2021-11-02

Distributed GFS Meta

David Kleingeld

Email
opensource@davidsk.dev

Contents

1 Introduction

In the last decade we have seen the rise of big data and the shift of everyday work to the cloud. With it rose the need for a highly available and scaling file system. The problem solved by distributed file systems is how to span, spread and keep consistent files across multiple machines. Generally there are two approaches, a dedicated node to control where each file is placed[1] or have a

[1] Inspired by GoogleFs[4] used by Hadoop file-system[7] (HDFS) and MooseFs[3]

distribution function decide where a file should be located[2]. The distribution function does not need access to any shared state, aiding scaling.

A key problem with the first approaches is the dedicated metadata node, as single point of failure. Implementations such as HDFS have expanded, adding standby nodes that can replace the metadata node. This requires either access to shared storage[1], moving the point of failure, or a separate cluster of journal-nodes[2]. - Here I made a prototype implementing a subset of GoogleFs using the raft[5] consensus algorithm to replace the one meta-data node with a failure proof cluster. The cluster has one node responsible for meta-data modifications, the other cluster members actively host consistent read-only meta-data.

In section 2 I will detail my implementation, then in section 3 we test the system. Finally, I discuss how well this approach works in section 4. Instructions on how to deploy the system for testing are in appendix A.

2 Implementation

Here I will discuss my implementation in three parts. I will begin by going over the states a cluster node goes through during its operation. Then I will show how the meta-data is changed, following a mkdir request through the cluster. Finally, I detail the technical side discussing the use of async over classical concurrency primitives. Some systems I will not discuss in depth, these are:

1. The client implementation. It automatically finds the right node to talk to and retries any request until accepted by the cluster. It will ask the cluster for new addresses if it notices a node going offline. If the cluster does not move too quickly it will stay operating even if the cluster moves to all new IP addresses.
2. The automatic cluster configuration. The cluster nodes do not need to know each other address. The nodes will use UDP multicasts to build an address book and keep it up to date while operating. Nodes can be added[3] while the cluster operates.

[2]Pioneered by Ceph[8] and also used by GlusterFs[6]
[3]as long as the cluster size does not grow beyond a statically set maximum

2.1 The life of a node

A node in the cluster can go through three states in its life: *disconnected*, *readserver* and *writeserver*. We can see the stages the node goes through in fig. 1. Every node starts *disconnected*. In this state its discovering cluster members and waiting till it has discovered at least 50% of the (maximum) cluster size.

When it has found enough cluster members our node becomes a *readserver*, it can then start serving metadata for read requests [4]. It will serve those requests as long as it's not *outdated*. A *readserver* starts outdated and can get up to date by syncing with an elected *writeserver*. At any time if no *writeserver* is found an election will start using the *raft*[5] consensus algorithm.

If our node wins the election it will become the current *writeserver*. As *writeserver* it will handle client requests modifying the metadata and maintain a heartbeat. The heartbeat is part of *raft* and ensures no new elections start.

2.2 Modifying metadata

Here I will describe what happens during a successful make directory request (mkdir) where some servers malfunctioned. The transaction is illustrated in fig. 2. It starts with the client sending a mkdir request over TCP. Servers and clients communicate between each other over TCP. Requests and responses consist of serialized[5] high level types.

Once the write server receives the request it will cancel the next heartbeat and instead publish the change to the cluster. To speed this up the writeserver caches connections to the read servers. It is no problem if a cluster member does not receive a request as:

1. If the member receives the next heartbeat it will notice it is outdated and stop hosting meta-data.
2. If the member fails to receive the next two heartbeats it will declare itself outdated and stop hosting meta-data.

[4] such as opening a file in read only mode or listing everything in a directory

[5] As serialization format I use Bincode which guarantees an encoded size no larger than the deserialized size

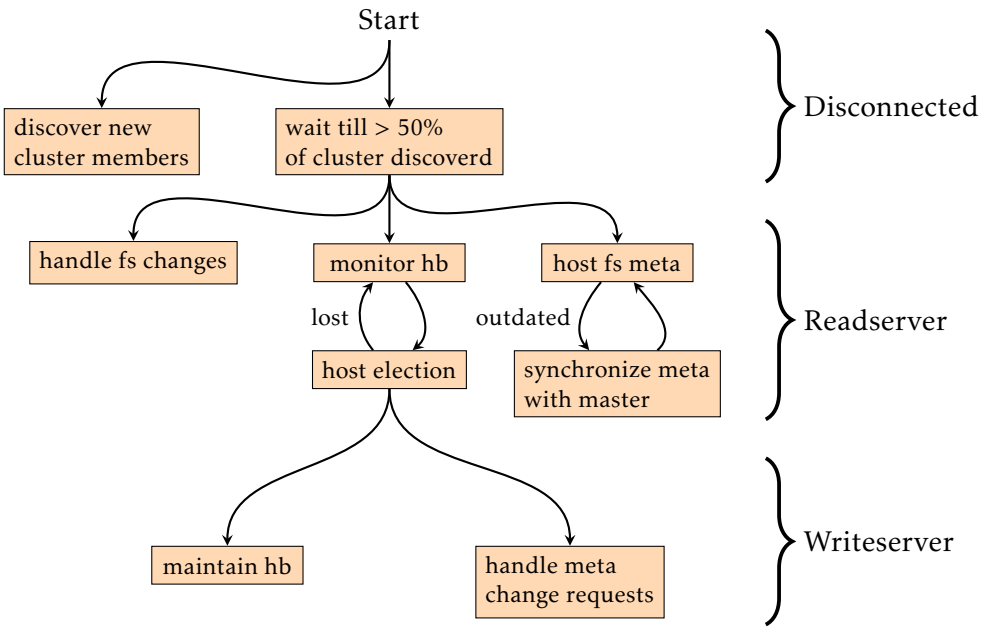

Figure 1: *The states a server node goes through (left of the braces) and the various processes it does*

After declaring itself outdated a server will ask and get from the write server an up-to-date copy of the file system directory.

Any read server that is not malfunctioning now receives the change and checks if it can safely apply it using the *Raft* consensus algorithm. Then it applies the change to its database and answers the writeserver all is well.

After publishing a change the writeserver can run into three scenarios:

1. *Everyone acknowledged*: continue
2. *A majority acknowledged*: some server(s) failed to answer, the new change will still be available to all clients two heartbeat durations after the writeserver published. By then any malfunctioning server(s) will have noticed it/they are outdated.
3. *A minority acknowledged*: the writeserver is no longer in control it will crash, and the cluster will go through an election.

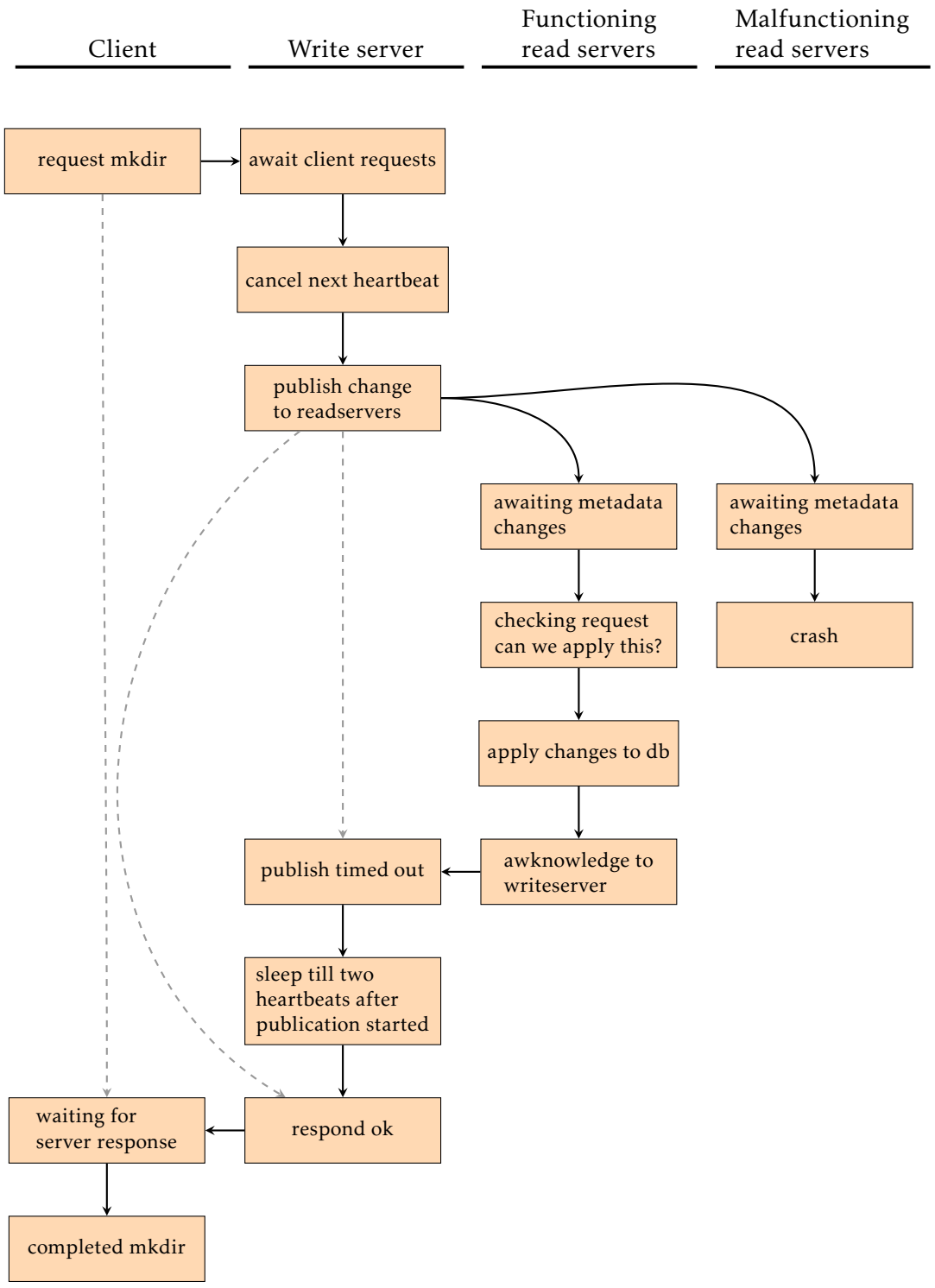

Figure 2: *The states the client and cluster go through during a mkdir request*

Now the request completes with the writeserver answers the client.

2.3 Technicalities

The implementation has been written in *Rust*: a high performance memory safe language without garbage collection. It makes extensive use of libraries for networking, the database and logging/tracing. The system is split into a binary for the server software, a libary defining the communcation protocol and a client libary for interacting with the server. The client libary provides various example binaries.

The implementation is made without any blocking code using only asynchronous (async) functions instead. The result is a fully concurrent system where most work runs in parallel. I have written a short primer on async in the past here included in appendix B though a far better resource is the *rust async book*[6]. In general a machine can handle many more asynchronous tasks than threads. Async is a good fit here as consistency requires us to wait before answering and finishing many tasks.

Debugging distributed systems is notoriously hard. To help development this implementation uses 'tracing', a framework for structured logging. The information is sent to a (potentially distributed) tracing collection system which provides a web interface to inspect the structured logs of all the nodes in a central location.

3 Results

To quantify the stability and availability of the system I presented it with a number of scenarios in various tests. I will only report the setup and whether a test succeeded for the simpler scenarios. For the more challenging test I will present a more detailed look at system performance which will allow us to make conclusions in the next section.

The most basic test checks the file system functions and is consistent with these steps:

1. Use the *mkdir* request to build a small flat directory tree of 10 directories on the cluster.
2. Querying the content of the cluster using the *ls* request.

[6] freely available at: https://rust-lang.github.io/async-book

3. Check these answer matches with the mkdir requests from step one.
4. Remove the directories using the *rm* request.
5. Again query the content of the cluster.
6. Check the answer to verify the cluster is now empty.

All the request are send from a single client. The writeserver will redirect any request for the clusters content to a random readserver. This means if the test succeeds we know the changes are propagated to at least one readserver. The test ran without problems for 10 runs giving a high chance the probability propagate to all readservers.

For the second test I focussed on the availability of the cluster. While the cluster was running without processing any client request the writeserver was killed using SIGKILL before running the first test. If the first test still completes a new master must have been elected and accepted. This test succeeded without problems.

As third test I focussed on the stability and performance of the system under trivial load. A single machine send out 300 *mkdir* requests sequentially. These tests failed every time. From the server logs I can plot the time left till the writeservers heartbeat times out, see fig. 3.

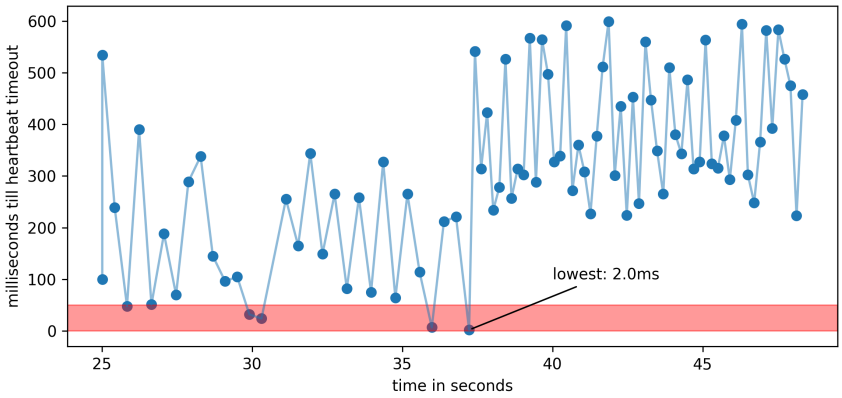

Figure 3: *Time left before the heartbeat times out while receiving sequential make directory requests. Around 37 seconds the writeserver timed out and a new writeserver took over because another readserver timed out. The client errored and stopped sending requests. Unencumbered by the mkdir requests the new writeservers heartbeat arrive well on time.*

Finally we test 'read' performance using the list directory request. I used a single client to open between 2 and 4096 simultanious connections to the cluster. Over each connection a list directory request was send 100 times. The total time between start and finish was measured for various combinations of connections and cluster size. These times are converted to milliseconds per request. The results can be seen in fig. 4 and fig. 5. It was difficult to gather the data as larger cluster sizes often ran into heartbeat issues. Except for the run with 6 nodes the data presented here is selected from successful tests. To get this data the tests had to be run many times. Note in fig. 4 as the number of requests increases the average request time goes down logarithmically. In fig. 5 we see that (excluding the malfunctioning 6 node run) larger cluster sizes give shorter response times.

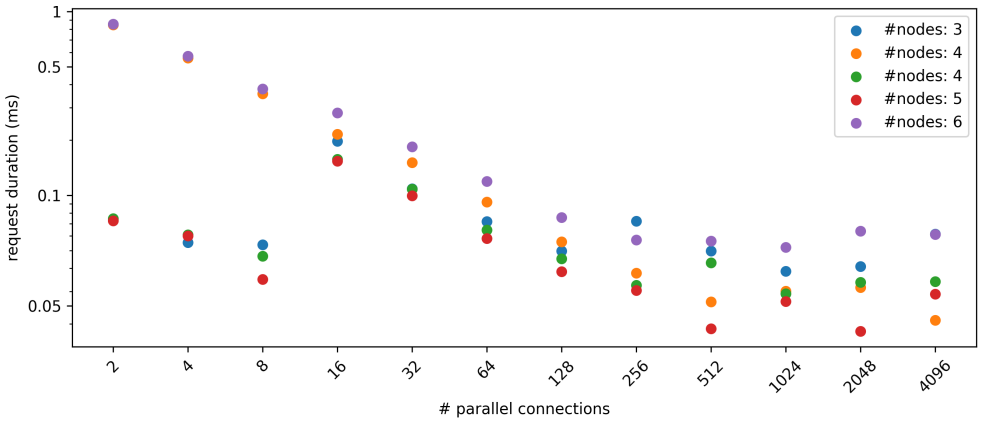

Figure 4: *Average duration of a list directory request vs number of simultaneous connections for various cluster sizes (#nodes). The run with 6 nodes experienced problems during its run slowing it down.*

4 Conclusion

The file system is consistent and stable under the lightest write and high read loads. Since I did not test other file systems under the same workload I can not conclude wheather the read performance is good.

In fig. 3 we see a make directory load throws off the heartbeat timing of the writeserver resulting in readservers starting elections. The writeserver does

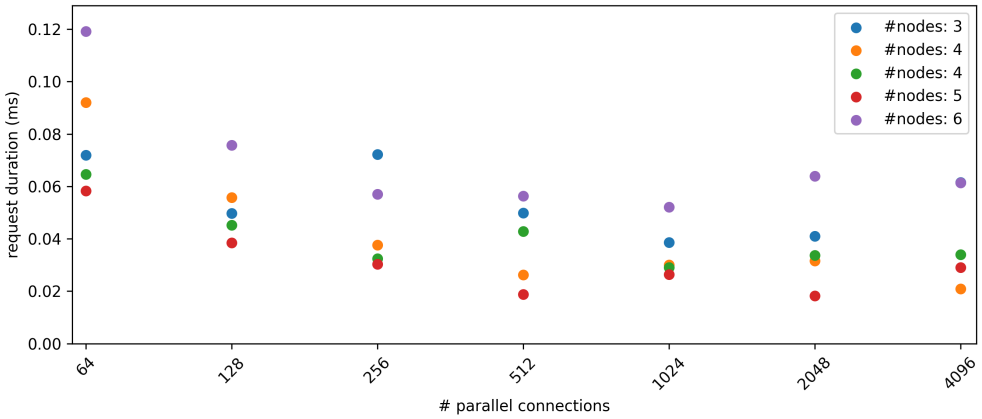

Figure 5: *Average duration of a list directory request vs number of simultaneous connections for various cluster sizes (#nodes). The run with 6 nodes experienced problems during its run slowing it down.*

not convert to a readserver[7] which is an implementation error. The readservers, having elected a new leader, no longer acknowledge the writeservers changes which it handles by crashing. This brings the cluster to a consistent state but errors the client connection which stopped the test. There are two underlying problems here:

1. A load of 5 requests per second should not throw off heartbeat timing. An increase in heartbeat timeout did not solve the problem pointing to a problem in my consistency model. It might have to do with the way heartbeat messages are canceled and replaced by publish change messages (see fig. 2 *'cancel next heartbeat'*).
2. The writeserver should put up some *back-pressure* when the load becomes to high. It should stop handling new client requests instead of missing heartbeats.

I also ran into a fundamental issue with the FS. Only a single change can be published to the readservers at the time. If multiple changes are sent concurrently the readserver will eventually receive some changes out of order and conclude it is outdated and go down.

I can conclude the system, though consistent, is highly unstable during writes.

[7] follower in *Raft* terms

The current implementation is faulty and there is a fundamental problem using raft that limits performance. It seems that read performance scales linearly with cluster size.

4.1 Future work

I think the fundamental issue can be solved with a slight change to *Raft*. In *Raft* a message is rejecting if the previous log index (*prevLogIndex*) of the message is not found in the receivers log. I propose to introduce a pending state in which a message can be partially accepted for some duration. If the previous log index is not found a message will not (yet) be accepted but be put into a pending state until a message with the previous log index is accepted. The message is only acknowledged to the master if one with the missing previous log index is accepted. This should be worked out on paper then implemented.

Further work should add a form of back-pressure and change task scheduling such that heartbeats will be sent on time even under the highest possible load. As we use cooperative multitasking this should be possible.

To quantify performance I need to compare the system to existing solutions such as HDFS. For this a (minimal) data plane needs to be added and a real world workload selected. Then both systems can be benchmarked on the same cluster.

Currently, a readserver that is outdated asks the writeserver for the complete directory. With a large directory and a high amount of changes this can result in an endless loop as the readserver is outdated by the time the writeserver has sent the directory. This can be fixed with the log entry part of *Raft* or another system for incremental updates.

Finally, there are a lot of optimizations possible. For example currently, except for publishing changes, a new connection is started for each request.

A Run Instructions

To run the implementation clone the git repository[8] on a cluster which uses preserve for node allocation and which has bash, wget, make > 1.7, tar and curl available. Then adjust the *SCRATCH* target at the top of the Makefile to point to a directory with about 5 gigabytes of storage[9]. Now the experiments can run using (make <target-name> and one of these targets:

1. `test_mkdir`: Runs the steps of the first test described in section 3. Prints the nodes used and provides live log output in tmux for the various nodes. Used in combination with some manual `ssh`, `ps aux | grep mock` and `kill` to kill the writeserver in the second test.
2. `bench_mkdir`: Recreates the third test.
3. `bench_ls`: Runs the fourth and final test: read performance.

This will set up a rust tool chain in your *SCRATCH*, use it to build the needed binaries and then deploy them to the nodes. To change the experiments edit the shell scripts in the `scripts/bench` and `scripts/test` directories. The logs where reduced to raw data using the following bash line:

```
cat hb_timeout.txt \
  | uniq -u \
  | grep "timeout␣in" \
  | cut -d "␣" -f 1,6 \
  | cut -d ":" -f 3- >> hb_timeout.dat
```

And then the python script `data/main.py` created the plots.

B Async

Async is a syntactic language feature that allows for easy construction of asynchronous non-blocking functions. *Asynchronous* programming lets us write concurrent, not parallel, tasks while looking awfully similar to normal blocking programming. It is a good alternative to *event-driven* programming which tends to be verbose and hard to follow. All Async systems are build around special function that do not return a value but rather a *promise* of a

[8] https://github.com/dvdsk/mock-fs
[9] The rust build system is hungry for storage

future value. When we need the value we tell the program not to continue until the promise is fulfilled. Let's look at the example of downloading 2 files:

```
1  async fn get_two_sites_async() {
2    // Create two different "futures" which, when run to
3    // completion, will asynchronously download the web pages.
4    let future_one = download_async("https://www.foo.com");
5    let future_two = download_async("https://www.bar.com");
6
7    // Run both futures to completion at the same time.
8    let futures_joined = join!(future_one, future_two);
9    // Run them to completion returning their return values
10   let (foo, bar) = futures_joined.await;
11   some_function_using(foo,bar);
12 }
```

Notice the `async` keyword in front of the function definition, it means the function will return a promise to complete in the future. The `join!` statement on line 8 combines the two promises for a future answer to a single promise for two answers. In line 10 we await or 'block' the program until `futures_joined` turns into two value. Those can then be used in normal and async functions.

The caller of our `async` *get_two_sites_async* function will need to be an another async function that can await *get_two_sites_async*, or it can be an executor. An executor allows a normal function to await async functions.

Let's go through our example again explaining how this mechanism could work. The syntax and workings of async differ a lot here we will look at the language *Rust*. In rust these promises for a future value are called futures. Until the program reaches line 10 no work on downloading the example sites is done. This is not a problem as the results, *foo* and *bar*, are not used before line 11. The runtime will start out working on downloading `www.foo.com`, probably by sending out a DNS request. As soon as the DNS request has been sent we need to wait for the answer, we need it to know to which IP to connect to download the site. At this point the runtime will instead of waiting start work on downloading bar where it will run into the same problem. If by now we have received an answer on our DNS request for *www.foo.com* the runtime will continue its work on downloading foo. If not the runtime might continue on some other future available to it that can do work at this point.

References

[1] Apache. *HDFS High Availability*. https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html. 2021.

[2] Apache. *HDFS High Availability Using the Quorum Journal Manager*. https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html. 2021.

[3] Yucheng Fang et al. "Modeling and Verifying MooseFS in CSP". In: July 2018, pp. 270–275. DOI: 10.1109/COMPSAC.2018.00043.

[4] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. "The Google File System". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, 2003, pp. 20–43.

[5] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[6] Manikandan Selvaganesan and Mohamed Ashiq Liazudeen. "An Insight about GlusterFS and Its Enforcement Techniques". In: *2016 International Conference on Cloud Computing Research and Innovations (ICCCRI)*. 2016, pp. 120–127. DOI: 10.1109/ICCCRI.2016.26.

[7] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972.

[8] Sage A Weil et al. "Ceph: A scalable, high-performance distributed file system". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 307–320.