Bringing theoretical design and observed performances face to face

Published on **September 26, 2022**
24 min reading time 4717 words

For the past years, we have extensively analyzed possible design decisions and their theoretical tradeoffs on Garage, especially on the network, data structure, or scheduling side. And it worked well enough for our production cluster at Deuxfleurs, but we also knew that people started discovering some unexpected behaviors. We thus started a round of benchmark and performance measurements to see how Garage behaves compared to our expectations. We split them into 3 categories: "efficient I/O", "myriads of objects" and "resiliency" to reflect the high-level properties we are seeking.

⚠️ Disclaimer

The following results must be taken with a critical grain of salt due to some limitations that are inherent to any benchmark. We try to reference them as exhaustively as possible in this section, but other limitations might exist.

Most of our tests are done on simulated networks that can not represent all the diversity of real networks (dynamic drop, jitter, latency, all of them could be correlated with throughput or any other external event). We also limited ourselves to very small workloads that are not representative of a production cluster. Furthermore, we only benchmarked some very specific aspects of Garage: our results are thus not an overview of the whole software performance.

For some benchmarks, we used Minio as a reference. It must be noted that we did not try to optimize its configuration as we have done on Garage, and more generally, we have way less knowledge on Minio than on Garage, which can lead to underrated performance measurements for Minio. It must also be noted that Garage and Minio are systems with different feature sets, *eg.* Minio supports erasure coding for better data density while Garage doesn't, Minio implements way more S3 endpoints than Garage, etc. Such features have necessarily a cost

that you must keep in mind when reading plots. You should consider Minio results as a way to contextualize our results, to check that our improvements are not artificials compared to existing object storage implementations.

The impact of the testing environment is also not evaluated (kernel patches, configuration, parameters, filesystem, hardware configuration, etc.), some of these configurations could favor one configuration/software over another. Especially, it must be noted that most of the tests were done on a consumer-grade computer and SSD only, which will be different from most production setups. Finally, our results are also provided without statistical tests to check their significance, and thus might be statistically not significant.

When reading this post, please keep in mind that **we are not making any business or technical recommendations here, this is not a scientific paper either**; we only share bits of our development process as honestly as possible. Read **benchmarking crimes**, make your own tests if you need to take a decision, and remain supportive and caring with your peers...

About our testing environment

We started a batch of tests on **Grid5000**, a large-scale and flexible testbed for experiment-driven research in all areas of computer science, under the **Open Access** program. During our tests, we used part of the following clusters: **nova**, **paravance**, and **econome** to make a geo-distributed topology. We used the Grid5000 testbed only during our preliminary tests to identify issues when running Garage on many powerful servers, issues that we then reproduced in a controlled environment; don't be surprised then if Grid5000 is not mentioned often on our plots.

To reproduce some environments locally, we have a small set of Python scripts named **mknet** tailored to our needs[1]. Most of the following tests were thus run locally with mknet on a single computer: a Dell Inspiron 27" 7775 AIO, with a Ryzen 5 1400, 16GB of RAM, a 512GB SSD. In terms of software, NixOS 22.05 with the 5.15.50 kernel is used with an ext4 encrypted filesystem. The `vm.dirty_background_ratio` and `vm.dirty_ratio` have been reduced to `2` and `1` respectively as, with default values, the system tends to freeze when it is under heavy I/O load.

Efficient I/O

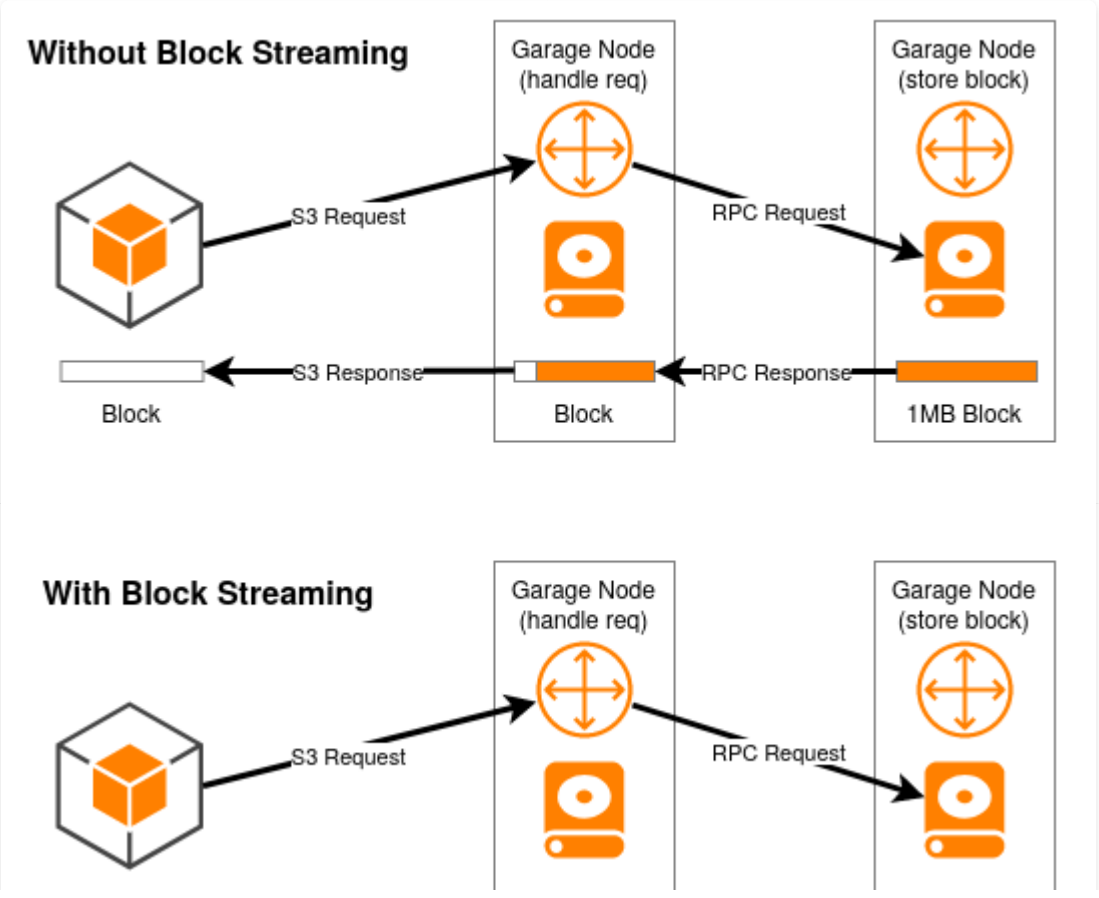
The main goal of an object storage system is to store or retrieve an object across the network, and the faster, the better. For this analysis, we focus on 2 aspects:

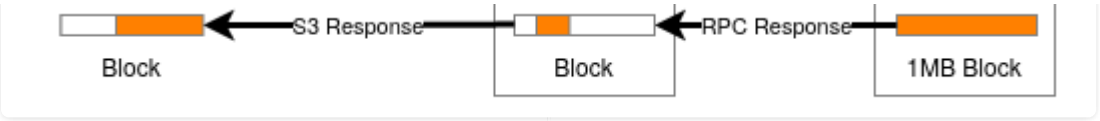
time to first byte, as many applications can start processing a file before receiving it completely, and generic throughput, to understand how well Garage can leverage the underlying machine performances.

Time To First Byte - One specificity of Garage is that we implemented S3 web endpoints, with the idea to make it the platform of choice to publish your static website. When publishing a website, one metric you observe is Time To First Byte (TTFB), as it will impact the perceived reactivity of your website. On Garage, time to first byte was a bit high.

This is not surprising as, until now, the smallest level of granularity internally was handling full blocks. Blocks are 1MB chunks (this is **configurable**) of a given object. For example, a 4.5MB object will be split into 4 blocks of 1MB and 1 block of 0.5MB. With this design, when you were sending a GET request, the first block had to be fully retrieved by the gateway node from the storage node before starting to send any data to the client.

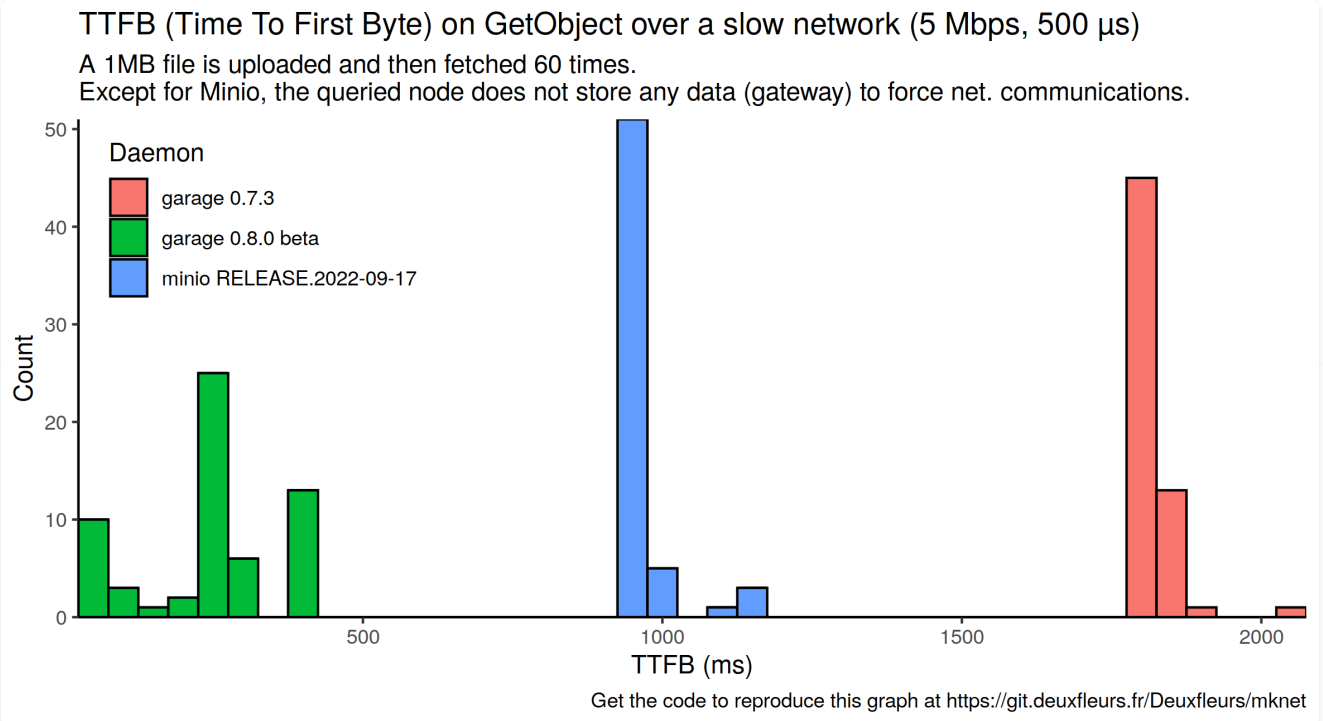
With Garage v0.8, we integrated a block streaming logic that allows the gateway to send the beginning of a block without having to wait for the full block from the storage node. We can visually represent the difference as follow:





As our default block size is only 1MB, the difference will be very small on fast networks: it takes only 8ms to transfer 1MB on a 1Gbps network. However, on a very slow network (or a very congested link with many parallel requests handled), the impact can be much more important: at 5Mbps, it takes 1.6 seconds to transfer our 1MB block, and streaming could heavily improve user experience.

We wanted to see if this theory holds in practice: we simulated a low latency but slow network on mknet and did some requests with (garage v0.8 beta) and without (garage v0.7) block streaming. We also added Minio as a reference. To benchmark this behavior, we wrote a small test named **s3ttfb**, its results are depicted in the following figure.



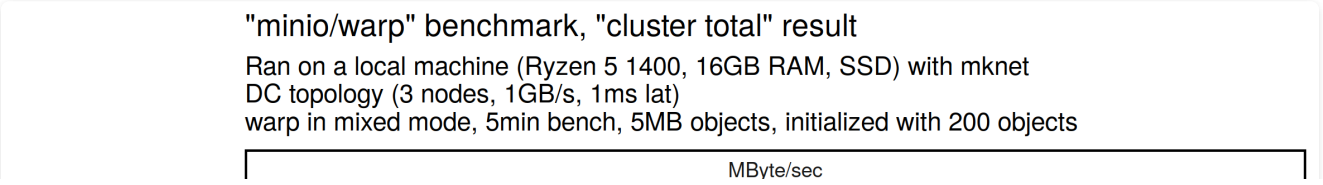
Garage v0.7, which does not support block streaming, features TTFB between 1.6s and 2s, which corresponds to the theoretical time to transfer the full block. On the other side of the plot, Garage v0.8 has a very low TTFB thanks to the streaming feature (the lowest value is 43 ms). Minio sits between the two Garage

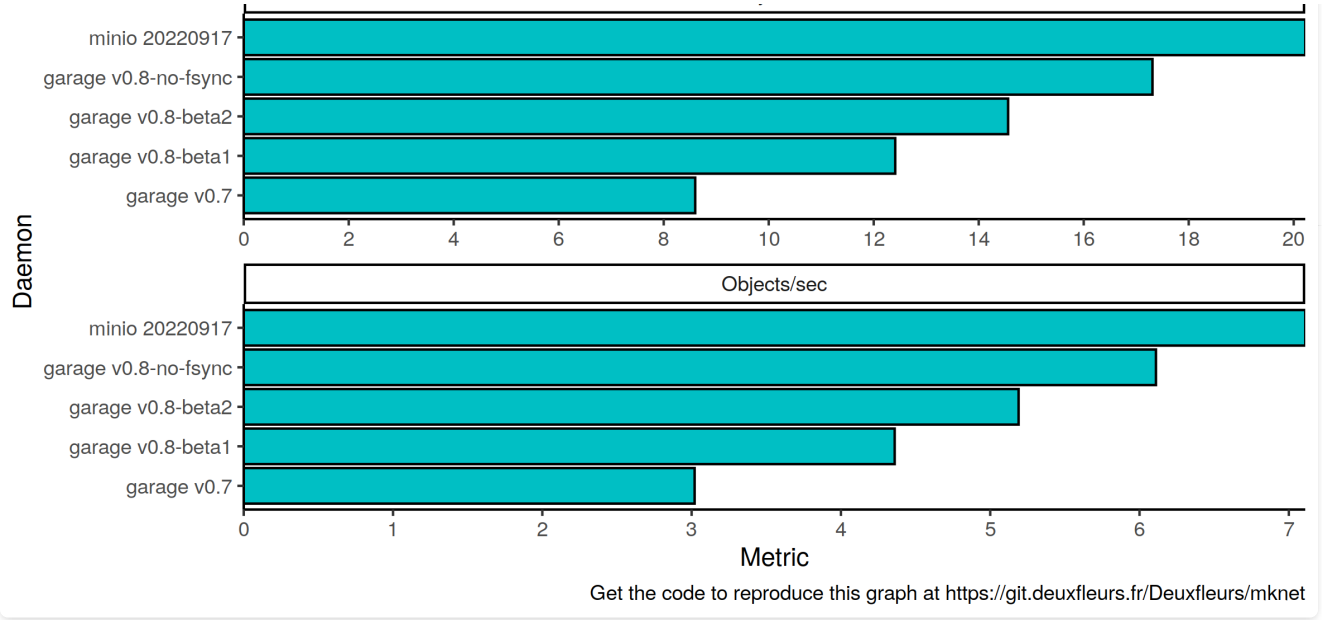
streaming feature (the lowest value is 45 ms). Minio sits between the two Garage versions: we suppose that it does some form of batching, but smaller than 1MB.

Throughput - As soon as we publicly released Garage, people started benchmarking it, comparing its performances to writing directly on the filesystem, and observed that Garage was slower (eg. **#288**). To improve the situation, we put costly processing like hashing on a dedicated thread and did many compute optimization (**#342**, **#343**) which lead us to `v0.8 beta 1`. We also noted logic we wrote (to better control resource usage and detect errors, like semaphores or timeouts) was artificially limiting performances. In another iteration, we made this logic less restrictive at the cost of higher resource consumption under load (**#387**), resulting in `v0.8 beta 2`. Finally, we currently do multiple `fsync` calls each time we write a block. We know that this is expensive and did a test build without any `fsync` call (**see the commit**) that will not be merged, just to assess the impact of `fsync`. We refer to it as `no-fsync` in the following plot.

*A note about fsync: for performance reasons, operating systems often do not write directly to the disk when a process creates or updates a file in your filesystem, instead, the write is kept in memory, and flushed later in a batch with other writes. If a power loss occurs before the OS has time to flush the writes on the disk, data will be lost. To ensure that a write is effectively written on disk, you must use the **fsync(2)** system call: it will block until your file or directory has been written from your volatile memory to your persisting storage device. Additionally, the exact semantic of fsync **differs from one OS to another** and, even on battle-tested software like Postgres, **they "did it wrong for 20 years"**. Note that on Garage, we are currently working on our "fsync" policy and thus, for now, you should expect limited data durability in case of power loss, as we are aware of some inconsistency on this point (which we describe in the following and plan to solve).*

To assess performance improvements, we used the benchmark tool **minio/warp** in a non-standard configuration, adapted for small-scale tests, and we kept only the aggregated result named "cluster total". The goal of this experiment is to get an idea of the cluster performance with a standardized and mixed workload.





Minio, our ground truth, features the best performances in this test. Considering Garage, we observe that each improvement we made has a visible impact on its performances. We also note that we have a progress margin in terms of performances compared to Minio: additional benchmarks, tests, and monitoring could help better understand the remaining difference.

A myriad of objects

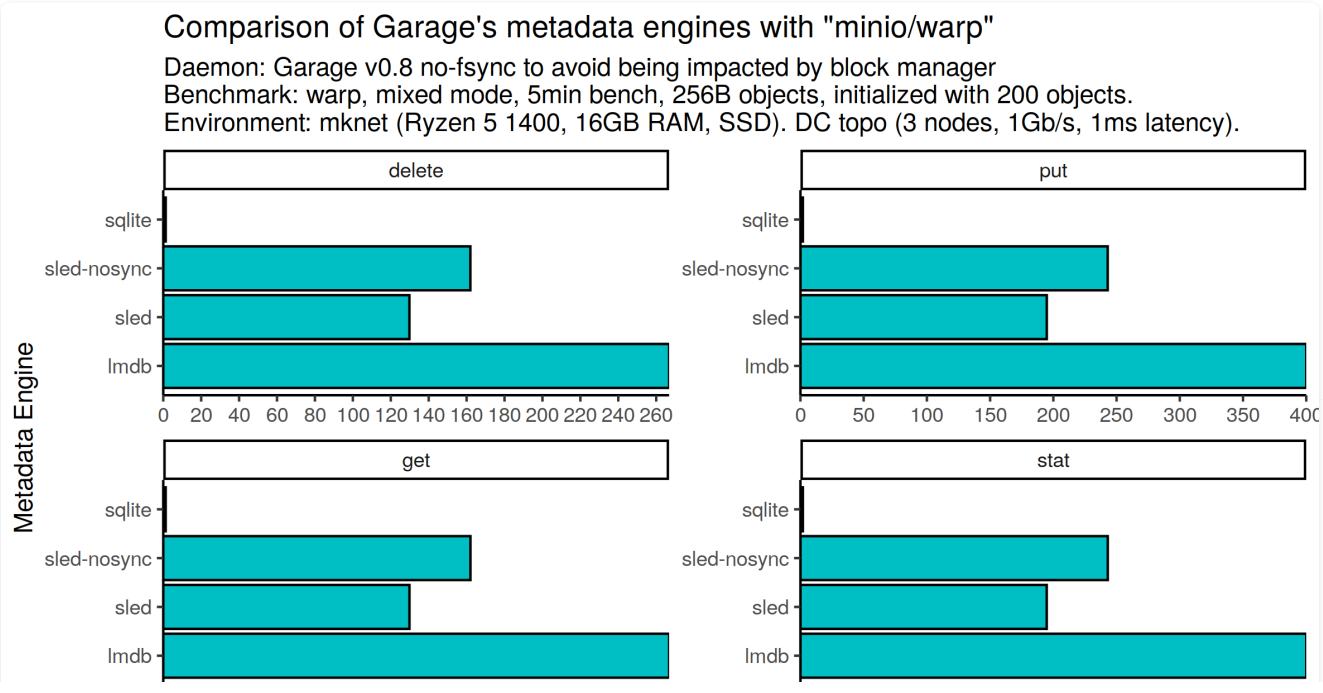
Object storage systems do not handle a single object but a myriad of them: Amazon claims to handle trillions of objects on their platform, and Red Hat communicates about Ceph being able to handle 10 billion objects. All these objects must be tracked efficiently in the system to be fetched, listed, removed, etc. In Garage, we use a "metadata engine" component to track them. For this analysis, we compare different metadata engines in Garage and see how well the best one scale to a million objects.

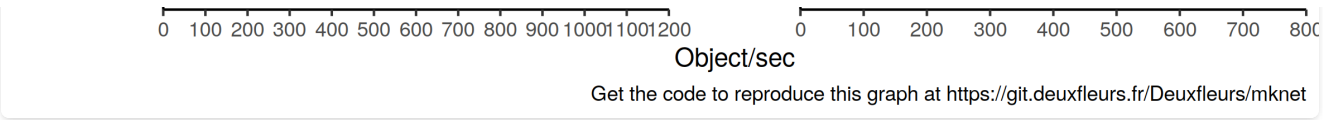
Testing metadata engines - With Garage, we chose to not store metadata directly on the filesystem, like Minio for example, but in an on-disk fancy B-Tree structure, in other words, in an embedded database engine. Until now, the only available option was **sled**, but we started having serious issues with it, and we were not alone (**#284**). With Garage v0.8, we introduce an abstraction semantic over the features we expect from our database, allowing us to switch from one backend to another without touching the rest of our codebase. We added two additional backends: lmdb (**heed**) and sqlite (**rusqlite**). **Keep in mind that they are both experimental: contrarily to sled, we have never run them in production for a long time.**

Similarly to the impact of fsync on block writing, each database engine we use has its own policy with fsync. Sled flushes its write every 2 seconds by default, this is **configurable**). lmdb by default does an `fsync` on each write, on early tests it led to very slow resynchronizations between nodes. We added 2 flags: **MDB_NOSYNC** and **MDB_NOMETASYNC** which deactivate fsync. On sqlite, it is also possible to deactivate fsync with `pragma synchronous = off;`, but we did not start any optimization work on it: our sqlite implementation fsync all the data on the disk. Additionally, we are using these engines through a Rust binding that had to do some tradeoff on the concurrency part. **Our comparison will not reflect the raw performances of these database engines, but instead, our integration choices.**

Still, we think it makes sense to evaluate our implementations in their current state in Garage. We designed a benchmark that is intensive on the metadata part of the software, ie. handling tiny files. We chose again minio/warp but we configure it with the smallest possible object size supported by warp, 256 bytes, to put some pressure on the metadata engine. We evaluate sled twice: with its default configuration, and with a configuration where we set a flush interval of 10 minutes to disable fsync.

Note that S3 has not been designed for such small objects; a regular database, like Cassandra, would be more appropriate for such workloads. This test has only been designed to stress our metadata engine, it is not indicative of real-world performances.





Unsurprisingly, we observe abysmal performances for sqlite, the engine we have the less tested and kept fsync for each write. lmdb performs twice better than sled in its default version and 60% better than the "no fsync" version in our benchmark. Furthermore, and not depicted on these plots, LMDB uses way less disk storage and RAM; we would like to quantify that in the future. As we are only at the very beginning of our work on metadata engines, it is hard to draw strong conclusions. Still, we can say that sqlite is not ready for production workloads, LMDB looks very promising both in terms of performances and resource usage, it is a very good candidate for Garage's default metadata engine in the future, and we need to define a data policy for Garage that would help us arbitrate between performances and durability.

*To fsync or not to fsync? Performance is nothing without reliability, so we need to better assess the impact of validating a write and then losing it. Because Garage is a distributed system, even if a node loses its write due to a power loss, it will fetch it back from the 2 other nodes storing it. But rare situations where 1 node is down and the 2 others validated the write and then lost power can occur, what is our policy in this case? For storage durability, we are already supposing that we never lose the storage of more than 2 nodes, should we also expect that we don't lose power on more than 2 nodes at the same time? What should we think about people hosting all their nodes at the same place without a UPS? Historically, it seems that Minio developers also accepted some compromises on this side (**#3536**, **HN Discussion**). Now, they seem to use a combination of `O_DSYNC` and `fdatasync(3p)` - a derivative that ensures only data and not metadata are persisted on disk - in combination with `O_DIRECT` for direct I/O (**discussion**, **example in minio source**).*

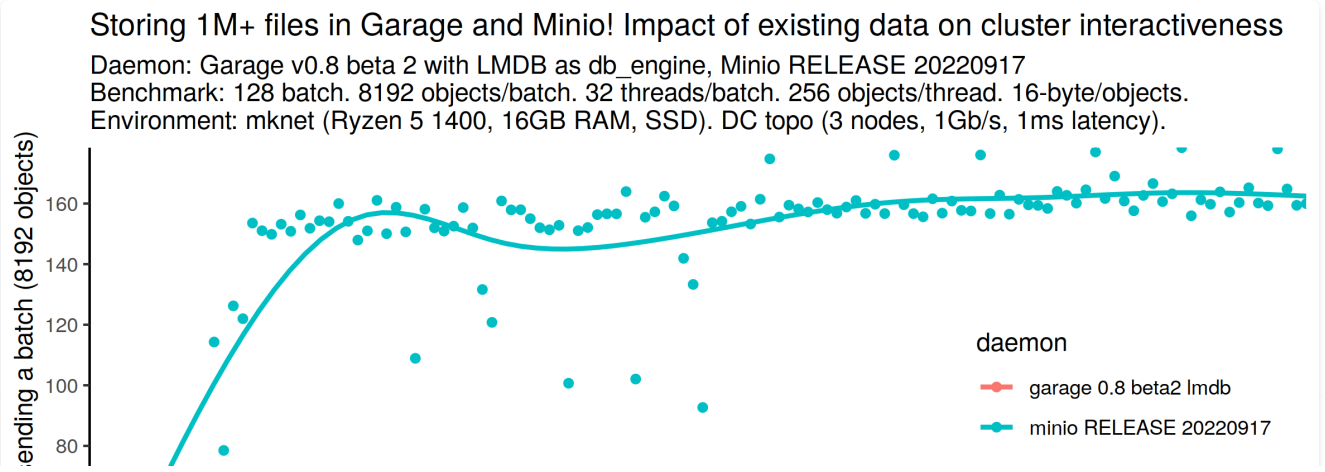
Storing a million objects - Object storage systems are designed not only for data durability and availability but also for scalability. Following this observation, some people asked us how scalable Garage is. If answering this question is out of the scope of this study, we wanted to be sure that our metadata engine would be able to scale to a million objects. To put this target in context, it remains small compared to other industrial solutions: Ceph claims to scale up to **10 billion objects**, which is 4 orders of magnitude more than our current target. Of course, their benchmarking setup has nothing in common with ours, and their tests are way more exhaustive.

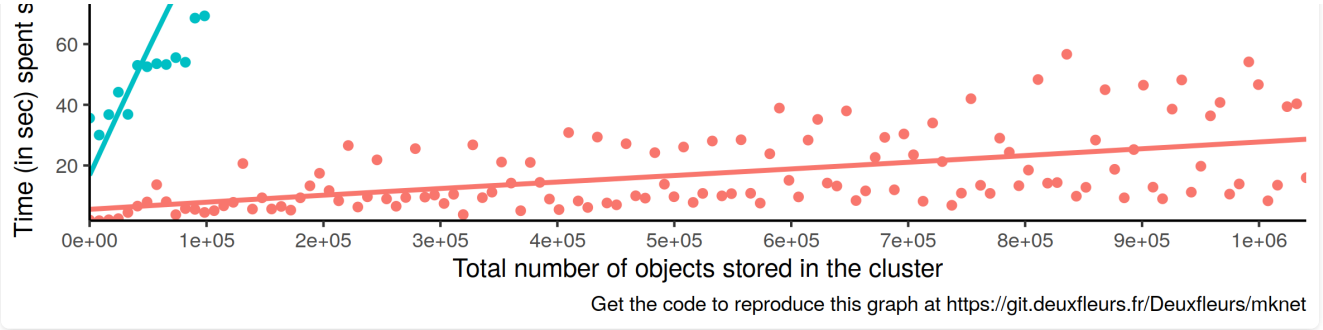
We wrote our own benchmarking tool for this test, **s3billion**[2]. It concurrently sends a defined number of very tiny objects (8192 objects of 16 bytes by default) and measures the time it took. It repeats this step a given number of times (128 by default) to effectively create a certain number of objects on the target cluster (1M by default). On our local setup with 3 nodes, both Minio and Garage with LMDB were able to achieve this target. In the following plot, we show how many times it took to Garage and Minio to handle each batch.

Before looking at the plot, **you must keep in mind some important points about Minio and Garage internals**.

Minio has no metadata engine, it stores its objects directly on the filesystem. Sending 1 million objects on Minio results in creating one million inodes on the storage node in our current setup. So the performance of your filesystem will probably substantially impact the results you will observe; we know the filesystem we used is not adapted at all for Minio (encryption layer, fixed number of inodes, etc.). Additionally, we mentioned earlier that we deactivated fsync for our metadata engine, minio has some fsync logic here slowing down the creation of objects. Finally, object storage is designed for big objects: this cost is negligible with bigger objects. In the end, again, we use Minio as a reference to understand what is our performance budget for each part of our software.

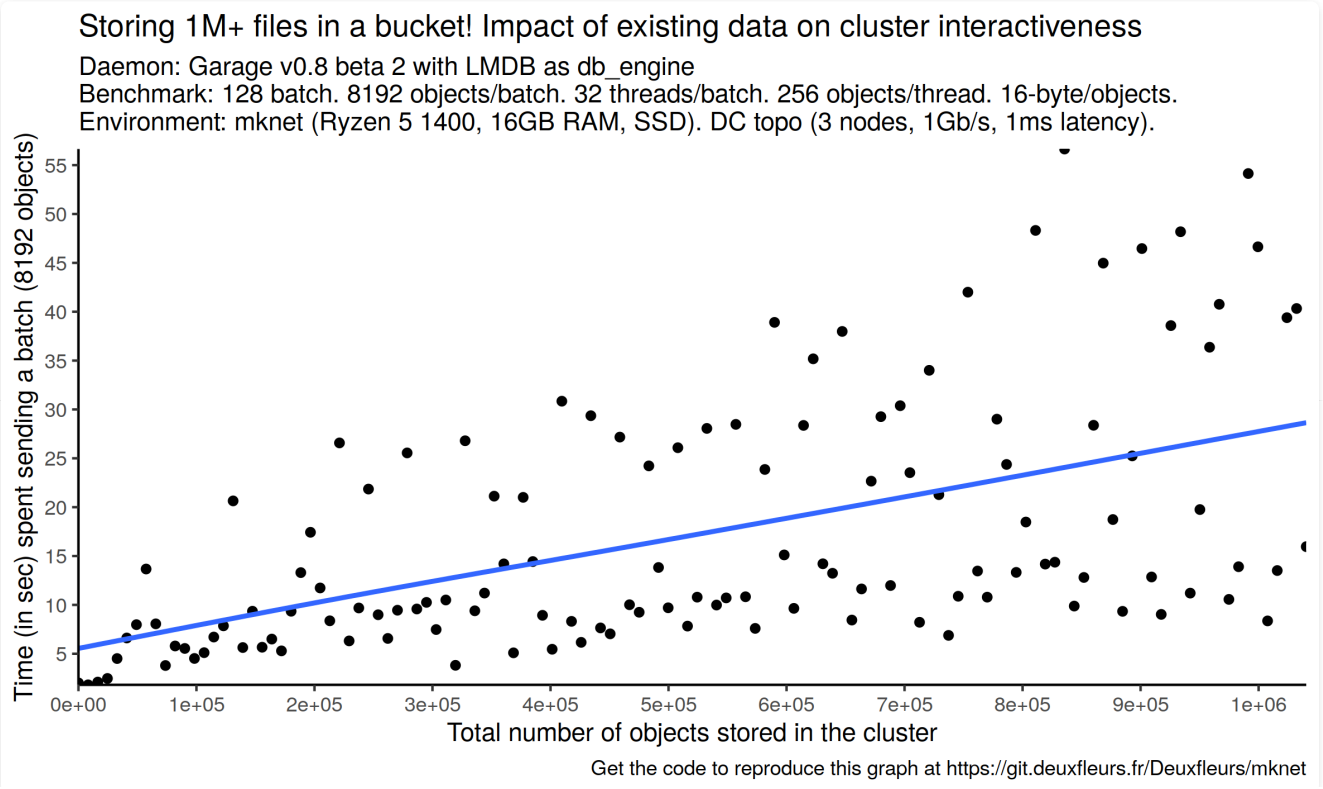
Conversely, Garage has an optimization for small objects. Below 3KB, a block is not created on the filesystem but the object is directly stored inline in the metadata engine. In the future, we plan to evaluate how Garage behaves with 3KB+ objects at scale, probably way closer to Minio, as it will have to create an inode for each object. For now, we limit ourselves to evaluating our metadata engine and thus focus only on 16-byte objects.





It appears that the performances of our metadata engine are acceptable, as we have a comfortable margin compared to Minio (Minio is between 3x and 4x times slower per batch). We also note that, past 200k objects, Minio batch completion time is constant as Garage's one remains linear: it could be interesting to know if Garage batch's completion time would cross Minio's one for a very large number of objects. If we reason per object, both Minio and Garage performances remain very good: it takes respectively around 20ms and 5ms to create an object. At 100 Mbps, if you upload a 10MB file, the upload will take 800ms, for a 100MB file, it goes up to 8sec; in both cases handling the object metadata is only a fraction of the upload time. The only cases where you could notice it would be if you upload a lot of very small files at once, which again, is an unusual usage of the S3 API.

Next, we focus on Garage's data only to better see its specific behavior:



Two effects are now more visible: 1. batch completion time is linear with the number of objects in the bucket and 2. measurements are dispersed, at least more than Minio. We discussed the first point previously but not the second one on measurement dispersion. This instability could be an issue as it could be a symptom of what we saw with some other experiments in this machine: sometimes it freezes under heavy I/O operations. Such freezes could lead to request timeouts and failures. If it occurs on our testing computer, it will occur on other servers too: it could be interesting to better understand this issue, document how to avoid it, or change how we handle our I/O. At the same time, this was a very stressful test that will probably not be encountered in many setups: we were adding 273 objects per second for 30 minutes!

To conclude this part, Garage can ingest 1 million tiny objects while remaining usable on our local setup. To put this result in perspective, our production cluster at **deuxfleurs.fr** smoothly manages a bucket with 116k objects. This bucket contains real data: it is used by our Matrix instance to store people's media files (profile pictures, shared pictures, videos, audios, documents...). Thanks to this benchmark, we have identified two points of vigilance: putting object duration seems linear with the number of existing objects in the cluster, and we have some volatility in our measured data that could be a symptom of our system freezing under the load. Despite these two points, we are confident that Garage could scale way above 1M+ objects, but it remains to be proved!

In an unpredictable world, stay resilient

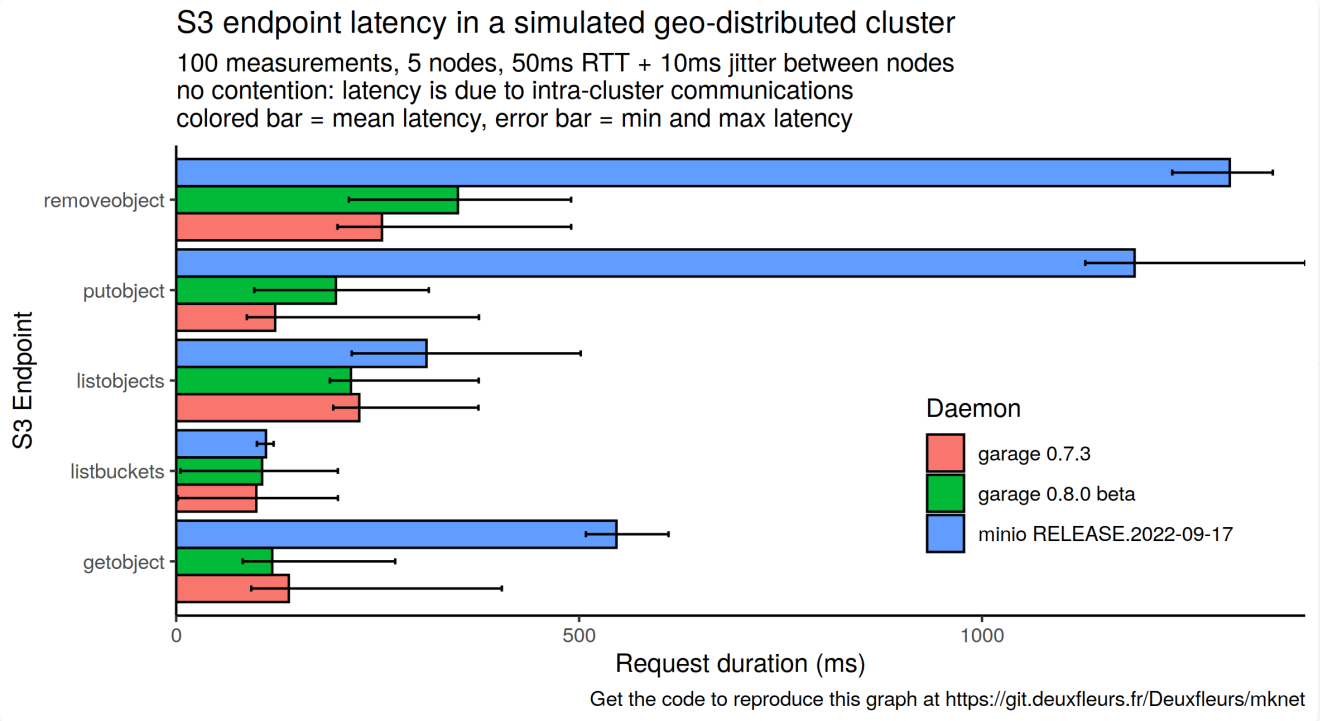
Supporting a variety of network properties and computers, especially ones that were not designed for software-defined storage or even server purposes, is the core value proposition of Garage. For example, our production cluster is hosted **on refurbished Lenovo Thinkcentre Tiny Desktop computers** behind consumer-grade fiber links across France and Belgium - if you are reading this, congratulation, you fetched this webpage from it! That's why we are very careful that our internal protocol (named RPC protocol in our documentation) remains as lightweight as possible. For this analysis, we quantify how network latency and the number of nodes in the cluster impact S3 main requests duration.

Latency amplification - With the kind of networks we use (consumer-grade fiber links across the EU), the observed latency is in the 50ms range between nodes. When latency is not negligible, you will observe that request completion time is a factor of the observed latency. That's expected: in many cases, the node

of the cluster you are contacting can not directly answer your request, it needs to reach other nodes of the cluster to get your information. Each sequential RPC adds to the final S3 request duration, which can quickly become expensive. This ratio between request duration and network latency is what we refer to as *latency amplification*.

For example, on Garage, a GetObject request does two sequential calls: first, it asks for the descriptor of the requested object containing the block list of the requested object, then it retrieves its blocks. We can expect that the request duration of a small GetObject request will be close to twice the network latency.

We tested this theory with another benchmark of our own named **s3lat** which does a single request at a time on an endpoint and measures its response time. As we are not interested in bandwidth but latency, all our requests involving an object are made on a tiny file of around 16 bytes. Our benchmark tests 5 standard endpoints: ListBuckets, ListObjects, PutObject, GetObject and RemoveObject. Its results are plotted here:



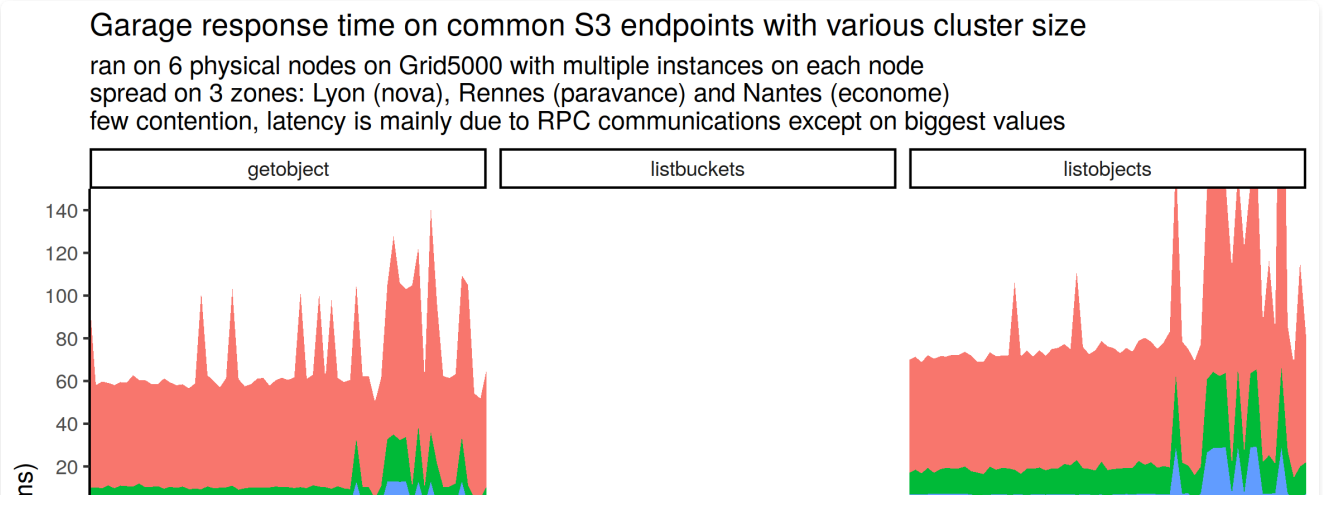
As Garage has been optimized for this use case from the beginning, we don't see any significant evolution from one version to another (garage v0.7.3 and garage v0.8.0 beta here). Compared to Minio, these values are either similar (for ListObjects and ListBuckets) or way better (for GetObject, PutObject, and

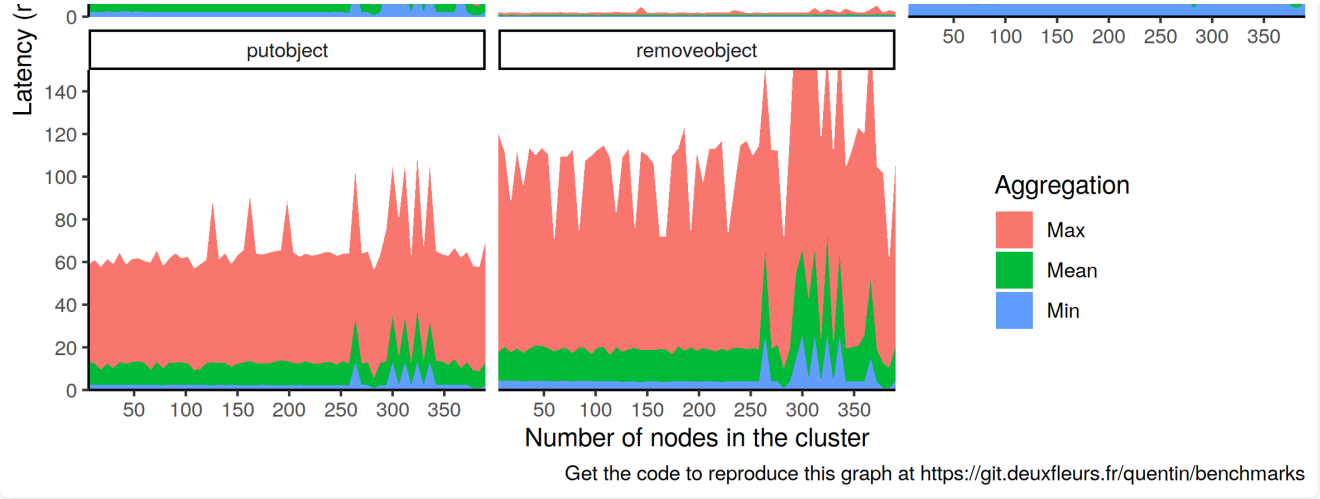
RemoveObject). It is understandable: Minio has not been designed for environments with high latencies, you are expected to build your clusters in the same datacenter, and then possibly connect them with their asynchronous **Bucket Replication** feature.

*Minio also has a **Multi-Site Active-Active Replication System** but it is even more sensitive to latency: "Multi-site replication has increased latency sensitivity, as MinIO does not consider an object as replicated until it has synchronized to all configured remote targets. Replication latency is therefore dictated by the slowest link in the replication mesh."*

A cluster with many nodes - Whether you already have many compute nodes with unused storage, need to store a lot of data, or experiment with unusual system architecture, you might want to deploy a hundredth of Garage nodes. However, in some distributed systems, the number of nodes in the cluster will impact performance. Theoretically, our protocol inspired by distributed hashtables (DHT) should scale fairly well but we never took the time to test it with a hundredth of nodes before.

This time, we did our test directly on Grid5000 with 6 physical servers spread in 3 locations in France: Lyon, Rennes, and Nantes. On each server, we ran up to 65 instances of Garage simultaneously (for a total of 390 nodes). The network between the physical server is the dedicated network provided by Grid5000 operators. Nodes on the same physical machine communicate directly through the Linux network stack without any limitation: we are aware this is a weakness of this test. We still think that this test can be relevant as, at each step in the test, each instance of Garage has 83% (5/6) of its connections that are made over a real network. To benchmark each cluster size, we used **s3lat** again:





Up to 250 nodes observed response times remain constant. After this threshold, results become very noisy. By looking at the server resource usage, we saw that their load started to become non-negligible: it seems that we are not hitting a limit on the protocol side but we have simply exhausted the resource of our testing nodes. In the future, we would like to run this experiment again, but on way more physical nodes, to confirm our hypothesis. For now, we are confident that a Garage cluster with 100+ nodes should work.

Conclusion and Future work

During this work, we identified some sensitive points on Garage we will continue working on: our data durability target and interaction with the filesystem (`O_DSYNC`, `fsync`, `O_DIRECT`, etc.) is not yet homogeneous across our components, our new metadata engines (lmdb, sqlite) still need some testing and tuning, and we know that raw I/O (GetObject, PutObject) have a small improvement margin.

At the same time, Garage has never been better: its next version (v0.8) will see drastic improvements in terms of performance and reliability. We are confident that it is already able to cover a wide range of deployment needs, up to a hundredth of nodes and millions of objects.

In the future, on the performance aspect, we would like to evaluate the impact of introducing an SRPT scheduler (**#361**), define a data durability policy and implement it, and make a deeper and larger review of the state of the art (minio, ceph, swift, openio, riak cs, seaweedfs, etc.) to learn from them, and finally, benchmark Garage at scale with possibly multiple terabytes of data and billions of objects on long-lasting experiments.

In the meantime, stay tuned: we have released **a first release candidate for Garage v0.8**, and we are working on proving and explaining our layout algorithm (**#296**), we are also working on a Python SDK for Garage's administration API (**#379**), and we will soon introduce officially a new API (as a technical preview) named K2V (**see K2V on our doc for a primer**).

Notes

[1] Yes, we are aware of **Jepsen** existence. This tool is far more complex than our set of scripts, but we know that it is also way more versatile.
[2] The program name contains the word "billion" and we only tested Garage up to 1 "million" object, this is not a typo, we were just a little bit too enthusiast when we wrote it.

We tried IPFS over Garage →