process getting the file from `cluster1`; it contains the fragment locating, getting, and merging. In this example, the file stream contains only two fragments, and the `cluster1` has only two data servers. This process start when the API server sends locate requests concurrently to the `locateExchange`. Every data server with a matching routing key gets the message from the exchange concurrently. In this example, the data server 1 has the fragment 1, so it replies its address to the API server. The API server has separate queues to locate each fragment, the routing key of the queue is included in the locate request. Therefore, the data server knows where it should reply the result, if it has the related fragment.

Listing 7 introduces the function `concGetReaderAndMeta` of the API server. The `maxIndex` in the metadata of the file is received by `getMetaAndAddress`, where in a for loop all the fragments' readers which implement the interface `io.Reader` are retrieved concurrently. The `MultiReader` merges all those readers of fragments into a single reader of the complete file.

The `WaitGroup` causes `concGetReaderAndMeta` to wait for all the goroutines that get the fragment. Before the goroutine is created, one is added to the `WaitGroup`, and it is removed by the function `Done` after the goroutine's job is ended.

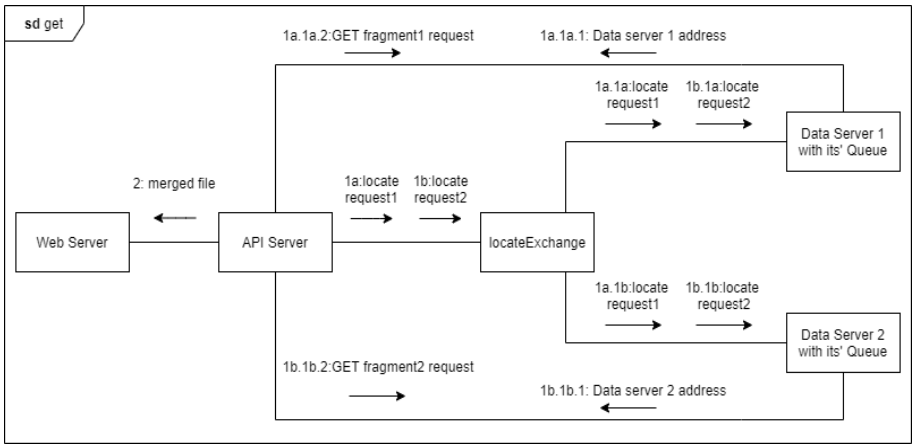


Figure 6. Distributed `get` process

The `limit` is used to limit the number of goroutines that run at the same time. It is a Go channel of integers with size of `maxGoroutine`. Before creating the goroutine, the parent goroutine tries to insert a integer one to the `limit`. If the `limit` is already full, the insertion is blocked. It waits until a integer one is removed by another goroutine. The goroutine removes a integer one from the `limit` after the job is done.

The `readerMap` is used to collect the fragment's reader. The fragments in the `readerMap` are not ordered. The `keys` are ordered and are used together with the `readerMap` to get an ordered fragment slice `readerList`. The `readerList` is used to get a merged file reader by using the function `MultiReader`. The `readerMap` needs to be locked when the child goroutine tries to modify it.

The function `getMetaAndAddress` locates, gets and unmarshals the metadata of a file, it returns the metafile address and the metafile itself.

In the `locate` function, an exchange `locateExchange` of type `direct` is declared. A queue `q` with generated name is bound to the default exchange in RabbitMQ server by default, which is used for receiving a reply from the data server. The id is sent to the exchange `locateExchange` with `clusterName` as the routing key. All the data servers of this cluster receive a message locating the fragment with this id. In the end, the API server waits for a reply from a data server with a timeout limit.

In the function `getHandler`, the file name is retrieved from the URL. The function `Open` of the package `os` is used to open the file locally and the file content is copied to the response.

5.3. Distributed `delete` process

The distributed `delete` requests from the API server contains the file Id.

The API servers locate and delete all the fragments and metadata in both of the two clusters concurrently. The API server deletes the fragments by sending HTTP DELETE requests to the data servers. When the data server of one of the clusters received a `delete` request from the API server, it deletes the fragment which is stored locally.

The `WaitGroup` and the `limit` are used for the same purpose as the distributed `get` process introduced in Subsection 5.2. Each fragment is deleted in a child goroutine.

The function `deleteFragment` makes the HTTP DELETE request to a TCP address with a timeout limit.

Whenever a `delete` request comes to the data server, the function `deleteHandler` in the data server is called to handle the request. The function `deleteHandler` uses the `Remove` function in the package `os` to remove the file locally.

5.4. Monitor service

The monitor server checks the status of the data servers. It can list the free space for each data server of a cluster. The monitor server also handles the store request message when it is re-queued by the data servers too many times. It also gets all the free space left of all the data servers of a cluster, and a web user interface shows the result. For the web interface, we have used the Go package `html/template`.

The function `monitorHandler` gets the cluster name from the URL. This function calls the function `list-FreeSpace` to get a table of the results, which is passed by parameter when executing the template. The function `listFreeSpace` sends a message to all the data servers of a cluster and waits for the free space reports (see Listing 8). In the function `listFreeSpace`, an exchange `freeSpaceExchange` of type `direct` is declared. A queue `q` of generated name is declared to receive the reply message from the data servers. The queue name is used as `ReplyTo` when sending a message to the `freeSpaceExchange`. After sending the message, this function waits for free space reports from the data servers of a cluster until there is no report coming within a timeout. The report is put on a map that maps the address to free space and it is returned at the end.

In Listing 9, the data servers handle the command messages received from the monitor server which ask them to list their free hard disk spaces. In the function `startHandleListFreeSpace`, a queue `q` with generated name is declared `exclusive`. This queue is bound to the `freeSpaceExchange` with the

cluster name as the routing key. This queue is used to receive the message from the monitor server. For each message received, the data server creates a free space report, which includes the address and the free space left in this data server. This report is encoded and sent to the default exchange with the `ReplyTo` as the routing key. This message is sent to the monitor server.

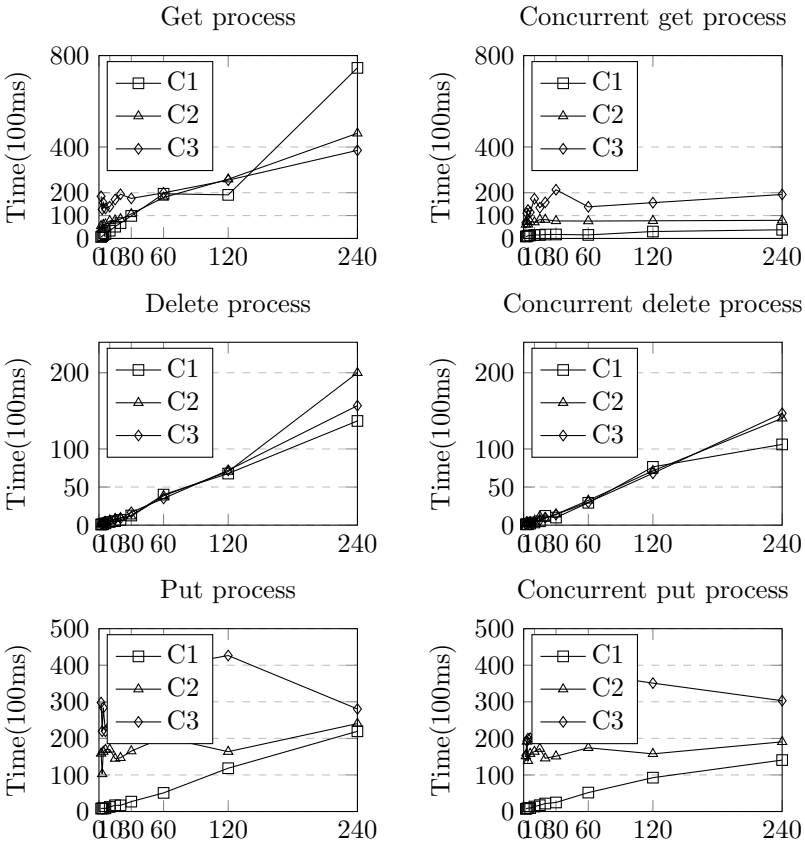
6. Measurement results

This test is done to compare the original version and the concurrent version of the distributed processes.

Three computers are involved. The server computer used in the test has the following configuration: processor, Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99GHz. RAM, 16G. Cores, 4. Logical processors,8. Operating System, Windows 10 professional. The first slave computer used in the tests has the following configuration: processor, Intel(R)Celeron(R) CPU N2930 @ 1.82GHz 1.83GHz. RAM, 4G. Cores, 4. Logical processors,4. Operating System, Windows 7 Enterprise. The second slave computer used in the tests has the following configuration: processor, Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz 2.20GHz. RAM, 4G. Cores, 2. Logical processors,4. Operating System, Windows 10 Chinese family version. The server computer runs the following five servers: API server, monitor server, web server, data server of cluster1, and data server of cluster2. The slave computers run two data servers: one data server for each cluster.

In the diagrams, `C1` means one computer: only the server runs. `C2` means two computers are running: the server and a slave. `C3` means three computers are involved: the server and two slaves. The horizontal axes of the pictures represent the number of fragments a file decomposed, while the vertical axes depict the running times in 100ms.

It is observed in the tests that the concurrent versions of distributed `get` and `delete` processes are faster. Therefore, they can get efficiency benefits from the concurrency. The concurrent version of the `put` process does not show a more significant improvement than the original version. There are two reasons for these test results. The first reason is that the original version already takes benefits from the concurrency when storing concurrently the original file stream and the duplicated file stream in different clusters. The second reason is that the operations on the fragments cannot get benefits from concurrency. Since this pattern neither loads the file to be stored into the memory nor stores it temporarily somewhere else, the file stream can only be dealt with sequentially. In other words, in this pattern the fragments of a file stream can not be stored concurrently.



7. Conclusion

The created pattern gets benefits from the distribution, due to which the data servers with space control can cooperate in a distributed and balanced way to store a large file that cannot be stored by one single data server. The stored file is safer because the file is cut into fragments, and more copies of the fragments are stored in a distributed manner in separate clusters of data servers which can be used by recovery services. It is observed in the tests that the distributed `get` process and the distributed `delete` process can obtain efficiency benefits from the concurrency. In the distributed `put` process, the duplicated file stream and the original file stream are stored in both clusters concurrently.

Therefore, this pattern takes benefits both from the distribution and the concurrency and provides a guideline for the software architect who desires to implement a distributed system by using the Go with AMQP.

References

[1] AMQP 0-9-1 Model Explained, https://www.rabbitmq.com/tutorials/amqp-concepts.html Last Accessed on 23. Apr. 2024.

[2] **Khatami, A.A., Y. Purwanto and M.F. Ruriawan,** High availability storage server with kubernetes, in: *2020 International Conference on Information Technology Systems and Innovation* (*ICITSI*), Bandung, Indonesia, 2020, pp. 74–78.

[3] **Weil, S., S.A. Brandt, E.L. Miller, D.D. Long and C. Maltzahn,** Ceph: A scalable, high-performance distributed file system, in: *Proceedings of the 7th Conference on Operating Systems Design and Implementation* (*OSDI'06*), 2006, pp. 307–320.

[4] **Li, F., X. Yu and G. Wu,** Design and implementation of high availability distributed system based on multi-level heartbeat protocol, 2009 IITA, Zhangjiajie, 2009, IEEE, pp. 83–87.

[5] **Li, Jianhao,** *Message Queue based Generic Pattern for Distributed Storage*, Master Thesis, Eötvös Loránd University, 2020.

[6] Go, https://golang.org. Last Accessed on 23. Apr. 2024.

[7] Kubo: IPFS Implementation in GO, https://github.com/ipfs/kubo. Last Accessed on 23. Apr. 2024.

[8] RabbitMQ, https://www.rabbitmq.com. Last Accessed on 23. Apr. 2024.

[9] RabbitMQ Consumers, https://www.rabbitmq.com/docs/consumers. Last Accessed on 23. Apr. 2024.

[10] Seaweedfs, https://github.com/chrislusf/seaweedfs. Last Accessed on 23. Apr. 2024.

[11] **Lackschewitz, N.M., S. Krey, H. Nolte, S. Christgau, S. Oeste and J. Kunkel,** Performance Evaluation of Object Storages (NHR2022).

[12] **Ionescu, V.M.,** The analysis of the performance of RabbitMQ and ActiveMQ, In: *14th RoEduNet International Conference-Networking in Education and Research* (*RoEduNet NER*), 2015 September, pp. 132–137. IEEE.

[13] **Tsoukalos, M.,** *Mastering Go: Create Golang Production Applications Using Network Libraries, Concurrency, Machine Learning, and Advanced Data Structures*, Packt Publishing Ltd., 2019.

[14] **Daniel, E. and F. Tschorsch,** IPFS and friends: A qualitative comparison of next generation peer-to-peer data networks, *IEEE Communications Surveys & Tutorials*, **24(1)** (2022), 31–52.

A. Code listings

```
func putHandler(w http.ResponseWriter, r *http.Request) {
    piper,pipew:=io.Pipe(); tee :=io.TeeReader(r.Body, pipew)
    c:=make(chan error); c1:=make(chan uint64)
    go func(){ stored, err := storeFile(tee, fileId, fileName,
    clusterOneName)
                pipew.Close(); c<-err; c1<-stored }()
    stored2, err2 := storeFile(piper, fileId,fileName, clusterTwoName)
    err1:=<-c; stored1:= <-c1 }
```

Listing 1. Function putHandler of API server

```
func storeFile(r io.Reader, fileId string,
fileName string, cluster string)  (storedFragmentNumber uint64, e error){
    var index uint64 = 0
    for {
        piper, pipew := io.Pipe()
        go func(id uint64){
            err = storeRequestSend(cho, cluster, q.Name, fragmentSize)
            select {
            case msg:= <-msgs:
                promiseToken := msg.CorrelationId;
                dsAddress := string(msg.Body)
                fragmentId := fileId+"_"+ strconv.FormatUint(id,10)
                err:= storeFragment(dsAddress,
                promiseToken, piper, fragmentId)
                msg.Ack(false)
            case <-time.After(promiseReceiveTimeOut):
                piper.Close(); return
            }}(index)
        _,err:=io.CopyN(pipew, r, fragmentSize); pipew.Close(); index++ }
    metaId := fileId+"_meta"; dateNow := time.Now().Format(dateLayout)
    metaData:= Meta{ FileId:fileId, FileName:fileName,
    CreateTime:dateNow, MaxFragmentIndex:index}
    metaDataJson, err := json.MarshalIndent(metaData, "", "␣")
    metaDateReader :=bytes.NewReader(metaDataJson);
    metaBytes :=len(metaDataJson)
    err = storeRequestSend(cho,cluster, q.Name, int64(metaBytes))
    select {
    case msg := <-msgs:
        promiseToken := msg.CorrelationId; dsAddress := string(msg.Body)
        err= storeFragment(dsAddress, promiseToken, metaDateReader, metaId)
        msg.Ack(false)
    case <-time.After(promiseReceiveTimeOut):
        return index+1, fmt.Errorf("timeout") }
    return index+1,nil }
```

Listing 2. Function storeFile of API server

```
type promiseData struct {
        size uint64
        createTime time.Time }
var mutex sync.Mutex
var promisedSpaceB uint64
var promiseTable = make(map[string]promiseData)
func main() { go startHandleStoreRequest(); go startRemoveExpiredPromise() }
```

Listing 3. Function main of Data server

```
func startHandleStoreRequest(){
    q, err := chi.QueueDeclare(clusterName, false, true, false, false, nil)
    err = chi.QueueBind(q.Name,q.Name,"storeRequestExchange", false, nil)
    msgs, err := chi.Consume(q.Name, "", false, false, false, false, nil)
    for msg := range msgs {
        republishCount, ok := msg.Headers["republish-count"].(int32)
        if !ok || republishCount> republishCountLimit{
            dateNow := time.Now().Format(dateLayout)
            loopMsg:= LoopMsg{Exchange:msg.Exchange,
            Key:msg.RoutingKey, ReplyTo:msg.ReplyTo,
            Body:msg.Body, DetectedTime:dateNow}
            loopMsgJson, err := json.MarshalIndent(loopMsg, "", "␣")
            body := loopMsgJson
            err = cho.Publish("loopMsgExchange","loopMsgQueue",
            false,false,amqp.Publishing{ContentType: "text/plain",
            Body:body})
            fmt.Printf("Sent␣%s", body); msg.Nack(false,false) ;continue}
        requestSpace, err:= strconv.ParseUint(string(msg.Body), 10, 64)
        if isSpaceEnough(requestSpace){
            promiseToken := randomString()+"_token"
            atomic.AddUint64(&promisedSpaceB, requestSpace); mutex.Lock()
            promiseTable[promiseToken] = promiseData{requestSpace,
            time.Now()}
            mutex.Unlock(); body := tcpNetworkAddress
            err = cho.Publish("",msg.ReplyTo,false,false,
                amqp.Publishing{ ContentType: "text/plain",
                Body: []byte(body)})
            fmt.Printf("Sent␣%s\n", body)
            msg.Ack(false)
        } else {msg.Headers["republish-count"] = republishCount + 1
              err = cho.Publish(msg.Exchange,msg.RoutingKey, false, false,
                amqp.Publishing{ContentType: "text/plain",
                Body:msg.Body,Headers:msg.Headers})
                fmt.Println("Republish␣success"); msg.Ack(false) } } }
```

Listing 4. Function start-HandleStoreRequest of data server

```
func storeFile2(r io.Reader, fileId string, fileName string,
cluster string, fileSize int64)  (storedFragmentNumber uint64,e error){
    preFragmentSize := fileSize/fixedFragmentCount
    lastFragmentSize := preFragmentSize+ (fileSize%fixedFragmentCount)
    for i:=uint64(0); i<fixedFragmentCount; i++{
        var bytesToStore int64
        if i != fixedFragmentCount-1{ bytesToStore = preFragmentSize
        } else{ bytesToStore = lastFragmentSize } } }
```

Listing 5. Function storeFile2 of API server

```
func concGetHandler(w http.ResponseWriter, r *http.Request) {
    fileId := strings.Split(r.URL.EscapedPath(), "/")[2]
    randomInt := rand.Intn(2)+1;
    clusterName := "cluster"+strconv.Itoa(randomInt)
    reader,meta, err := concGetReaderAndMeta(fileId, clusterName)
    if err != nil {
        anotherCluster :=  anotherCluster(randomInt)
        reader, _,err = concGetReaderAndMeta(fileId, anotherCluster)
        if err !=nil{ w.WriteHeader(http.StatusNotFound); return }
        go recoverFile(fileId, anotherCluster, clusterName) }
    w.Header().Set("Content-Disposition",
        "attachment;␣filename="+meta.FileName)
    _,err=io.Copy(w, reader) }
```

Listing 6. Function concGetHandler of API server

```
func concGetReaderAndMeta(fileId string,
clusterName string) (io.Reader, Meta, error) {
    _,maxIndex, err :=getMetaAndAddress(fileId, clusterName)
    maxIndex := meta.MaxFragmentIndex; var mutex sync.Mutex
        readerMap := make(map[uint64]io.Reader)
    var readerList []io.Reader; var index uint64 =0; var wg sync.WaitGroup
        limit := make(chan int, maxGoroutine); var keys []uint64
    for index <= maxIndex{
        limit <- 1 ;  wg.Add(1)
        go func(i uint64){
            defer wg.Done()
            url := "http://" + address + "/get/" + fragmentId
            client := http.Client{Timeout:httpRequestTimeOut}
            fragmentResponse, err := client.Get(url)
            mutex.Lock(); readerMap[i]= fragmentResponse.Body
            mutex.Unlock()
            <- limit }(index)
        keys = append(keys,index); index ++ }
    wg.Wait()
    for _,k := range keys{
        readerList=append(readerList,readerMap[k]) }
    reader := io.MultiReader(readerList...) }
```

Listing 7. Function concGetReaderAndMeta of API server

```
func listFreeSpace(cluster string) (map[string]uint64, error){
    err = chi.ExchangeDeclare("freeSpaceExchange","direct",
    false,true,false,false,nil)
    q, err := chi.QueueDeclare("",false,true,true,false,nil)
    msgs, err := chi.Consume(q.Name, "", false, true, false, false, nil)
    body := "listFreeSpace"
    err = cho.Publish("freeSpaceExchange",cluster,false,false,
        amqp.Publishing{ ContentType: "text/plain", ReplyTo: q.Name,
            Body: []byte(body),})
    fmt.Printf("Sent␣%s", body)
    result := make(map[string]uint64)
    L: for{
        select {
        case msg := <-msgs:
            freeSpaceReport:=FreeSpaceReport{}
            err =json.Unmarshal(msg.Body, &freeSpaceReport)
            result[freeSpaceReport.Address]= freeSpaceReport.FreeSpace
            msg.Ack(false)
        case <-time.After(reportReceiveTimeOut):
            fmt.Println("Timeout")
            break L } }
    return result,nil }
```

Listing 8. Function listFreeSpace of monitor server

```
func startHandleListFreeSpace(){
    q, err := chi.QueueDeclare("",false,true,true,false,nil)
    err = chi.QueueBind(q.Name,clusterName,"freeSpaceExchange",false,nil)
    msgs, err := chi.Consume(q.Name,"",false,true,false,false,nil)
    for msg := range msgs {
        storedB, err :=  dirSize(storageFolder)
        spaceLeft :=spaceLimitB-storedB-promisedSpaceB
        freeSpaceReport:= FreeSpaceReport{Address:tcpNetworkAddress,
```

```
        FreeSpace:spaceLeft}                                                    9
        reportJson, err := json.MarshalIndent(freeSpaceReport, "", "␣")        10
        body := reportJson                                                      11
        err = cho.Publish("",msg.ReplyTo,false,false,                           12
            amqp.Publishing{ ContentType: "text/plain", Body: body})            13
        fmt.Printf("Sent␣%s", body)  } }                                        14
```

Listing 9. Function startHandleListFreeSpace of Data server

J. Li and V. Zsók
Department of Programming Langauges and Compilers
Faculty of Informatics
Eötvös Loránd University
lijianhao@inf.elte.hu
zsv@inf.elte.hu