Flexible Large-Scale Data Storage
António José Freitas Pinheiro
antoniojfpinheiro@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisboa, Portugal
November 2018

Abstract
The ever increasing amount of data that companies have to process forces their infrastructure to
grow to continue serving the client’s demands. Furthermore, different classes of storage requests have
different requirements in terms of performance and consistency which leads to a tension between the
simplicity of using the same system for handling all requests and the ability to differentiate how these
requests are handled. However, companies that lack the dimension of major Internet players, find it
difficult to address this problem without reducing the quality of the service provided to the client. In this
work, we make a thorough analysis on the state-of-art regarding the different types of solutions capable
of dealing with request differentiation in large scale storage, to design a solution as capable as those
but at a severely reduced cost. After analyzing the state-of-art, this thesis proposes a hybrid approach
inspired by the software pattern called Command Query Responsibility Segregation. At its core, this
pattern segregates requests according to its functionality, whether it reads or updates data. However, in
our solution, we will go further and include the different levels of consistency in this differentiation. In
order to test our solution, we collaborated with Unbabel, a startup that offers translation as a service,
by implementing it on a specific module of their system. This solution managed to improve the read
response time of requests with more relaxed consistency requirements by achieving a speedup of 3.30
while maintaining the same performance for the remaining classes of requests.
Keywords: CQRS, Large scale storage, Availability, Consistency

1. Introduction
Nowadays, startups that process huge amounts of
data, but lack the tremendous scale and economic
power that companies like Google\(^1\) or Facebook\(^2\)
have, find some difficulty in storing and processing
that amount of information in a way that the qual-
ity of service provided to the user is not affected.

Typically, these startups use open-source stor-
age systems, for example, MongoDB \(^2\) or Cassan-
dra \(^1\), which makes them capable of coping with the
continuous growth of data through the usage of
sharding techniques, for instance. So far this
methodology has worked relatively well. However,
for simplicity, the same storage system is being used
for dealing with different data types as well as for
different kinds of requests.

Although MongoDB and Cassandra scale well in
terms of the amount of data stored, they don’t have
a way of differentiating the data types and request
types that the system has to deal with. Without
this differentiation, the load pressure related to the
read requests will affect the response time of the
write requests, and vice-versa.

\(^1\)https://www.google.com/
\(^2\)https://www.facebook.com/

In some cases, such as Unbabel\(^3\), the read load is
higher than the write load. Hence, the differentia-
tion of request type would allow them to tackle that
problem by, for instance, scaling the side which has
the most load pressure instead of their entire in-
frastucture, which naturally would reduce the cost
associated to this procedure.

A possible solution here would be to change the
storage system itself to incorporate a quality of ser-
vice concept. This would be within the reach of big
companies such as Google or Facebook, but for a
startup, which needs to focus on the rapid growth
of its business, it does not make sense to develop
in-house large-scale storage.

Currently, there are open source storage systems
capable of dealing with the data and request growth
through the usage of replication and sharding tech-
niques, such as MongoDB \(^2\), Cassandra \(^1\) or
Riak KV \(^3\). However, they do not offer data and
request differentiation which is useful to improve
the quality of service for different types of work-
loads.

On the other hand, there are systems that com-

\(^3\)https://unbabel.com/
to build a global storage system better than each one of them individually, for instance, RACS [4] or HyRD [12]. However, these systems have some objectives that differ from ours, for example, avoiding vendor-lock in.

Lastly, there are storage systems that focus on quality of service requirements. However, such systems, like Hippodrome [7] or Ergastulum [8], are often old prototypes that lack the widespread adoption or constant enhancement of a solution like MongoDB or Cassandra.

We aim to bring together the best of both worlds by using open source storage systems, benefiting from the work done by the open source community, and by introducing the concept of request and data type differentiation, where requests are segregated into two different flows according to its requirements and type.

The goal is to have the capacity to evolve along with the data growth without compromising the response times of the system. In the pursuit of this objective, our solution should be able to improve request response times and be easily scalable. Moreover, we should be able to employ this solution with the least amount of refactoring possible, with the objective of being a quicker and smoother process than completely changing the entire codebase of the system.

The main idea behind the solution is to combine different storage systems with different characteristics to create an enhanced global storage system capable of handling each request better than if we were to use the same storage system. Having this in mind, we will be exploring the CQRS [1] pattern. This pattern states that the write and read responsibilities are segregated and that should be individually addressed by different models. However, we will not only segregate requests based on their purpose, read-only or write, but also taken into account the different levels of consistency.

This paper is organized as follows: Our research on the related work is described in Section 2. Section 3 presents our proposed solution. Section 4 details the implementation. Section 5 describes the evaluation performed on our solution and Section 6 concludes the document.

2. Background
In this section we present our survey on the most relevant topics related to our work. As previously mentioned, our goal is to be capable of evolving along with the data growth without affecting the performance of the system. To that extent, there are a few systems with the capacity of mitigating this problem. However, they are not ideal or cost-effective for emerging companies.

Large-scale storage systems can be divided into two classes: SQL or NoSQL. Regardless of its type, these systems rely in replication as the path to guarantee availability and performance when processing huge amounts of data. Such as, MongoDB [2], Cassandra [11], Riak KV [3], Kafka [10], Bigtable [9], Redis [14] or BlinkDB [5].

Alternatively, we can focus on systems where the idea is to combine several systems with different characteristics in order to obtain a single global storage system with better characteristics than if we were using each one of those systems individually. These storage systems are mainly cloud storage systems, which are notorious for achieving higher availability and performance rates. Which is the case of CosTLO [17], RACS [4], SPANStore [16] or HyRD [12].

Lastly, we address this problem by instilling quality of service in storage systems. Typically, administrators configure storage manually using rules of thumb. However, taking into account that storage systems are complex and application workloads rather complicated, the resulting systems are costly or poorly setup. So, tools such as Ergastulum [8] or Minerva [6] have been proposed. However, in contrast to this class of system, we are not trying to control and improve the quality of service of a storage system by modifying that system, but instead our goal is to introduce a layer that mediates access to one or more storage systems and superimposes such quality of service guarantees on top of existing systems.

3. Proposed Solution
Taking into consideration the analysis of the solutions presented in Section 2, our proposed solution consists of combining storage systems with different characteristics to create an enhanced global storage system capable of storing and retrieving data to the user better than if the same storage system was used. However, instead of leveraging multiple cloud providers and storage systems available on the cloud, we will combine storage systems that can be all managed by the same cloud provider or even locally deployed by the organization using the system. To achieve our goal, we will explore a hybrid approach inspired by the software pattern called Command Query Responsibility Segregation [1]. This pattern states that we should segregate the models responsible for storing and retrieving data from the system. In other words, that requests that fetch information from the system should be possibly dealt with a different storage system than the requests that perform modifications on the stored data. This notion will hold true in most cases, except when dealing with requests that requires strong consistency. In this case, we will have to revert to a more costly protocol to retrieve information back to the user.
3.1. Requirements: the Unbabel use case

Unbabel\(^4\) is a company that provides translation as a service by combining humans and AI. Their ambition is to break the language barrier through the delivery of quality translations to its clients.

Their process is quite simple. Clients upload content specifying the desired language pairs, from which language to which language they want their content to be translated to, and then Unbabel returns their content translated into the chosen languages.

Internally, the uploaded content goes through a pipeline that, in a very high level, looks as follows:

1. The clients’ content is uploaded to the system
2. The content is split into smaller fragments
3. The fragments are translated using AI
4. The AI translations are edited by humans
5. The fragments are merged
6. The translated content is delivered to the client

This workflow is represented in Fig. 1\(^5\).

![Unbabel’s Internal Workflow](https://unbabel.com/)

**Figure 1: Unbabel’s Internal Workflow**

As we can see in the workflow depicted in Fig. 1, besides the clients, we also have editors, which are responsible for editing the translated text of the machine translation step. Each one of them interacts with the system differently, producing different kinds of workloads. Naturally, these workloads in conjunction, will put the system under pressure, consequently degrading the user experience.

To understand in more detail these requirements and how our solution operates, we now describe in more detail a specific aspect of this workflow, which is implemented by a subsystem called Tarkin.

Tarkin is a task manager, which means that its responsibility is to store and manage tasks. Tasks are portions of text, sent from their customers, to be translated.

At a high level, the main operations of Tarkin are:

- Assign tasks for translation to editors;
- Receive translated tasks from the editors;

Every time an editor issues a request to receive a task for translation, the task manager sends them the best suited task according to their profile.

The problem comes with the increase of the number of requests, performed by editors, and the write requests (insertions of new tasks in the system), performed by customers. It not only makes the system extremely concurrent, which may violate the condition imposed, where only one editor can access a task at a time, but it also degrades significantly the response time of the system, since the system needs to deal with both read and write workloads simultaneously. Another factor that contributes to the degradation of the response time is the need to calculate the number of available tasks for an editor that is requesting for new tasks.

3.2. Adapting the CQRS pattern

As previously introduced, in order to achieve this enhanced global storage system, we will be exploring the Command Query Responsibility Segregation [1] pattern. At a high level, this pattern states that you can differentiate the models that update and read information. A possible way to implement this is that, depending on whether the request is a read-only request or an update, the system will access two different storage systems.

However, in contrast to the original philosophy, the distinction will not only be made between read-only and update requests, but instead we will leverage the different levels of consistency between different classes of requests. This will enable us to take advantage of the benefits of this pattern and do so in a more generic way that applies to a broad range of settings that wouldn’t be applicable otherwise. For instance, two different classes of get requests with different consistency requirements can be segregated in our case, and wouldn’t be originally.

Throughout the document, requests that require strong consistency are dealt by the command side of the pattern (write layer) and read requests with weaker consistency demands are processed by the query side of the pattern (read layer).

3.3. Architecture

Fig.2 portrays the components that constitute our solution as well as the interactions that are performed between those elements. After the arrival of a request to the API layer, the best suited storage system is selected according to the requirements of that particular request.
3.4. Layers
The API is the entry point of our system. This component is responsible for processing requests. It also decides which storage system is more appropriate to help processing the incoming request, taking into account its characteristics.

The Read Layer, or the query side of the CQRS [1] pattern, contains the database that is accessed every time a request to retrieve data back to the user arrives. Since these requests are read-only, we have to take into consideration that the storage system that will be used has to perform well when dealing with this type of operations. This layer processes requests that don’t require strong consistency.

The Write Layer, or the command side of the CQRS [1] pattern, contains the database responsible for dealing with the requests that requires the insertion, modification or deletion of data from the system. When selecting which storage system to use in this layer, we need to consider that these requests are write intensive. This layer is also responsible for processing get requests with strong consistency requirements.

The Replication Layer is the pivotal point of our solution. Its main responsibility is to replicate the data from the command side to the query side, as fast as possible, thus minimizing the amount of time that the system could retrieve inconsistent data.

3.5. Data structures
Taking into consideration that scalability and performance is a must and that data structures play an important role in fulfilling these requirements.

3.5.1 Databases
Since our system is an hybrid approach to the CQRS [1] pattern, we opted to pick two different databases according to the workloads that they were going to handle. Hence, leveraging their strengths and routing the workloads that best suits each one of the storage systems.

The storage system of the write layer will maintain all the information of the system, in our use case, the editor’s profile, the tasks and the relation between editors and tasks. This information is needed to aid the POST, PUT and DELETE requests of the system. Meanwhile, the database of the read layer, only stores the number of available tasks of each editor, which is used to aid the GET requests related to the available tasks count.

3.6. Replicator queue
The replication process is one of the most crucial moments of the overall flow of the system. With this in mind, we had to be careful when designing the data structure that will aid this process. To avoid that it could become a bottleneck, and that we could easily scale, when facing huge workloads, we decided to create a queue that stores the information required to process each task (elements of the queue). For each task, we store the following information:

- The name of the function that will process the replication
- The arguments of the function, which contains the data that is going to be stored in the query side of the system

3.7. Algorithms
To provide an efficient solution, we rely on three main algorithms. The first algorithm operates similarly as a load balancer according to the characteristics of the request. The second ensures that, as soon as modifications on the stored data are detected, they are sent to the replication layer through a publish-subscribe system. At last, the third algorithm guarantees that the modifications on the system are propagated to the read layer.

3.7.1 Routing
The goal of the routing algorithm is to select which of the two storage systems, available in the read or write layer, can aid processing the request. Basically, if the request demands to insert, update or delete information from the system, then the API will access the write layer, otherwise, if the request needs to fetch information, then the read layer will be accessed. However, if a get request requires consistent data, that particular request will be routed to the write layer.

3.7.2 Replication
Every time that the system receives requests that perform modifications on the stored data, these modifications have to be replicated from the write layer to the read layer in order to keep the system
as consistent as possible. For this purpose, we have two algorithms:

- The first algorithm relies in a publish-subscribe system, which notifies the replication layer as soon as modifications on the data, stored in the write layer, are detected;
- The second, the replication layer makes use of a scheduler that pulls information from the write layer in order to keep the read layer consistent. This operation happens every x minutes, where x is configured a priori;

4. Implementation

As previously documented, our architecture is composed by four layers. In order to ensure that these layers are able to work as desired, it is necessary to transfer information from layer to layer. We will describe the interactions between those layers including the information that they share.

Starting with the API and the outside world, these two elements communicate through Representational State Transfer [15] (REST) endpoints. Which are:

- /api/v1/task
- /api/v1/editor/{editor_id}/available_tasks_count
- /api/v1/task/submit
- /api/v1/task/search

Every time new tasks arrive, they do it through the /api/v1/task endpoint, which is an HTTP POST request, that receives, for instance, the language pair of the task. The language pair is used to track the original language and the desired language of the task’s output.

The /available_tasks_count endpoint is an HTTP GET request that, given an editor id, returns the number of tasks that are available to be translated by that particular editor.

In order to retrieve a task to translation from the system, the /api/v1/task/search endpoint is used. This endpoint is an HTTP GET request that contains the editor, the language pair and the type of the task as the body of the request.

Finally, the last endpoint, /api/v1/task/submit is an HTTP POST request that succeeds a /api/v1/task/search request, which returns to the system the translation of the task previously fetched. This request has in its body the id of the task, the id of the editor that performed the translation and the translated data.

Regarding the interaction between API layer and the write layer, the communication is achieved through calls to the database that composes the write layer. When the API receives a request to store information on the write layer, the data of that particular request is forwarded to the write layer. As previously, get requests with tight consistency requirements are also addressed by the write layer, which is the case of the /api/v1/task/search endpoint.

The communication between the API layer and the read layer is effectuated in the same manner as the previous one. Every time a get request that doesn’t require strong consistency arrives, for instance, when an editor wants to know how many tasks are available for him, the API layer sends the id of the editor to the read layer and the read layer returns the number of available tasks.

To finalize, we need to address how the replication layer interacts with both the write and read layers. To ensure that the data of our solution is as consistent as possible, the write layer needs to replicate the new data to the read layer, which is the purpose of the replication layer. This process is accomplished through the usage of a publish-subscribe system and a pull mechanism. These interactions are performed by accessing both layer’s databases directly.

4.1. Components

**API Layer**

As described in the previous section, the API layer receives requests through REST endpoints. These requests are processed with the help of uWSGI\(^6\), a web server gateway server interface, and Flask\(^7\), which is a microframework for Python with RESTful request dispatching capabilities.

In this layer there are four endpoints. Three of them already existed in the previous version of Tarkin, the search, submit and task, while the fourth is an extension made by us. Their description goes as follows:

- /api/v1/search executes a stored procedure, in the write layer database, that makes sure that one task is only assign to an editor at a given time, with the help of locks and sql transactions;
- /api/v1/submit inserts into the write layer database a translated task;
- /api/v1/task inserts into the write layer database new tasks to be translated;
- /api/v1/editor/{editor_id}/available_tasks_count returns the number of available tasks of a given editor;

**Write Layer**

\(^6\)https://uwsgi-docs.readthedocs.io/en/latest/
\(^7\)http://flask.pocoo.org/
In the write layer we opted to use PostgreSQL [13] mainly because we are extending Tarkin, which is built on top of PostgreSQL, and also to take advantage of a couple of features, that will aid the transformation and replication process, such as triggers and notifiers.

As previously seen, every time a new request with storage intents enters the system, the API layer accesses this layer’s database and performs the duly operation. After a modification is made, we need to replicate the new data to the read side in order to keep the data as consistent as possible. This is accomplished through the usage of two different processes: a publish-subscribe system and a pull mechanism.

The publish-subscribe system is constructed on top of two stored procedures. The first one creates a trigger that will execute the second store procedure every time an INSERT, UPDATE or DELETE operation is detect on the table editor_task, referent to the relation between editor and task. This table contains the association of tasks to editors, thus affecting the result of the number of available tasks per editor. Hence the importance of sending this data as soon as possible to the replication layer. The flow of the trigger ends with the propagation of each iteration’s result through the publish-subscribe channel new_editor_task_trigger. The results are later processed by the replicator, which finishes the replication by transforming the received data and forwarding it to the read layer. The second stored procedure is used by the trigger to calculate the number of available tasks for each editor.

These interactions are performed with the help of SQLAlchemy [8], which is a object-relational mapper adapted to Python.

Read Layer
To store the information necessary to provide the results when GET requests arrive to the system, we opted to use Redis [14] due to its high performant characteristics when dealing with read requests. As an example, the data accessed by the available_tasks_count endpoint is stored in the following format:

```python
data = {'map_reduce_paid':
    {'en_es': 14},
    {'es_en': 0}
}
```

For each task type, it provides the number of available tasks per language pair of a given editor.

Replication Layer
The replication layer is the piece of the puzzle that ensures that the system is capable of retrieving data, to the outside world, as consistent as possible, when facing get requests with more relaxed requirements. Its main responsibility is to transform the data provided by the write layer into a more desired format that will be then replicated to the read layer. To this purpose, it makes use of two processes, as seen previously.

The publish-subscribe flow, that began in the write layer, ends with the replicator listening to the new_editor_task_trigger channel and transforming and forwarding the data to the read layer.

The second process used is a pull mechanism. Every x minutes, where x is defined in the configuration of the system, the replicator fetches from the write layer the number of available tasks for each editor.

As an example, the following data is received from the write layer and transformed from this format:

```python
data = [{"editor_id": 44, "language_pair": "en_es", "task_type": "map_reduce_paid", "devices": None, "editor_name": 'AS', 'count': 14},
        {"editor_id": 44, "language_pair": "es_en", "task_type": "map_reduce_paid", "devices": None, "editor_name": 'AS', 'count': 0}]
```

to:

```python
data = {'map_reduce_paid':
    {'en_es': 14},
    {'es_en': 0}]
```

After the transformation the data is then forwarded and stored in the read layer.

The functions that have impact in the transformation and replication of the are asynchronous tasks that are being processed with the help of Celery [9], which is a distributed task queue. Allowing the system to more easily scale.

5. Evaluation
The evaluation was conducted locally, using Docker [10] as the host of the components and JMeter [11] as the tool that executes the load tests. We opted to use JMeter due to its widely adoption and ability to simulate real-user behaviors for testing applications against heavy load, multiple and concurrent user traffic.

In order to have the system up and running, docker will launch the following containers: PostgreSQL, Redis, Tarkin, RabbitMQ [12], Scheduler, Replicator, Replicator Worker and Tarkin Worker.

---

9https://www.celeryproject.org/
10https://www.docker.com/
11https://jmeter.apache.org/
12https://www.rabbitmq.com/
Each container belongs to one of the four layers of our solution’s architecture. Below we map each container to the respective layer:

- PostgreSQL → Write Layer
- Redis → Read Layer
- Tarkin → API Layer
- RabbitMQ → Replication Layer
- Scheduler → API Layer
- Replicator → Replication Layer
- Replicator Worker → Replication Layer
- Tarkin Worker → API Layer

The PostgreSQL, Scheduler and the Tarkin Worker are containers that belong to the original Tarkin and are used by our solution as well. The containers are powered by an Intel i7-6660U CPU @2.4GHz, 16 GB 1867MHz LPDDR3 and 256GB SSD, running under macOS High Sierra version 10.13.6.

Our goal was to compare both solutions in the most realistic environment possible. Hence, we populated the PostgreSQL database using Unbabel’s staging environment database dump.

5.1. Workload

Tarkin was chosen as our use case not only because it’s one of the most important subsystems of Unbabel, due to its responsibility of linking machine and human translation, but also because it’s constantly under high load pressure, which makes a good use case for our solution. Having this in mind, we thought that the best way to test Tarkin was to simulate high load pressure by constantly sending concurrent requests through JMeter.

5.2. Metrics

To evaluate our solution, we decided to use the following metrics:

- Read Layer Latency
- Write Layer Latency
- Throughput

These metrics were selected taking into consideration the characteristics of the use case. Since Tarkin is a system under constant heavy load, whether from the assignment of tasks to editors or from the constant editor requests regarding their status, we found it would be good to compare the trace of both systems when dealing with this kind of requests. In addition, we would like to showcase the benefits of having requests being dealt with storage systems that suit them better.

5.3. Description

Each one of the tests was performed during one hour. Since our goal was to test our solution under intensive pressure, each one of the tests was carried out using six concurrent users.

For the read layer latency test, we executed two tests. The first is meant to compare both systems regarding the response time of read requests, taking into account that, with our solution, read requests with weaker consistency requirements are now handled by a storage system which one of its strengths is its read performance. The second test compares the benefits of segregating requests when dealing with them simultaneously, which is the case of a real workload.

Regarding the write layer latency test, we only conducted one test. Since the requests that are handled by the write layer of our solution are the same as the original system, they are only being routed differently than the read requests that don’t require strong consistency, there would be no point testing how the storage system would perform in terms of the write response time itself. Thus, we decided to test this latency when the system is being pressured with both types of requests.

Finally, the throughput section will illustrate the throughput results of the three tests that we just mentioned above.

5.4. Results

**Read Layer Latency**

As discussed in Section 5.3, the first test refers to how both systems perform when processing get requests. In order to test this, we will put the **available_tasks_count** endpoint under pressure.

![Available tasks count response time over time](image-url)

Figure 3: Available tasks count response time over time

As we can see, our solution, which is represented by the endpoint named as `/api/v1/editor/{editor_id}/available_tasks_count` (NEW), outperforms the old version of this endpoint. While the old system averaged read
response times of, approximately, 39 milliseconds, our solution was able to keep this value at the 12 ms mark. Mostly, as a result of being capable of differentiating requests and selecting the most appropriate storage system to process them.

The second test presents the results of processing the available_tasks_count endpoint while under high load pressure from the remaining endpoints. The result of such test is illustrated in Fig. 4.

![Figure 4: Available tasks count response time over time under high load pressure](image)

Table 1 details the results of this test.

**Write Layer Latency**

As previously explained, since we are using the same endpoints from the original Tarkin for the write layer, there would not be of any value to test each request individually. So, we performed a test to figure out if our solution would improve the write layer when the system is under high load pressure from both kinds of requests (the ones dealt by the read layer and the write layer).

Naturally, since we are running the tests locally, the results are not accurate and may be misleading due to the fact that every container is using the same resources. Nevertheless, we thought it would be interesting to see how both systems perform in the same environment.

To this end, Table 2 provides a summary of the endpoints, comparing the old against the new version of Tarkin. As we can see, in overall, our solution is able to slightly improve response times during load peaks, mainly by virtue of its architecture design. By routing requests through different storage systems, we are able to offload the undergoing load pressure, which allows each storage system to work with less pressure thus producing better results. However, we only managed to outperform the old version of Tarkin, under high load pressure from all kinds of requests, by a very fine margin. Which we think correlates to the fact that the tests were deployed and run in the same machine.

**Throughput**

In this section we compare the results of the throughput for each endpoint of both systems.

We start with the available_tasks_count endpoint. In Fig. 5 is illustrated the read throughput of this endpoint compared to the endpoint of the original Tarkin. As we can quickly see, there is a considerable improvement in the amount of transactions per second that the new system is capable of processing. In average, our solution can process, approximately, 67 transactions per second while the original Tarkin can only deal with 55 transactions per second.

![Figure 5: Available tasks count transactions per second](image)

Table 3 summarizes the throughput results for this test (the first test of the read layer latency section).

To finalize, since the remaining tests, the second one of the read layer latency section and the one performed in the write layer latency section, were conducted with the endpoints being under high load pressure from each other, we decided to group the results in Table 4.

6. Conclusions

Although there are different solutions to deal with the rapid increase on the amount of data that companies face nowadays, sometimes these solutions are sub optimal for emerging companies which lack on resources.

In this document, we started by analyzing the current solutions in order to identify and elaborate a solution to our problem. Due to the lack of work regarding the CQRS pattern, we decided to explore this idea. We used the knowledge gained from our research, for instance, which storage systems are better for certain types of requests as well as the idea of combining different storage systems, and built a solution inspired in this software pattern.

The results of the evaluation demonstrate that our solution is capable of processing and evolving along with the data growth while even averaging better response times. Mostly due to the ability to differentiate requests and optimize its result by selecting the best storage system according to its characteristics. Furthermore, with this architec-
Table 1: Available tasks count under pressure summary

<table>
<thead>
<tr>
<th>Endpoints</th>
<th># Samples</th>
<th>Average (ms)</th>
<th>Min (ms)</th>
<th>Max (ms)</th>
<th>90th pct (ms)</th>
<th>95th pct (ms)</th>
<th>99th pct (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/available_tasks_count (NEW)</td>
<td>10412</td>
<td>200.71</td>
<td>3</td>
<td>62053</td>
<td>644.70</td>
<td>890.00</td>
<td>1307.09</td>
</tr>
<tr>
<td>/available_tasks_count (OLD)</td>
<td>9716</td>
<td>209.09</td>
<td>1</td>
<td>60620</td>
<td>668.30</td>
<td>900.00</td>
<td>1328.43</td>
</tr>
</tbody>
</table>

Table 2: Write layer response times summary

<table>
<thead>
<tr>
<th>Endpoints</th>
<th># Samples</th>
<th>Throughput (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/available_tasks_count (NEW)</td>
<td>242790</td>
<td>67.44</td>
</tr>
<tr>
<td>/available_tasks_count (OLD)</td>
<td>197169</td>
<td>54.77</td>
</tr>
</tbody>
</table>

Table 3: Available tasks count throughput summary

<table>
<thead>
<tr>
<th>Endpoints</th>
<th># Samples</th>
<th>Throughput (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/available_tasks_count (NEW)</td>
<td>10412</td>
<td>2.89</td>
</tr>
<tr>
<td>/available_tasks_count (OLD)</td>
<td>9716</td>
<td>2.70</td>
</tr>
<tr>
<td>/task (NEW)</td>
<td>1956</td>
<td>0.54</td>
</tr>
<tr>
<td>/task (OLD)</td>
<td>2000</td>
<td>0.55</td>
</tr>
<tr>
<td>/task/search (NEW)</td>
<td>10406</td>
<td>2.89</td>
</tr>
<tr>
<td>/task/search (OLD)</td>
<td>9714</td>
<td>2.70</td>
</tr>
<tr>
<td>/task/submit (NEW)</td>
<td>14</td>
<td>0.01</td>
</tr>
<tr>
<td>/task/submit (OLD)</td>
<td>13</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 4: Throughput summary under high load pressure

future, we are able to offload the system by having different storage systems processing requests instead of pressuring only one.

There is also the opportunity to understand how our system could help the infrastructure achieving more stable numbers in terms of CPU and memory utilization, however, these are not tests that we were able to do, as explained in the previous chapter.

6.1. Future work

This thesis has focused in improving the performance of Tarkin regarding the throughput and latency of the different requests. However, due to the architecture of our solution, one of its limitations, which was not addressed in Section 5, is the possibility of delivering stale data to the user. Therefore, it would be interesting to evaluate the percentage of requests that deliver inconsistent data as well as the impact of configuring different intervals to the pull mechanism, of the replication layer, on this percentage.

Unfortunately, as a result of testing our solution locally, we were also not able to evaluate the potentially benefit of segregating the load between the different layers in terms of CPU and memory utilization inherent in each component, leaving it as future work.

In addition, taking into account the distributed architecture of this thesis, it would be worth exploring the following ideas:

- Exploit the ability of scaling each component independently according to the system’s workload;
- Exploit the benefits of using different storage systems on both read and write layers;

Acknowledgements

First and foremost, I would like to convey a special appreciation to my thesis advisor Rodrigo Rodrigues, for the dedication and endless support. My sincere gratitude also goes to my co-advisor Marcelo Lebre, for all the enthusiasm, motivation and wise suggestions throughout the last year.

A very special gratitude goes out to all down at Unbabel, for the support and friendship. With a special note to Eduardo Pereira and Emanuel Velho, for the insightful comments during this journey.

A special thanks to Sergio Mendes, Steven Silva, Rui Venancio and Paulo Figueira for the constructive criticism on the manuscript of this dissertation.

Finally, I must express my profound gratitude to my family and friends, for their wise consul and
guidance. It was their support and encouragement that got me through these hard-working years.

References