Distributed Job Scheduler

Georgi Koyrushki

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 24, 2018

Abstract

The purpose of this dissertation is to document the development of a distributed event-triggering system that can be used in the context of scheduling periodic tasks. The paper begins by introducing the reader to the subject matter and giving them an overview of the system as well as the challenges it tries to solve. It then outlines the requirements, and subsequently takes a detailed look into how the system was designed, implemented and evaluated. Throughout the dissertation an attempt is made to, as much as possible, justify the decisions taken by making references to both the wider community and the Software Engineering literature.

Keywords - periodic task, periodic job, job scheduler.

Acknowledgements

I would like to express my gratitude towards Dr. Dimitrios Pezaros for agreeing to supervise this project and for his invaluable advice and feedback throughout the academic year.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name:	Georgi Koyrushki	Signature:	G. Koyrushki

Contents

Chapter 1

Introduction & Background

In the Software Engineering industry nowadays nearly every medium-to-large-scale application requires the periodic execution of background tasks that run independently from the main system logic. For example, the web crawler component in search engines needs to regularly browse the World Wide Web in order to update the indices the engine uses to answer user queries [86]. Moreover, the users of many e-commerce platforms (e.g. Amazon, eBay, Shopify etc.) are provided with the option to be sent daily sales reports. They choose a time they want to receive the report and it is then the system's responsibility to generate it and make it available to the user in a timely manner. Event-reminder notifications in social media platforms, such as Facebook, are yet another example of an application of background jobs.

There are numerous existent software products available to the wider community that provide the functionality to execute periodic tasks. Each of them has a different architecture and is more appropriate in certain scenarios than its alternatives. The generic name of such a system is **job scheduler** (due to the application goal it needs to achieve - to schedule a job which is due for execution). This Level 4 Software Engineering project takes a look at a specific context in which a scheduler might be used - large organizations consisting of multiple sub-units (e.g. teams in a software company) that need the facilities provided by such a system. During an industrial placement I did last year I noticed that in a large company there are multiple teams that make use of a job scheduler to automate some parts of their workflow. Even more interestingly, there was not a single such service that was utilized by everyone. Rather, there was a tendency among teams to build one on their own. It was acknowledged that a single, multi-tenant scheduler that can be adopted by everyone would allow teams to focus efforts on developing their applications (as opposed to each building and maintaining a service that is used so commonly). Given this, the purpose of this project was to develop a job scheduler that is suitable for use in large, multi-tenant ecosystems. In such environments, the number of periodic tasks is expected to be large and the need for timely execution - essential. This hinted that the end product would likely have to take advantage of distributed computing as otherwise these demands might not be met. This, in fact, was one of the primary reasons why I took up this project - I had recently found a fascination with the field of Distributed Systems and the challenges it offers (e.g. synchronization of components, fault tolerance, etc.). As a result, I felt that building a scheduler, conforming to the aforementioned use case, would enable me to both fulfill a need in the industry, and even more importantly, allow me to enhance my working knowledge in the area of distributed computing.

The rest of this chapter provides an introduction to the project's subject matter. Firstly, a more in-depth clarification of what a job scheduler is and how it generally operates is given[1]. Consequently, the architectures of 2 schedulers available to the community are reviewed and critiqued in the context described in the previous paragraph. This is done in order to give the reader a wider understanding of the problems the project tries to

[1]It is important to note that this explanation is not based on scientific resources, such as research papers or textbooks. Rather, it is purely based on a research that was conducted at the beginning of the academic year into the workings of such a system by examining the architectures of several solutions.

solve. Finally, the chapter introduces the proposed system, along with its aims and objectives.

1.1 Job Scheduler

A job scheduler is an application that triggers the execution of background tasks (also referred to as jobs in this paper). A job is a unit of work that is scheduled for non-interactive execution by a scheduler. Usually, users of such system, provide as input the specification of a job. This includes (among other things) the actual executable to be ran by the scheduler, the ETA[2], and if the task is periodic - recurrence information (e.g. the interval between 2 executions). The system then takes responsibility for timing the execution of the job to match the ETA and (if applicable) the periodic interval.

A job scheduler is a computer process that operates based on ticks. A **tick** is a point in time when the scheduler wakes up and performs the generic steps described below:

1. The scheduler notes the current system time - t_{curr}.
2. It triggers the execution of all tasks that are due. This is achieved by comparing the time of each task - $t_{trigger}$ with the current system time taken in the first step. If $t_{trigger}$ is less than or equal to t_{curr}, then the task is due.
3. The scheduler updates the trigger times of the tasks in the previous step. For example, if a task's trigger time was 10:00 am, and it is executed every 5 minutes, its new trigger time will be 10:05.
4. The scheduler then determines the next time it should work - t_n. This is achieved by taking the minimum trigger time from the collection of tasks.
5. The scheduler sleeps until t_n (i.e. until the next time it should perform a tick).

1.2 Related Products

There is a myriad of schedulers available to the community - [94] [28] [17], all of which provide the functionality to execute tasks periodically. They all adopt different architectures that make them appropriate for certain use cases. Based on some research conducted, however, they all fall into 2 broad categories, differing by the design decisions chosen by the scheduler in question. First, there are systems that schedule periodic tasks for execution on the same node (as the one they run on) and usually achieve this either by creating new threads within the scheduler [94] [63] or by spawning new processes [28]. The second category consists of schedulers that, instead of being responsible for executing a given periodic job themselves, use message passing to notify an executor component [17]. The latter is then responsible for carrying out the application logic associated with the task. In other words, in this case job triggering is decoupled from job execution through a workflow that looks very similar to the Producer-Consumer model [75].

This report takes an in-depth look at 2 open-source schedulers, each representative of one of the 2 categories listed above. The primary reason for this is that due to their different architectures, they provide the reader with a broader context of the domain, and guide comparative evaluation of the proposed system in Chapter 5.

[2]Estimated Time of Arrival - the first time a task is triggered.

1.2.1 cron

cron is a popular software utility present in Unix-like operating systems. Task specifications are stored in a configuration file, which on system boot is read by **crond** - the cron daemon [110, Section 13.1]. An entry in the configuration file would consist of a shell command to execute, along with a recurrence string. The latter is called **cron expression** and has become the de facto representation of periodic intervals in job schedulers (including the proposed system). An example entry in a cron configuration file is shown below.

```
┌───────────── every 2nd minute (e.g. 2, 4, 6, ...)
│ ┌───────────── of every 5th hour (e.g. 5:02, 5:04, ..., 10:02, ...)
│ │ ┌───────────── of every day of the month (1-31)
│ │ │ ┌───────────── of every month of the year (1-12)
│ │ │ │ ┌───────────── of every day of the week (0-6) (Sunday to Saturday)
│ │ │ │ │
│ │ │ │ │
│ │ │ │ │
2 5 * * *  /home/Georgi/generate_sales_report.py
```

Figure 1.1: Example entry in a cron configuration file.

Execution of jobs is achieved by spawning a new process in which the command in question is ran. In other words, cron uses as execution unit the same node where the task specifications are stored, and is thus representative of the first category of schedulers described above. This makes the model simple to reason about and is appropriate for tasks that need to interact with the local file system. An example of this is a job which periodically checks if a directory contains more than certain amount of data, and if it does - deletes the oldest files to free up space. There are, however, some key drawbacks that make cron inappropriate for use in software organizations where the number of such periodic tasks is large and some of them do much more than simply access the file system on the local node. Those disadvantages are listed below:

- If the job needs to interface with an external system (e.g. a database), it needs to set up and tear down connections every time it is executed. This could be seen as a problem if there is a high number of tasks that execute regularly, all of which need to interact with the same external resource. For instance, all jobs that need to generate the daily sales reports in an e-commerce platform will likely query the same database server. In that case, the overhead of connection management would be paid every time. This is wasteful given that a pool of connections could be maintained and queried by each job at the beginning of its execution [106, p. 80-84]. Unfortunately, cron does not lend itself to this due to the way it is designed. Since new process is spawned for every task, the only way connection reusing could be achieved is if a pool is maintained by crond (the parent process). This pool can then be accessed by each job as a child attribute. This is inefficient since the pool needs to be accessed by processes (i.e. spawned tasks) that are running concurrently. This means that there needs to be mechanisms for inter-process synchronization as if 2 tasks access the same connection, race conditions could arise. Although modern operating systems, such as Linux, provide such mechanisms - e.g. Posix Semaphores [66, Chapter 53], it is needless to say that the resulting architecture will be extremely hard to reason about. The perceived reason for this complexity is that with cron, task triggering is very tightly coupled to task execution. Not only do both activities happen on the same node, but also the 2 processes - crond and the running task, are in a parent-child relationship. This made cron inflexible for the task at hand and it was concluded that changing its architecture to accommodate for reusing connections would be impractical. In the Evaluation chapter (Section 5.2) where cron and the proposed system are compared, the significance of the connection establishment overheads becomes more apparent.
- As was mentioned earlier, the job specifications are stored on the file system of the node. This means that

if the machine goes down for good, all the specifications will be lost. In a system where the number of periodic tasks is large, recovering from such a failure could be impractical.

- In order for a new task to be inserted (e.g. because a new customer has requested to be sent daily sales reports in an e-commerce platform), the configuration file must be updated and the daemon restarted, so that it can reload the file. This would incur significant overheads with huge files.

1.2.2 Celery

Celery is a software product that is primarily utilized as a distributed task queue[3] for real-time operation, but can also be used for scheduling periodic tasks. It adopts the notion of Enterprise Service Bus (ESB) [20, Chapter 1]. In the context of job schedulers this implies that there are consumers that listen on the bus for tasks. There is also an entity - the scheduler - which has knowledge of when each job is to be ran. When the time comes for **Task A** to be executed, the scheduler will send a message (encapsulating the task attributes) to the ESB, which is then directed to a consumer and acted upon. This could be viewed as an implementation of the Publish-Subscribe pattern with ETA on top [53, Chapter 4]. Equivalently, the resulting architecture is said to be event-driven due to the fact that consumers' actions are driven by events generated by the scheduler [69] [30]. Figures 1.2-1.4 illustrate this workflow.

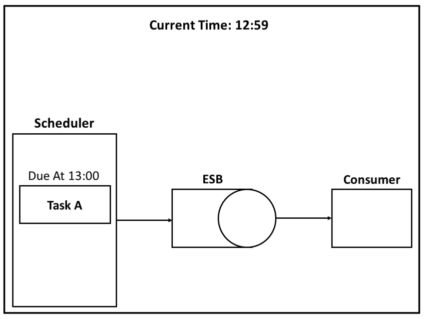


Figure 1.2: A minute before Task A is due.

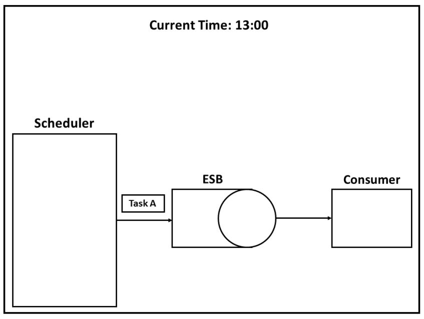


Figure 1.3: Task A is due - Scheduler sends the message to the ESB.

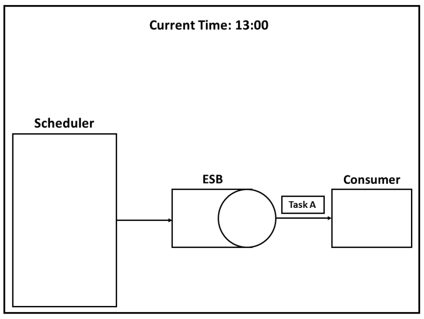


Figure 1.4: Task A is due - the ESB routes the message to the Consumer.

One of the benefits of the ESB is that the scheduler is decoupled from the task execution. Consumers could be placed on different nodes in case a high number of tasks are due at the same time, and thus the load would be balanced across them - a feature which is not present in cron where each task is executed on the machine it is stored on. Additionally, each consumer is a process listening on the ESB for events. This means that connections to any external resources could (if needed) be established only once - when the consumer process is launched, and reused thereafter. This solves yet another problem with cron mentioned earlier.

Even though the above points mean that Celery can be seen as a significant improvement over cron when it comes to scheduling in large-scale systems, it has some disadvantages that might make it unappealing. Those are listed below:

- The scheduler component of Celery is a single-threaded process running on a single node. During a tick, it needs to perform 2 resource-intensive tasks for each periodic job it triggers. First, it determines the next time the job should be ran and updates its data structures to record this value. The reason this calculation is expensive is that the recurrence information (e.g. the cron expression) needs to be parsed, the current system time needs to be accessed, and combining the two - the new trigger time produced. Second, the scheduler needs to send the message associated with the task. These 2 operations happen one after another

[3]A paradigm via which work is distributed across a set of worker threads.

in the same order as they were introduced. The below diagram, depicting how the first calculation scales with increasing the number of tasks, gives intuition as to why this is problematic.

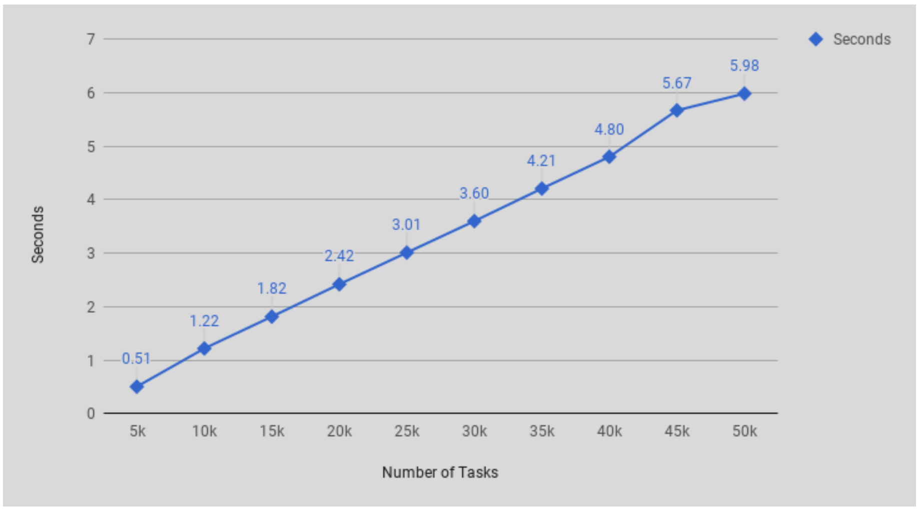


Figure 1.5: Elapsed time against number of tasks.

The duration for calculation next run times for 25k tasks is around 3 seconds and for 50k - nearly twice as much. This means that the second 25k messages will start being sent at least 3 seconds after the time they are due and there will be messages (the last few, to be precise) delayed by approximately 6 seconds. Note that this is the case when the operation that actually sends the message is assumed to have zero overhead, which is not the case in reality. Delays of this magnitude could be unacceptable in some applications. The more important issue with the above design choice is that the 2 operations (calculating next run times and sending messages) are independent from one another and could be executed in parallel. Also, each of them could be further parallelized - i.e. there can be more than 1 thread calculating run times and more than 1 thread sending messages. As a net result of the sub-optimal implementation used in Celery, the overall message sending rate (when the system is saturated) is not satisfactory. The proposed system tries to improve on this. In the Evaluation chapter (Section 5.2.2), it is compared against Celery precisely on this criterion - message sending rate.

- The scheduler reads task specifications from a local configuration file and stores them in convenient data structures in RAM. Throughout its life time, the scheduler will need to update those structures - e.g. it needs to update trigger times, as was explained above. This poses the exact same problem as with cron - if the node where the process runs crashes, all the information that was stored in RAM will be lost. Reading the configuration file if the process is subsequently restarted, would not mean that the system is restored to the state before the crash.

- A corollary problem from the previous one (using RAM instead of persistent storage) is that functionality already built efficiently in existent storage systems needs to be reinvented. For example, in Section 1.1 the generic set of actions performed by a scheduler during a tick were outlined. In the second step, the scheduler needs to get hold of all jobs that are due. This is achieved through finding all tasks whose trigger times are not in the future. Using a traditional RDBMS[4] this can be achieved with a single SQL query. Behind the scenes, the RBDMS will perform query optimization and make sure a good retrieval algorithm on the underlying persistent storage is chosen [58]. Additionally, with the help of database indices, the speed of the operation could be increased even further [42, Chapter 18]. The way the scheduler component of Celery achieves this is using a linear scan through all task specifications. This could slow down the system a lot when the number of tasks is large.

[4]RDMBS - Relational Database Management System.

- Identically to cron, adding a task to the configuration file requires the scheduler component to be restarted. This can bear the same consequences when the file is large.
- While it can be claimed that cron has a primitive form of authentication and authorization (through the file system permissions imposed by the underlying OS), Celery entirely lacks the concept of users. Who submitted **Task A**? Who is allowed to read the messages associated with it off the ESB? These can be deemed concerns for a multi-tenant system and neither is handled by Celery.

1.3 Proposed System

The aims of the project is to design and build a job scheduler that is suitable for integration within a large-scale software system. In such an environment the number of periodic tasks is expected to be large and timeliness of their execution is likely to be important. The perceived way to achieve this is to namely make an attempt to develop a system that provides solutions to the problems outlined in the above sections.

The proposed system (referred to as the Distributed Scheduler in the rest of the document) adopts an architecture similar to that of Celery. The scheduler is an event-emitting component that sends trigger messages onto the ESB, which are then directed to the appropriate consumers. The system tries to overcome the disadvantages of cron and Celery mentioned above in the following ways:

- It uses a reliable data store for maintaining task specifications. This entity is not necessarily located on the same node as the scheduler and written data can also be replicated (out-of-the-box with some RDMSs [90] [80]). This additionally entails a powerful querying language so that coming up with algorithms for complex computations, such as retrieving all due tasks, is deferred to the database.
- An approach is taken that tries to balance the workload across a set of replicated scheduler instances. Additionally, each replica further tries to parallelize its actions, as was outlined in the previous section. This is done in the hope that the cluster of schedulers will achieve a higher sending rate than when only a single instance is deployed. The success of this approach is reviewed in the Evaluation chapter.
- The Distributed Scheduler is built with adding new task specifications in mind. In other words, the proposed system tries to avoid the overheads of restarting the components every time a new task is added.
- Each task is associated with a user. Thus, just as in any multi-tenant system, users are isolated from each other through authentication and authorization mechanisms.
- The Distributed Scheduler tries to adopt fault tolerance mechanisms that make the system robust against different kinds of failure common in distributed systems (e.g process failures, node failures, etc.).

1.4 Chapter Summary

This chapter began by introducing the concept of a job scheduler and gave the reader some intuition as to why such system would be useful in the industry. It then outlined and critiqued the architecture of 2 related products in the context of large-scale applications. Finally, it listed the aims and objectives of the proposed system.

The rest of the document is structured as follows: The next chapter lists the Distributed Scheduler's Requirements. Chapter 3 discusses the design of the system from a top-down perspective. Chapter 4 focuses on the system's implementation. Chapter 5 outlines how evaluation of the system was carried out. Chapter 6 is a conclusion that summarizes the project's achievements and learning outcomes, and proposes future work that can be carried out.

Chapter 2

Requirements

The very first stage in nearly every customer-based Software Engineering undertaking is Requirements Elicitation [108, Chapter 4]. However, since the project was self-defined, conducting this was not necessary. Instead, the idea of the Distributed Scheduler was communicated to the project supervisor at the beginning of the academic year. Once the level of understanding between the 2 parties on the proposed system reached satisfactory levels, the requirements were explicitly defined and prioritized.

2.1 Prioritization Technique

The MoSCoW technique was used in prioritizing the requirements of the system [111, Chapter 16]. The reason for employing it was that it had proven to be particularly useful in last year's TP3 course. Then, at the beginning of the second semester, the client came up with a new set of requirements. This was because the project was nearly complete at that very early stage. However, not all of those requirements were achievable in the remaining time. MoSCoW was used alongside the customer in assigning a level of priority to each. This resulted in the most important ones being successfully delivered at the end of the project.

MoSCoW acknowledges that all requirements are important, but splits them into 4 categories, so as to ensure that the most beneficial ones are delivered first. Those categories are listed below:

1. **Must have** - requirements which are critical in order for the system to be labelled MVP[1].
2. **Should have** - requirements which are important (some could be even as important as must have), but do not need to be present in the current delivery time box.
3. **Could have** - desirable, but not necessary requirements.
4. **Won't have (this time)** - the least critical requirements - they are not even considered for the current delivery time box.

The next 2 sections list the functional and non-functional requirements of the Distributed Scheduler, prioritized using the MoSCoW planning technique. Additionally, keywords from RFC 2119 are used to indicate requirement level [14].

[1]MVP - Minimum Viable Product.

2.2 Functional Requirements

2.2.1 Must have

The single must have functional requirement is the **scheduler service**. This is the entity which was thoroughly discussed in the first chapter. The scheduler (as outlined in Section 1.3) **must** implement a tick by reading all tasks that are due from a database, sending messages onto an ESB, and synchronizing the newly computed trigger times into the database. It was acknowledged that it is the most important component of the system from the outset and that the project would not be complete without it.

2.2.2 Should have

The list of should have requirements comprises different features in an API[2] to the scheduler's database (referred to as the System API in the rest of the document). This, in essence, **should** be the component through which users interact with the system. The System API **should** provide a level of abstraction from the database so that the latter is not accessed directly. With this component in place, the system could be viewed as an implementation of the Producer-Consumer model [75] where the Producer is the API, the Consumer is the scheduler, and both components interact via the database. Note that this sounds very similar to a famous Software Anti-Pattern - Database-as-IPC where a database is used as a message queue for routine IPC[3] [31] [103]. However, this is not the case in the Distributed Scheduler's architecture since the database is primarily used for persistent storage of job data - something discussed in Section 1.3. The list of the different System API functions follows:

- **New task submission** - there **should** be an API endpoint through which task specifications are inserted into the scheduler's database.
- **Updating existent tasks** - for example, changing the recurrence information (e.g. from "every 5 minutes" to "every 1 hour"), changing the message payload, disabling the task, etc.
- **Viewing task details** - an API endpoint through which the attributes of a given job can be obtained. This **should** include: last trigger time, next trigger time, total number of triggers, task identifier, etc.
- **User authentication and authorization** - to implement a multi-tenant system. As it was discussed in the Introduction chapter, in a software organization, it might be the case that more than 1 teams need access to a single deployment of the Distributed Scheduler. In that case, even though the environment is expectedly friendly, 2 different teams **should** be isolated from accessing the same resources as this could lead to effects whose cause is unpredictable. For example, 2 consumers from a RabbitMQ[4] cluster could unwittingly listen on the same queue. In that case, it would appear to the consumers that some messages are being lost, while it is the other that receives them. The way this was implemented is discussed in Section 4.3.4.

2.2.3 Could have

There were 2 functional requirements conceived to fall into the could have category. They are listed below:

[2]API - Application Programming Interface.
[3]IPC - Inter Process Communication.
[4]The concrete ESB implementation used in the project. It is mentioned here for the sake of the example. It will be later discussed in greater detail.

- **Web UI** - there **could** be a web application that interfaces with the System API to provide an alternative, non-programatic way to access the system. Most websites nowadays are built following similar architecture. There is a RESTful API that speaks mostly lightweight communication protocols - e.g. JSON, so that the server side is not overwhelmed by HTML rendering [35]. On the client side, a front-end framework (e.g. React [97] or AngularJS [6]) is used that queries the API for data and dynamically builds HTML which is injected into the DOM [5]. This type of web applications are referred to as Single Page Applications and have become very popular [76].
- **Scaling up the system** - it **could** be desirable to be able to add more scheduler replicas through a simple interface so as to match an increased demand in the system.

2.2.4 Won't have

Initially, a way of **calculating per-user utilization** of the system was considered. This could have been achieved by, for instance, measuring the number of triggered tasks per user per month. The metrics could have been used in an official deployment of the system for the creation of a pricing model (this is done, for example, in [4] and [87]). Alternatively, administrators in organizations could impose quotas on different teams so that the resources are equally allocated. This requirement was dropped, however, as the scope of the project would have become unmanageable.

2.3 Non-Functional Requirements

2.3.1 Must have

- **Replication of Schedulers** - the scheduler service **must** be built in such a way that it can be replicated across a cluster of machines. The challenge here is to synchronize the different replicas so that they access disjoint portions of the task data.
- **Fault tolerance** - the system **must** survive crashes. For example, if a node where a scheduler runs goes down, that replica must be restarted on another machine. For this to make any sense, it is important that the scheduler service is stateless [102, Chapter 8] - i.e. it does not maintain data persistently, but for as little time as possible. If that was not the case (as in Celery's case, for example - refer to Section 1.2.2), restarting a crashed replica would not change the fact that all the data is lost.

2.3.2 Should have

Data consistency should be ensured in the system if the API functional requirement is implemented. In that case, there will be 2 entities writing to the database. It **should** be ensured that the scheduler service and the System API do not step on each other's toes and the appropriate synchronization mechanisms are in place.

2.3.3 Could have

It **could** be useful to **automate instrumentation**[6] of the system. It needs to be evaluated in any case, so building it with this in mind could save time.

[5]DOM - Document Object Model.

[6]Instrumentation is gathering performance measures of a given system as it runs.

2.3.4 Won't have

During first 2 weeks of the project it was researched if the system could be made **real-time**. In the context of the Distributed Scheduler this meant placing certain constraints on the system's performance. For example, would it be possible to ensure that a message is sent no later than 500 ms after it is due? The reason this requirement went into the won't have category is that the distributed system being built is asynchronous and components are interacting over an unpredictable network [24, Section 2.4]. For example, there is no way to formally predict how long it will take a scheduler replica to query the database. Additionally, at the end of a tick, the scheduler sleeps until the earliest trigger time, as was mentioned in Section 1.1. General-purpose operating systems, such as Linux, cannot guarantee timeliness of the sleep operation [72, p. 354]. Several RTOSes[7] were researched, but all had limited-to-no support for a lot of useful tools, and as a result using an RTOS was considered to fall outwith the scope of the project.

2.4 Chapter Summary

This chapter discussed the Requirements of the Distributed Scheduler. It started by outlining the requirements prioritization technique adopted in the project, which was consequently used to list the functional and non-functional requirements of the system.

Before moving onto the Design chapter, it is worth mentioning that, apart from the won't have requirements, the Web UI was also dropped from the list of deliverables at the beginning of the second semester. After a discussion with the project supervisor, it was agreed that without it the project would still have large enough scope. Moreover, the feature was not critical as it would only provide a more human-friendly way of interacting with the system. At the same time, it involved high risk as new technologies had to be researched, and in general, trying to implement it could have taken unreasonable amount of time.

[7]Real Time Operating System.

Chapter 3

Design

Design of the Distributed Scheduler started early in the development process. According to Storer there are 2 general approaches to tackling design problems in Software Engineering - top-down refinement and bottom-up composition [111, Section 21.1.2]. The former tries to first envision the overall architecture of the system. The output is a set of diagrams, depicting an abstract view of the product, which is proven to meet the high-level specification (i.e. the requirements). On the other hand, the bottom-up approach to design aims to initially address problems that are concerned with the actual implementation - e.g. researching existent frameworks and libraries that are suitable for the project. It is also usually the case that some software components are built at an early stage and orchestrated into larger sub-systems before a higher-level architecture is drawn.

The two approaches are not necessarily mutually exclusive and are sometimes used to complement each other. Even so, the one employed in this project (at least at the very beginning) was top-down refinement. The primary reason for this was that experiences from past projects have shown that an abstract view of the target system, with the different components clearly identified, makes it easier to later implement each in isolation, while keeping focused on its responsibilities and providing stubs[1] for where external dependencies are yet to be integrated.

3.1 Design Principles

Throughout the designing stage, the principles prescribed by Storer in [111, Section 21.2] were strictly followed so that the resultant architecture conforms to the best Software Engineering practices. Those principles are summarized below:

- **Separation of Concerns** - the designer should try to identify self-contained reusable components whose individual concerns can be addressed in isolation.
- **Maintaining Sufficient Abstraction** - this allows for grasping the essence of the system without having to know unnecessary details.
- **Reuse and Reusability** - building the components to be as generic as possible, while reusing existing designs, frameworks and libraries (i.e. not reinventing the wheel). At the same time, the designer should be wary of introducing anti-patterns, such as the Inner-Platform Effect [65] [55].

[1]A stub is a piece of code used to replace some other programming functionality.

- **Design for Testing** - a good design should guide the respective implementation to be testable. For example, if a component needs to interact with an external resource (e.g. a database), it should be possible to mock the behaviour of the latter so that the component in question can be easily unit tested.
- **High Cohesion** - keeping related parts together and separating them from other parts of the system. If this principle is not in place, the resultant system becomes very hard to reason about.
- **Loose Coupling** - keeping dependencies between different components of the system to a minimum. This implies that parts of the system could be extended or replaced without this affecting other sub-systems. High Cohesion and Loose Coupling are often deemed to be the most important principles when designing software products.

3.2 System Use Case Model

Use Case modelling is fairly often used in the design phase of a software project to give a simple overview of the interaction between a system and its users. It is useful because it explicitly states the set of functions that should be present in a product. Use Case modelling is usually achieved through UML Use Case Diagrams (UCD) [108, Section 5.2.1]. Before breaking up the Distributed Scheduler into its constituent components, a UCD that clearly defines the system's boundaries and distinguishes it from the surrounding environment is reviewed. This diagram is shown below:

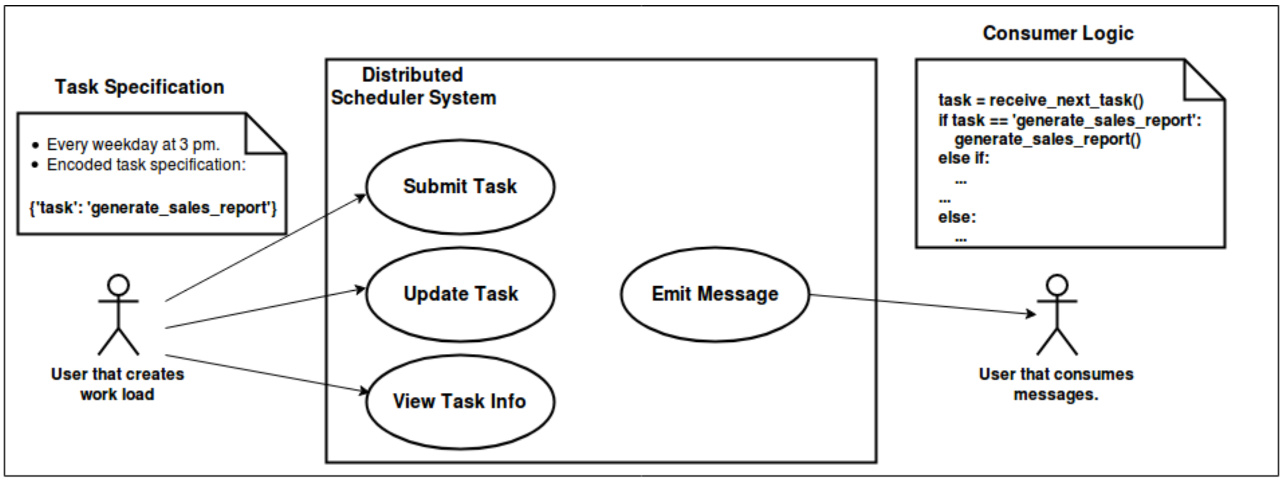

Task Specification
Every weekday at 3 pm.
Encoded task specification:
{'task': 'generate_sales_report'}
User that creates work load
Distributed Scheduler System
Submit Task
Update Task
View Task Info
Emit Message
Consumer Logic
task = receive_next_task()
if task == 'generate_sales_report':
generate_sales_report()
else if:
...
...
else:
...
User that consumes messages.

Figure 3.1: System Use Case Diagram.

The diagram clearly shows that the system has 2 points of interaction with its users. On the left-hand side, there is the actor that creates workload by submitting tasks to the system. A sample task specification is also shown on the diagram. Its first attribute is the recurrence information. This was discussed in Chapter 1 and defines the moments in time when a given task is to be triggered. Also, the purpose of the task is specified. In this example it is encoded in JSON and signifies the generation of a sales report. Other possible interactions this actor has with the system is to update tasks and view task information (functional requirements from Section 2.2.2).

On the right-hand side, a consumer of the scheduler's events is depicted. Here, this is a Python script that continuously listens for messages and acts upon their receival. A concrete choice of technology (Python) was made just to guide the explanation of the diagram. In reality, the consumer could be written in any language that has a client library to the particular ESB deployment (RabbitMQ in the case of the Distributed Scheduler). Additionally, for the sake of simplicity of the diagram, there is only one consumer listening for triggers. As was

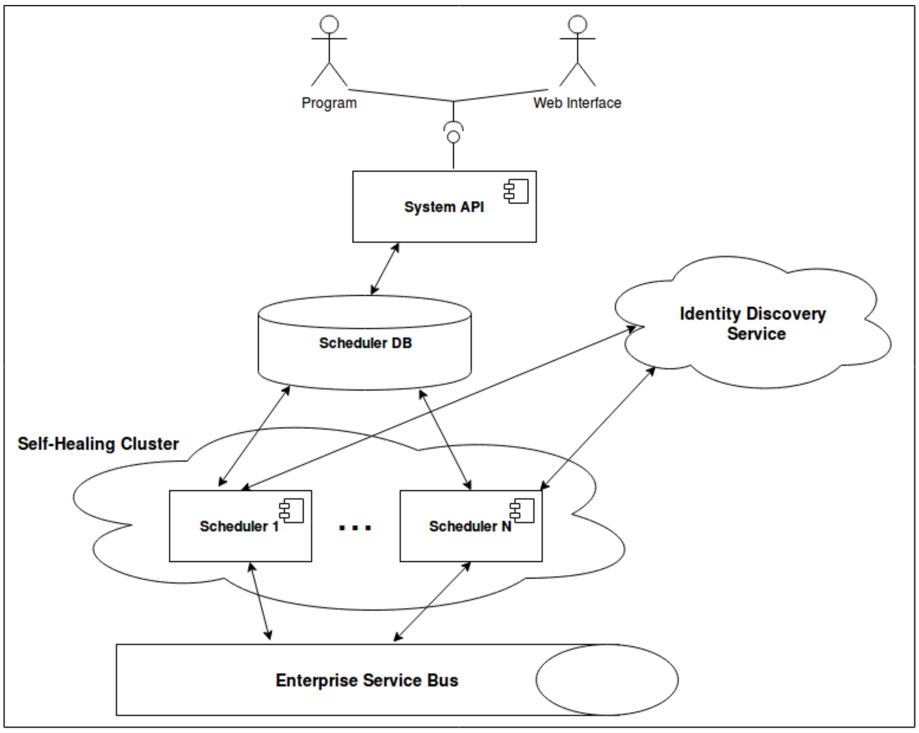


Figure 3.2: Top-Down Architecture Diagram.

discussed in Chapter 1, multiple such entities can be deployed on different nodes. A user might do that so as to improve the throughput of task execution if one node is not enough to appropriately handle the load.

3.3 Top-Down System Architecture

This section identifies the key components in the system, discusses the purpose of each and comments on how they interact with one another. Figure 3.2 is used to guide the explanation. It is important to note that not all of the components on the diagram have been developed as part of the project, but have, rather, been employed as third-party software tools. In fact, the only 2 that are deliverables of this project are the System API and the replicable scheduler. Regardless, the rest are discussed in the dissertation because they provide useful services without which the system would be incomplete.

3.3.1 System API

This is the API to the system that was outlined in Section 2.2.2. It exposes an interface that is used either programatically or through a web application to add multi-tenancy to the Distributed Scheduler. The System API is the component through which users submit task specifications to the scheduler's database. It is agnostic of the scheduler components in that it does not need to know the nodes on which they run or how they operate. The System API does need to be aware of the number of schedulers in the system (N on the diagram). It uses this information so that it can equally load balance task responsibility amongst the schedulers using a simple mechanism. Whenever a new task is submitted by a user, the System API associates an integer value with it - one in the closed interval [1, N]. This value is the identifier of the scheduler instance that will be responsible for that task. The concrete load balancing strategy is Round-Robin [107, Section 6.3.4], as it will become apparent when the implementation of the system is discussed.

3.3.2 Identity Discovery Service

In the previous paragraph, it was implied that each scheduler replica has an associated identifier (ID) which it uses to determine the set of tasks it is responsible for. 2 ways of assigning IDs to a replica were considered. In the first one, IDs are hard-coded in each scheduler's source code, prior to build-time. This was deemed an undesirable solution as the scheduler instances would not really be replicas (since they would differ by only one attribute - the hard-coded ID). It is worth noting (with the risk of going into too much detail on the implementation) that this could subsequently lead to placing too much memory load on the nodes in the cluster where the system would be deployed. This is because N scheduler instances would result in N different Docker images, all of which could end up being downloaded by a node in the cluster (e.g. if all nodes but one crash, that node will have to accommodate all replicas that have been deployed). The other way employed an Identity Discovery Service that assigns IDs to schedulers at run-time. This did not suffer from the above problem as there would be a single Docker image for all scheduler instances. Thus, it was considered to be a superior approach to the first one and was adopted in the design. An explanation of the concrete implementation and further justification of why this approach was chosen is provided in Section 4.2.3.

3.3.3 Replicated Scheduler

Figure 3.2 also shows the stateless replicated scheduler instances and how they interact with the rest of the system. The very first step a scheduler replica takes when it is launched is to acquire a unique ID through the Identity Discovery Service. From that point onwards, it begins performing tick operations, as was discussed in Section 1.1. In addition to this, there are 2 invariants concerning the set of scheduler instances that need to be present in the system. First, each replica is responsible for a **disjoint set of all tasks in the system** (otherwise messages might be duplicated). This is achieved by using the ID of a given scheduler as part of the criteria which it uses to query the database. Second, **the union of the per-scheduler sets of tasks, must equal all tasks in the system** (otherwise some messages will not be sent at all). This will hold, provided that the System API does not assign to a task an ID which is not in the closed integer interval [1, N]. When a task is due, the scheduler instance responsible for it will send the task message onto the ESB. The scheduler will only ensure that the task is delivered to the appropriate endpoint. How it is acted upon by the receiver is part of the logic of the user application.

3.3.4 Self-Healing Cluster

The scheduler replicas are ephemeral processes that are deployed onto a cluster that automatically monitors its state. This means that if the node where a scheduler instance runs crashes, this will be noticed and acted upon by the cluster, without the manual intervention from the administrators. This would result in starting a new replica on an available node in the cluster. Since the scheduler is a stateless process, this would mean that it can resume its work from where it was interrupted by the failure. Some of the technologies available to the community that achieve this are outlined in Section 4.4.1, where also a justification of the particular choice made is provided.

3.3.5 Others

The 2 components that have not been explicitly discussed yet are the scheduler's database and the Enterprise Service Bus. The former is where the task specifications are stored. It is used by the System API - to submit, update and view task specifications. In addition, the scheduler replicas use the database to read all due entries and update their trigger information while performing a tick. The ESB is the message broker that receives task messages sent by the scheduler and delivers them to the designated end-points.

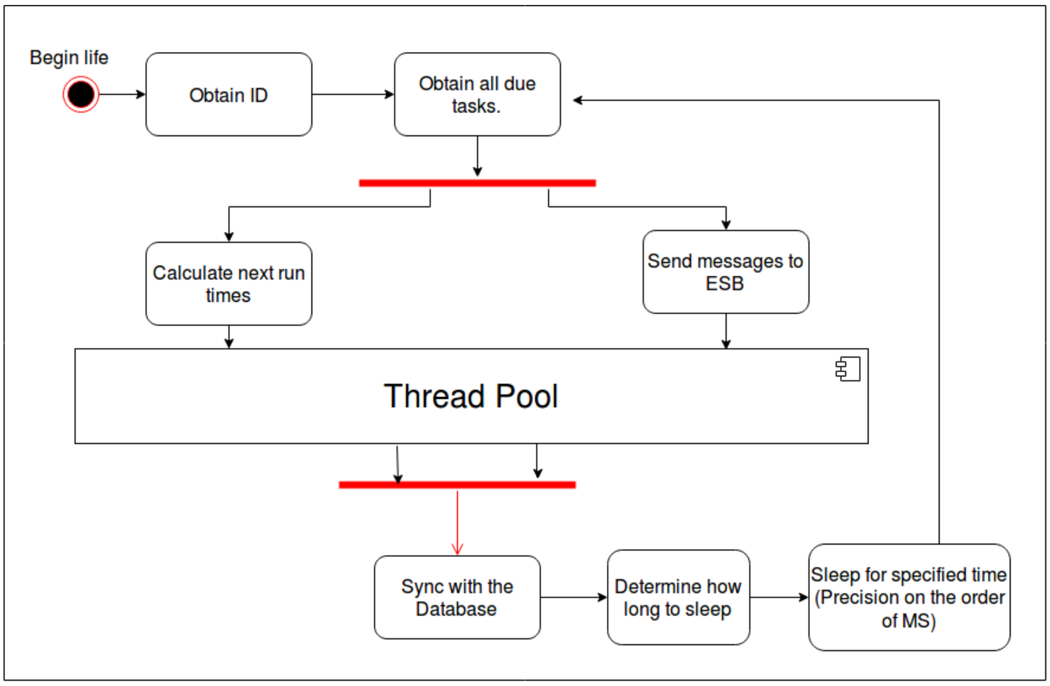


Figure 3.3: Scheduler Life Cycle Activity Diagram.

3.4 Scheduler Life Cycle

The scheduler is the key service in this project and because of this a quick view is taken at how each replica operates as a final section of this chapter. Figure 3.3 is an activity diagram that shows how the generic steps in Section 1.1 are implemented. The scheduler first obtains its unique identifier through the Identity Discovery Service. It then enters the loop where it performs ticks operations. On a scheduler tick, all tasks that are due are transferred from the database to the memory address space of the scheduler. The activity flow then is split into the 2 independent operations. Those were described in detail in Section 1.2.2 where the disadvantages of Celery were pointed out. On one hand, calculation of next trigger times for the due tasks is carried out (refer to Section 1.1). On the other, messages are sent onto the ESB which will route them to the correct endpoints. Each of those 2 operations can be further parallelized onto multiple execution streams. For example, more than 1 thread can be tasked to compute next run times. Identical is the case with sending messages. It is important to note that the latter can execute in parallel if and only if the hosting node has more than one physical network interface cards (NIC) [82]. Otherwise, the different threads will all send the messages through the same NIC, and there would be no benefits to parallelizing that operation. As the next activity, the scheduler performs a sync to the database, which means the updated run times are written back. Finally, it computes the earliest time in the future at which a task will be due again, and sleeps until then.

3.5 Chapter Summary

Chapter 3 of the dissertation went through the design of the Distributed Scheduler. It first explained the design strategies and principles that were adopted. Subsequently, it discussed the top-down architecture of the system by outlining all participating components. In the end, it gave an overview of the life cycle of a scheduler replica.

Chapter 4 concerns the implementation of the Distributed Scheduler. It will try to concretize the abstract concepts outlined in this chapter and will also discuss some of the challenges faced while building the system.

Chapter 4

Implementation

The implementation of the Distributed Scheduler was the most involved part of this Level 4 project. This chapter begins with a section outlining the key Software Engineering practices that guided and facilitated the development of the system. Afterwards, the reader is provided with a detailed technical explanation of the 3 milestones that were reached during the implementation stage, split into 3 sections. Each includes an outline of the technologies used, a breakdown of the key achievements, and a description of the main issues that were faced and how they were handled.

4.1 Software Engineering Practices

In the Software Engineering field Agile is an approach to project management that emphasizes adaptive planning, incremental development, continuous improvement and early delivery of small pieces of functionality [109, Chapter 2]. Agile is seen as an alternative to classic waterfall models of software development where work on the fundamental process activities (e.g. specification, development, validation, etc.) is carried out in a linear fashion and it is assumed that it is impractical to revisit a previous stage (e.g. changing the design of a system due to a realization reached by the developers during implementation) [108, Section 2.1.1]. With waterfall models, all details in the development life cycle must be carefully considered up-front as change due to poor planning is not well regarded. While this approach is suitable when project requirements are well-defined and understood (e.g. in other Engineering areas), it is hardly ever practical for Software Engineering undertakings. This is primarily because software projects are almost always characterized by uncertainty in requirements and high risk of delivering the wrong product - one of the most common reasons for project failures [18]. Due to these limitations of waterfall models, Agile is considered to be the preferred approach to managing processes in a Software Engineering project.

Agile is a methodology that is usually adopted inside a team of engineers. Admittedly, if a look is taken both at the Agile manifesto [73] and the Agile Software Development Principles [91], the ”team” is viewed as the focal point of project activities. Because of this, working on the Distributed Scheduler did not fully lend itself to Agile. To clarify, some Agile frameworks, such as Extreme Programming and SCRUM encapsulate techniques that could not be applied to an individual project - e.g. Pair Programming, SCRUM Retrospectives, etc. [108, Section 3.3] [109, Chapter 4]. Regardless of this, a great deal of effort was put into adopting as much Agile techniques that did not focus on team work as possible. This lead to a structured and transparent development process, which improved the quality of the end product. These techniques are described in greater detail in the paragraphs below.

It was decided from the outset that work on the Distributed Scheduler would be split into several equal units of development, each expected to deliver a set of functions that can be considered a whole. SCRUM refers to these partitions of the work on a software project as **Sprints** [120]. Initially, it was planned that there would be 4 Sprints for the project, each spanning about a month and a half (leaving around a month and a half to produce the dissertation). Those would focus, respectively, on the scheduler service, the System API, the Web UI and deployment of the system. However, after finishing the first 2, it was acknowledged that working on the Web UI might pose a risk to the successful completion of the project and, at the same time, the feature would not bring much value to the end product (as was discussed at the end of Chapter 2). As a result, the Sprint was cancelled and work continued on deployment of the system. In-depth analysis of the 3 completed Sprints will be given in Sections 4.2-4.4. The beginning of each Sprint was preceded with **Sprint Planning** where the **Sprint Backlog** was produced [121] [119]. This would consist of the tasks that had to be successfully completed before the deadline. The Sprint Backlog was documented in GitLab [46] as GitLab issues. Each issue had (among other) the following attributes:

- A short description of the purpose of the issue.
- Several labels signifying the intrinsic properties of the issue (e.g. bug-fix, feature, enhancement, etc.).
- A weight denoting how important the completion of the issue is. This prioritization technique allowed for specifying the order in which issues would be completed and proved to be an extremely efficient way of managing time while working on the project. Note that this is unlike the more widely used task cost estimation, which tries to determine how long an issue would take [21, Chapter 8].
- An association with a given Sprint. Note that Sprints are called milestones in GitLab.

In addition to SCRUM, several techniques were taken from the Extreme Programming framework. For example, at the end of each Sprint, new code would be subject to **Code Reviews**. On numerous occasions, this lead to **Refactoring** sessions that resulted in more readable and higher quality code [104, Chapter 9].

To conclude this section, every single piece of functionality was written with testing in mind. A very famous Extreme Programming technique is Test-Driven Development (TDD), where writing tests precedes the development of the underlying functionality [108, Section 8.2]. This has been proven to lead to better quality software [57]. Although TDD was not employed in the project in its strictest form (because it requires a lot of planning and is usually time-consuming), both **Unit and Integration testing** would be carried out after the set of issues for a particular Sprint was completed. In order to reduce the overheads of the process, testing focused mainly on the API of the module under development. In other words, if it was verified that the API works correctly, it would be assumed that the underlying utilities also behave as expected. A more in-depth explanation of how testing was conducted is given in Section 5.1.

4.2 Scheduler

4.2.1 Choice of Technologies

It was particularly important that the right set of technologies were picked for the development of the scheduler service since it was the most important deliverable of the project (refer to Section 2.2.1). Additionally, as it was shown in Figure 3.2, the scheduler replicas are the components in the system that interact with the largest number of external entities. This, combined with the fact that building the scheduler was the task of the first Sprint, meant that a lot of very important decisions had to be taken from the very start.

To begin with, a suitable **programming language** (PL) had to be chosen as the technology that would power the scheduler. The circle for consideration was quickly narrowed down to 3 PLs - Java, Python, and C++. This was primarily due to the decent familiarity at the time with the aforementioned and the fact that learning a new PL would probably be time-consuming, and hence risky for the successful completion of the project. The PLs were mainly judged on 4 criteria - performance, built-in support, community engagement, and ease of use. Regarding performance, the 3 PLs were tested on how fast they would carry out one of the main operations the scheduler needs to perform - calculation of next trigger times for a given task. Similarly to Figure 1.5, it was examined how calculation time scales with increasing the number of tasks that need to be updated at the same time. The difference here is that the performance of all 3 PLs were plotted together. The experiments were carried out on a commodity i7-7500U machine and included both a single-threaded and parallel versions of the computation. The results are, respectively, shown in Figures 4.1 and 4.2.

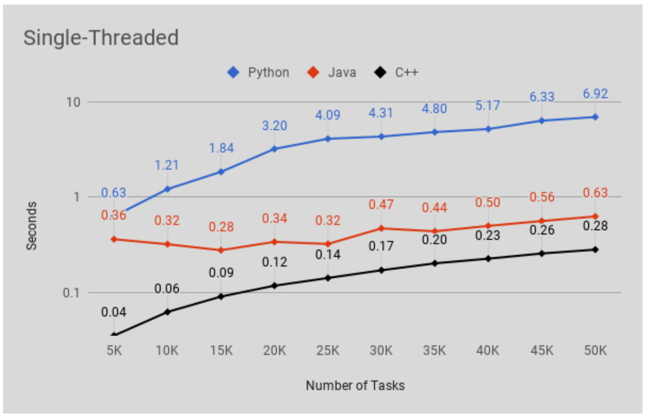


Figure 4.1: PL Performance - Single-Threaded.

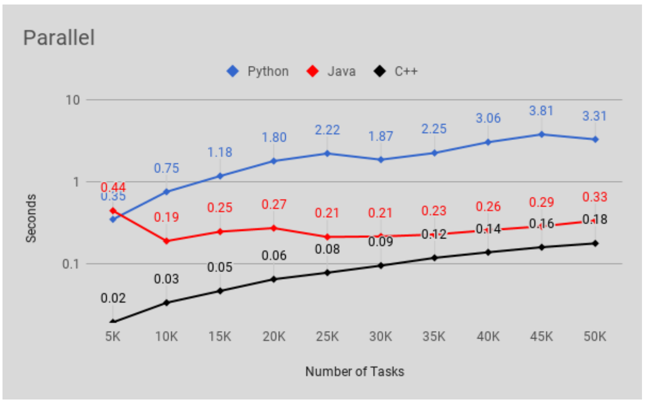


Figure 4.2: PL Performance - Parallel.

The y-axis of both plots uses a logarithmic scale because Python was at least an order of magnitude slower than the other 2 for large number of tasks (e.g. in the range between 20 to 50K). As a result, if a normal scale was used, it would appear as if the difference between Java and C++ is negligible. In addition, in the reference Python implementation - CPython, parallel multi-threading is impossible due to the infamous Global Interpreter Lock (GIL) [64]. Because of this, the smallest execution unit that allows for parallelism is an OS process [107, Chapter 3]. Conveniently for Python developers, there is a module in the Python Standard Library that provides the same API as what one would expect for multi-threading, with the difference that it is based on processes [52, Section 10.4.1]. However, process concurrency incurs overheads due to the expensive (when compared to threads) Inter-Process Communication and Synchronization. All the points mentioned so far can be considered hurdles to performance and since this was the dominating criterion for the choice of PL, it was concluded that Python was not a good option for the project.

In Java's case, it is interesting to note that the curve is not strictly increasing (but has some local maxima, followed by valleys). For example, in the single-threaded experiment, computation time has increased between 15k and 25k tasks, and then decreased between 25k and 30k. This is attributed to the fact that most JVM implementations have a built-in JIT compiler[1] that performs significant optimizations at run-time [114] [12]. Despite the fact that the Java's run-time has improved over the years [11], virtual machine interpretation has almost no chance against native code execution [117, p. 432]. As one would expect, both figures show that C++ completed the tasks faster than Java (at least twice as fast, to be precise). In other words, if performance was solely considered, C++ would be the obvious choice for the scheduler. However, after further research into the language, it became apparent that it falls greatly behind Java and Python on the other 3 criteria which were listed above. For example, at the time when this dissertation was produced, there was not a single C++ standard where the corresponding standard library had a thread pool. Since such a construct was needed for the scheduler (refer to the activity diagram in Section 3.4), the option was either to resort to external libraries (e.g. Boost [13]) that provided not always well documented ways of achieving the desired functionality or to implement a customary thread pool, based on the underlying multithreading facilities provided by the language. Furthermore, many

[1]Just-In Time compilation allows for dynamic compilation of code at run time.

client bindings to the external resources accessed by the scheduler (e.g. PostgreSQL, RabbitMQ, etc.) were poorly documented and lacked community support. Last, but not least, few would argue that Java and Python are much simpler to use than C++. The former alleviate many concerns that need to be explicitly handled in the lower-level C++, the most prominent of which is automatic memory management through garbage collection [2, Section 7.5]. On account of all of this, the PL that was chosen for the scheduler service was Java as it was considered the best trade-off between performance and ease of development.

Another particularly important decision w.r.t. technologies for the scheduler was the choice of concrete **ESB implementation**. The 2 widely used solutions that were considered were RabbitMQ - a traditional message broker [95], and Kafka - a distributed streaming platform, which (among other things) can be used as a message broker [8]. Both are very strong in terms of client libraries and have Java bindings. Furthermore, they are both built with high availability in mind and are fault tolerant in the event of node crashes. Additionally, messages that have been sent to the broker, but not yet delivered to consumers, can be protected through durable resources (e.g. exchanges and queues in RabbitMQ; topics in Kafka) and replication [116, Chapter 5]. RabbitMQ provides complex routing capabilities, which are useful when a message needs to be delivered to more than one consumer in a non-broadcast fashion [116, Section 2.3]. Additionally, messages are deleted from the broker once they are sent to all intended consumers. This conformed to the needs of the scheduler as messages are not needed once they are received by the subscribed endpoints. By contrast, Kafka stores the messages persistently as it makes the assumption that they may be accessed by consumers in the future. RabbitMQ is also self-contained and very easy to install and integrate into a project. On the other hand, Kafka requires other services to be running (e.g. Apache Zookeeper). Taking these points into account, RabbitMQ seemed to be the better option for the scheduler and was consequently adopted as the ESB.

When researching **database technologies**, the very first question that had to be answered was whether an SQL or a NoSQL solution was more appropriate for the project [100, Chapter 1]. During the first weeks of the academic year, it became apparent that there is presence of schema in the data that needs to be kept in the system. Additionally, as it will be discussed in-depth in Section 4.2.6, the transactions that were to be supported had to exhibit the ACID properties [42, Chapter 21]. These 2 points implied that an SQL database would be more appropriate for the project, and the problem was reduced to deciding on a particular RDBMS. PostgreSQL [90] was chosen after very little deliberation. The reasons for this were positive experience from previous projects and the fact that it supports advanced transaction features that were deemed important for the scheduler - e.g. configurable locking granularity that allowed for choosing between table-level locking and row-level locking for specific transactions.

Another technology which was used in the development of the scheduler was Apache Zookeeper [9]. Apache Zookeeper is a replicable coordination service that provides facilities for synchronization of distributed systems. It uses a very intuitive data model that resembles a directory tree in traditional file systems [51, p. 31-32]. The main reason why Zookeeper was used in the project was that it provides the primitives to implement distributed locks. These low-level constructs were utilized to build the Identity Discovery component described in Section 3.3.2. Details on how this is achieved are provided in Section 4.2.3.

In the end, it is worth noting that early development efforts were marked with manual management of project dependencies. In the case of Java, this meant that jar files were downloaded from the Internet (e.g. from the Maven Central Repository) and manually linked into the project's CLASSPATH. This was extremely laborious and time-consuming, especially when transitive dependencies[2] had to be installed as well. In the wider community, this is referred as dependency hell [32]. As a result, it was decided that a **build automation tool** that provided dependency management functionality would be incorporated into the project. Gradle [48] was chosen as it is extremely easy to use, popular IDEs provide support for it, and it is very configurable and extensible.

[2]If package A depends on package B, and package B depends on package C, then package A depends on package C.

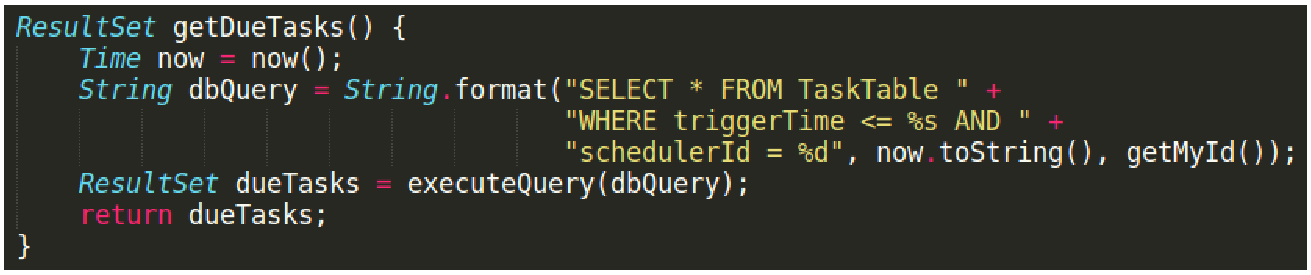
4.2.2 Data Access Layer

One of the first steps in writing the code for the scheduler was to create simple data access layers (DAL) that wrap over the low-level APIs exposed by the client libraries for some of the external resources. This included RabbitMQ and PostgreSQL (a simple and efficient DAL on top of Zookeeper's official API was already available as an open-source package [7]). The effort was needed for several reasons. Firstly, it was estimated that there would be multiple places in the code where, for example, SQL queries would be issued. Since the database server is accessed across a network, connection failures (e.g. due to network partitions and/or temporary unavailability of the database) were something that had to be accounted for. As a result, if there were try/catch blocks in each of those snippets to perform error handling, the code would be bloated and unreadable. The DALs would extract these failure condition checks in a single place and would abstract away the recovery logic from the client code. It should be noted that connection drops are noticed whenever interaction with the service occurs (e.g. when an SQL query is issued). When such a failure is spotted, the DAL logic would try to reconnect to the resource several times and either re-execute the operation that caused the failure (if the previous succeeds) or cause the whole OS process to terminate cleanly. In addition to error handling, there are certain optimizations that could be achieved when accessing both RabbitMQ and PostgreSQL. For example, the JDBC API demands for the creation of **Statement** objects through which the actual data is accessed [61]. Reusal of those objects (as opposed to instantiating new ones every time) in subsequent SQL queries implies less RTTs[3] to the database server, and consequently better performance and network utilization. Similarly, in RabbitMQ's case the scheduler needs to make sure that it has declared certain resources before sending a message (e.g. an exchange, a queue, etc.). This also requires interaction with the broker, and it is recommended that it is done only once for a specific resource. The implementation uses a simple memoization mechanism where before declaring a specific resource, it is ensured that this is not already done in the past [23, Chapter 15]. Last, but not least, connection establishment incurs non-negligible overheads (as it will be discussed in the Evaluation chapter in the context of comparing the Distributed Scheduler with cron). Because of this, connections to both RabbitMQ and PostgreSQL are not closed after every operation, but are rather kept for as long as possible. Since there are multiple threads that access the external resources, a single JDBC **Conenction** object is kept for each. In other words, in addition to the DAL optimizations described above, a very simple resource pool that reuses connections is implemented. All these functions are implemented inside the **uk.ac.gla.dcs.dal** package of the Java project.

4.2.3 Identity Discovery

As was discussed in Section 3.3.2, there had to be a mechanism through which each scheduler replica gets assigned a unique integer identifier in the closed interval between 1 and N (where N is the total number of schedulers deployed in the system). This was needed so that each scheduler can take responsibility for a disjoint subset of all tasks in the system (those that have been assigned to that replica by the System API, to be precise). The way this is achieved is through Apache Zookeeper and the distributed locking facilities it provides [9]. The way locks can be implemented in Zookeeper is very well outlined in [51, p. 98-100] and this will not be done here in order to keep the description concise. The way the ID discovery algorithm works is that each scheduler replica tries to acquire N locks in turn at the very beginning of its life (i.e. when the code in the constructor of the instantiated object is executed). It will eventually manage to acquire some of the locks, say, lock j ($1 \leq j \leq N$). At this point, an invariant that will hold in the system (given Zookeeper's guarantees) is that no other scheduler replica will have acquired (or will acquire) lock j. As a result, j is the ID of the scheduler and it can now start interacting with the database, using j in the queries. A similar technique has been used in the implementation of Google's NoSQL database - BigTable. There, distributed locking through Google's proprietary coordination service - Chubby, is used to provide unique naming for the different storage nodes [19] [16]. To give an example of how the unique ID is used in the system, Figure 4.3 shows simplified Java pseudo code of the query a scheduler replica needs to execute in order to read all due tasks.

[3]Round Trip Time.



```
ResultSet getDueTasks() {
    Time now = now();
    String dbQuery = String.format("SELECT * FROM TaskTable " +
                                   "WHERE triggerTime <= %s AND " +
                                   "schedulerId = %d", now.toString(), getMyId());
    ResultSet dueTasks = executeQuery(dbQuery);
    return dueTasks;
}
```

Figure 4.3: Pseudo code for reading due tasks.

This simple mechanism for ID discovery suits particularly well the need for fault tolerance. If the node where a given scheduler instance with id j ($1 \leq j \leq N$) dies, the replica will release its lock. This is achieved thanks to Ephemeral Nodes in Zookeeper [51, p.33]. When the health monitoring infrastructure (discussed in Section 4.4) in the cluster notices this, it will start a new scheduler replica on another node. This replica will then go through the same process of Identity Discovery mentioned in the previous paragraph, and since no one holds lock j, it will acquire it. As a result, the system will continue functioning as if a failure did not happen (apart from the slight glitch that might cause consumers to receive a message later and/or receive the same message twice; the delivery guarantees of the scheduler are discussed in Section 4.4.4).

A good question here is what will happen if the node where Zookeeper runs dies. In that case the monitoring infrastructure in the cluster would not help since Zookeeper is a stateful service and simply restarting it on another node would not mean that the lost coordination data is recovered. The most common way failures are handled in applications that need to store information persistently is through data replication [102, Chapter 8] [70]. This is exactly how Zookeeper can be set up to run - in replicated mode. This means that multiple Zookeeper instances have copies of the same configuration files and if one of them dies, the distributed coordination needed by client systems (such as the scheduler) will continue functioning. The particular way Zookeeper achieves data replication is explained in [51, p. 47]. In addition, Zookeeper handles network partitions through employing a master-slave communication strategy between nodes in the cluster. Every update of a Zookeeper node must go through a single master replica or it is not committed. Thus, if a network partition were to happen, replicas that are not on the side of the master would not be able to perform writes. In other words, Zookeeper is C and P, but not A, with reference to the CAP theorem [15]. In the case of the scheduler, this means that in the event of a network partition, it can never be the case that 2 instances think they hold the same ID. The functionality described in this section is implemented in the **uk.ac.gla.dcs.identity_discovery** package.

4.2.4 Scheduler Tick

Once the work related to the previous 2 sections was complete, it was time to implement the tick operation of a scheduler. As a reminder, a tick is the point in time when the scheduler (1) wakes up, (2) reads all tasks that are due (for which it is responsible), (3) calculates the next times those tasks should be triggered and sends the associated messages onto the ESB, (4) syncs the updated task information to the database, (5) determines the next time it should work and (6) falls asleep until this time (also refer to Figure 3.3).

Reading due entries was achieved through simply using the DAL from Section 4.2.2. Using database indices on the query fields (e.g. trigger time) significantly improved the performance of the operation when large number of tasks were due at once [42, Chapter 18]. Furthermore, the initial way syncs to the database were implemented was that for each task, a separate **UPDATE** SQL query was issued to the database server. This was simple and it seemed reasonable for a small number of tasks (e.g. on the order of 100). However, once stress testing of the system commenced, it became apparent that this approach is extremely slow. This problem and its solution are further discussed in Section 4.2.5. Last, but not least, the 4th step in the tick - determining the next time to wake up, is performed using a single SQL query right after the sync phase. Note that it will be problematic if, for example, the scheduler sleeps for 100 seconds and in the mean time a new task is inserted into the database that is due in less than that, say, 20 seconds. The reason is that the task in question will be delayed for 80 seconds. To handle this scenario, whenever a new task is submitted whose first trigger time is before the expected time of the next tick, the producer of scheduler workload in the system (i.e. the System API) notifies a thread running in the scheduler process. When this thread receives such a notification, it will wake up the main thread in the scheduler (i.e. use notify() in Java), which is then going to perform a possibly empty tick[4] and will come up with the new time it needs to sleep until. The ESB is used as the communication mechanism between the 2 processes, again exemplifying the usefulness of the Publish-Subscribe pattern on which the whole project is based [53, Chapter 4].

The third step in a scheduler's tick is highly parallelizable, as was explained while discussing the disadvantages of Celery in Section 1.2.2. To recap, next trigger time calculation could happen in parallel with sending messages. In Computer Science allocating disjoint tasks onto independent execution streams is a form of parallelism, called Task Parallelism [67]. Additionally, the 2 operations can further be parallelized - i.e. there can be multiple threads sending messages and multiple threads calculating next run times. This is also called Data Paralleism in our field (refer to the preceding citation). The way this is achieved in the project is with the help of **ExecutorService** in Java [60]. This is an implementation of a thread pool, which abstracts a lot of the details of the **Thread** metaphor in the language [62]. It also allows for reusal of threads, so that the overhead of thread spawning is mitigated. In the case of the scheduler, the tasks are broken down into 2 implementations of the **Callable** interface [59], namely **MessagesSender.java** and **NextRuntimeCalculator.java** (respectively for sending messages and calculating next trigger times). Conceptually, each of those takes a list of objects - either instances of **Message.java** or **ScheduleForUpdate.java** (those 4 Java classes are part of the **uk.ac.gla.dcs.message_sending** package in the project) and 2 integers. The latter denote the lower and upper indices in the list that will be operated on by the specific Callable. Note that this implies that each worker in the pool can operate on a disjoint portion of the input data. This means that there is literally no overhead of synchronization and/or deep copying of the objects accessed by the threads, which is one of the fastest ways to do parallel computing [50, Chapter 7]. The total number of Callable objects for a given tick equals the number of threads in the ExecutorService (e.g. if there are 8 threads, there would be 4 instances of MessagesSender and 4 instances of NextRuntimeCalculator). Finally, the set of all Callable objects are submitted to the ExecutorService and the main thread waits for all computations to finish before moving on to the next step of the tick.

Sleeping in the scheduler uses the Thread.sleep() function (to which the number of milliseconds is passed). As it was previously discussed, the timeliness of this call depends heavily on the scheduling guarantees of the underlying OS. Linux was chosen as the target platform (as it will become apparent in Section 4.4.1), and because it provides no real-time guarantees, the actual wake-up time of the sleeping thread is in essence non-deterministic. Even so, the scheduling algorithms in Linux are known to be quite fair and it is hardly likely to observe starving processes[5] [107, Section 6.7.1].

A final point regarding the scheduler tick concerns fault tolerance. It was implied in the previous paragraphs that the scheduler sleeps until either the earliest trigger time of a task it is responsible for or if it is woken up by the System API. In either case, this could mean that the scheduler sleeps for a long period of time. This makes perfect sense as if there is no work, it will be wasteful to perform empty ticks. However, there are certain types of process omission failures caused by semantic errors in the code that do not result in a crash of the process (i.e the process does not terminate cleanly) [24, Section 2.4]. Those are very hard to debug as, on the face of it, the system appears to function as if there are no problems. Examples of such failures include deadlocks, infinite loops, etc. A well-considered implementation should take these into account and should (as much as possible) attempt to raise them to the developer's attention. The way this was achieved in the project was to make the scheduler sleep at the end of a tick for the minimum between the amount of time until the next earliest trigger time and a predefined constant. The value of this constant in the project is 5 seconds. With this in place, the scheduler will tick at least every 5 seconds, and this can be used to determine if there is, for example, a deadlock.

[4]An empty tick is one during which no messages are sent.

[5]Processes that get CPU time infrequently.

One way to achieve this is if scheduler records in an accessible place (e.g. the database) the last time of its tick. The monitoring infrastructure in the cluster could then periodically observe those values. If the scheduler has not completed a tick in the last, say, 1 minute, it may be assumed that the process is in a deadlock. The right thing to do in that case would be to kill the scheduler and send a notification to the developers. They can then observe the logs and see exactly where the process got stuck, which can help them locate the problem. The net effect of all of this is that the maintainers are provided with much more clarity when it comes to spotting semantic failures in the system. This point is also addressed in Section 4.4.2 where the parts of the monitoring infrastructure used in the project that provide these facilities are discussed in greater depth.

4.2.5 Issue with Slow Updates

Initially, in order to perform a sync, all tasks that were due were updated in the database using separate SQL queries. This turned out to be a problem when the system was stress tested and the number of tasks to sync at once was high. The amount of time a sync would take grew very fast with the increase in the number of tasks. This is shown in Figure 4.4. It took nearly 17 seconds to update 100k rows. This problem meant that the whole system would be significantly slowed down if it is under load as the sync operation and the reading of due entries for the subsequent tick of the scheduler must be in a happened-before relationship (as otherwise the same tasks would be read, and consequently sent again - check Figure 3.3) [24, p. 607].

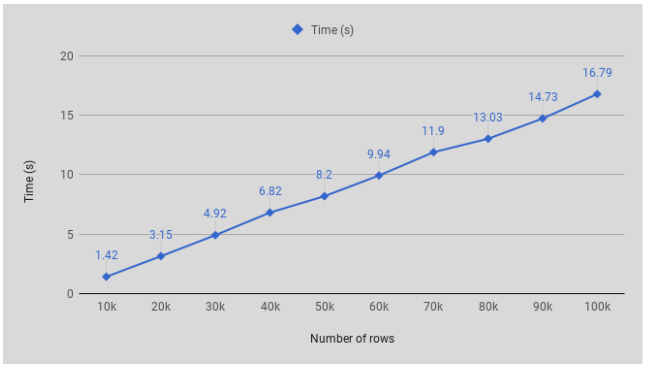


Figure 4.4: Database Access: Naive Update.

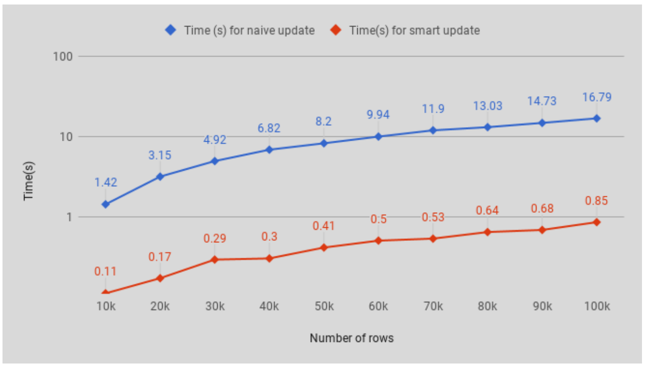
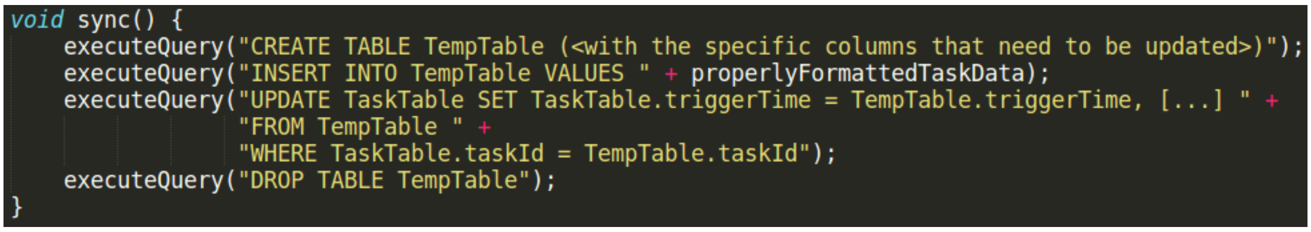


Figure 4.5: Database Access: Naive vs Smart Update.

There were multiple attempts to solve this problem. One tried to keep the tasks in a Redis cache [98] and sync to the database every once in a while (e.g. every 3 minutes). Although this significantly improved the average time a sync takes, it introduced other problems. On one hand, if the scheduler dies and the cache is not synced with the database, then a large amounts of intermediate data would be lost (i.e. there would not be much improvement over what Celery does - refer to Section 1.2.2). On the other, when the System API tries to get information about given task(s) (a functional requirement of the system - Section 2.2.2), it would also need to access the cache of each scheduler replica, which made retrieval of information in the system a very complicated process. Another solution that was tried involved appending updated tasks into the database as new rows in the corresponding table. This allowed to use a single bulk insert statement, which improved the time a sync would take, but not by a desirable factor. The primary reason the improvement was not significant was that when all the rows were inserted, the various database indices also had to be updated, which is by no means an inexpensive operation [42, Chapter 18]. Furthermore, if this approach was used, the size of the database would grow very fast, and there would need to be periodic jobs that delete older entries. It should be noted that each such job would also trigger the update of all indices.

Battling the problem with slow updates actually proved to be the biggest challenge that this Level 4 project posed (as unexpected as it may have been). The recommended book for the DB3 course - 'Fundamentals of Database Systems' by R. Elmasri and S.B. Navathe, was thoroughly researched, but alas, a viable technique that could help in this case was not found. In addition, not much people had been faced with this challenge on the



```
void sync() {
    executeQuery("CREATE TABLE TempTable (<with the specific columns that need to be updated>)");
    executeQuery("INSERT INTO TempTable VALUES " + properlyFormattedTaskData);
    executeQuery("UPDATE TaskTable SET TaskTable.triggerTime = TempTable.triggerTime, [...] " +
                 "FROM TempTable " +
                 "WHERE TaskTable.taskId = TempTable.taskId");
    executeQuery("DROP TABLE TempTable");
}
```

Figure 4.6: Pseudo code for sync.

Internet, and the recommended solutions did not seem to give good results (e.g. the one with appending rows to the database was taken from a SO article). Just before quitting and re-evaluating the requirements of the project, the following article [115], was noticed in the 3rd or 4th page of the results from a Google search about the problem. What it suggested is having a separate, temporary database table, where all the entries that need to be updated are written, using a bulk insert. Since this table would not have indices and would be empty, the insert would be very fast. Afterwards, the temporary table is joined into the main table. In the end, the temporary table was either deleted or completely truncated. Pseudo code for this approach is shown on Figure 4.6. The results were hard to believe. Just as a quick example, updating 100k rows in this way took around 0.8 seconds - more than an order of magnitude faster than the initial approach. To get a better idea of the scale of the improvement, take a look at Figure 4.5 where the performances of the 2 approaches are plotted using a logarithmic scale for the y-axis. The cause of the problem turned out to be the number of SQL queries issued to the database. Initially, if there were N tasks, there would be N SQL **UPDATE** statements hitting the database. With the new approach, for N tasks, the number of update statements is always constant. All of this is while the net effect on the database state is the same with the 2 approaches (i.e. the new one doesn't produce different results). Thinking about it now, it makes perfect sense why the first approach took so long. Query optimization in a database engine usually occurs on a per-query basis [42, Chapter 19]. Thus, the larger the number of updates, the larger the number of individual writes to disk. On the other hand, by having a fixed number of SQL statements, the database engine can perform optimizations that span more data that needs to be written to disk at once. As a result, both the RDBMs is less loaded and the update time is greatly increased.

4.2.6 Need for Database Transactions

As long as the number of entries in the database was fixed and the task data was manipulated only by the scheduler replicas, there would be no need for complex database transactions. To illustrate, all replicas access disjoint subsets of the stored data. As a result, it can never be the case that 2 replicas perform an update operation on the same row. In other words, there is no possibility for race conditions. However, when the System API comes into the picture, the situation becomes a bit more complicated. While it will still be the case that 2 replicas cannot interfere with each other, when the API needs to update a task, the likelihood of inconsistent state of the database is high. To clarify this, consider for simplicity that there is only 1 scheduler in the system. It is responsible for triggering **Task A**, and let's assume that it currently performs a tick during which this will happen. In the mean time, a user of the system updates **Task A** and disables it (i.e. sets **enabled** to **false**) through a single transaction that commits before the scheduler performs a sync. When the scheduler has finished its work and is about to sync to the database, it will have the old value - **true** - of **enabled** in its address space. When it performs its update, it will set **enabled** back to **true**, and thus invalidate the user's transaction. The exact same sequence of actions is depicted on Figure 4.7 for better clarity. Race conditions, such as this one, are a classic example of the problems a Producer-Consumer model needs to take into account [107, Section 5.1]. They way to handle those in the world of databases is through **ACID transactions** [42, Chapter 21]. The acronym is broken down below:

- **A for Atomicity**. This requires that each transaction either finishes in its entirety before committing or

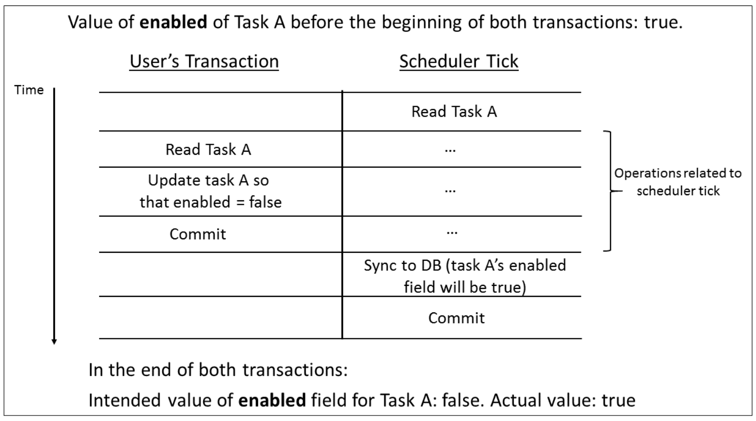

Value of **enabled** of Task A before the beginning of both transactions: true.
User's Transaction
Scheduler Tick
Time
Read Task A
Read Task A
...
Update task A so that enabled = false
...
Commit
...
Operations related to scheduler tick
Sync to DB (task A's enabled field will be true)
Commit
In the end of both transactions:
Intended value of **enabled** field for Task A: false. Actual value: true

Figure 4.7: Database Update: Possible Race Condition.

does not commit at all.

- **C for Consistency**. The database must be left in a consistent state after a number of transactions that execute concurrently all commit.
- **I for Isolation**. Intermediate results of a transaction should not be visible to other transactions until the first one commits. If this is not in place, there are possibilities for data inconsistencies.
- **D for Durability**. Although this property is not related to the above problem, it ensures that once a transaction has committed, the state of the database will not change even if a crash occurs. This is achieved with the help of Write-Ahead Logging [42, Section 23.1.3].

To solve this problem, a feature in PostgreSQL called Row-Level Locking was used [89]. When the scheduler begins its tick, it initiates a new transaction (**BEGIN** SQL statement) and it locks all table rows that represent due tasks through **SELECT FOR UPDATE** (for more information, refer to the previous citation). This means that any other transaction trying to update those rows would block until the scheduler commits. The latter happens only after the sync is performed. This, in essence, solves the problem described above. To illustrate, in Figure 4.7, the SQL transaction initiated by the user will be able to update the database only after the scheduler commits. As a result, the state of the data will be consistent. It can be seen that all of the 4 ACID properties will be satisfied in the end, especially Consistency, which is the one that is the most important in this scenario.

Note that the need for ACID transactions fully justifies the reason why an SQL database is used. Even though it is not uncommon for NoSQL databases to be used with data that has schema, they are definitely not suitable for complex updates, such as the one described above.

4.3 System API

4.3.1 Choice of Technologies

The particular **architectural paradigm** followed when building the System API was REST[6], as described by Roy Fielding in his famous doctoral dissertation [44]. The primary reason for this choice was that client and server here are very loosely-coupled because REST demands the use of hypertext. In other words, they do not need to use the same technology stack or make any assumptions about the internal workings of the other. This is

[6]REpresentational State Transfer.

unlike other Remote Procedure Call (RPC) schemes, such as Java RMI[7] which restricts participants to the JVM. Also, REST is usually used alongside HTTP, which meant that the Distributed Scheduler would be interoperable with the rest of the World Wide Web. This would be very suitable if the Web UI requirement (refer to Section 2.2.3) was actually built as the System API could be used directly by a front-end framework, such as React or AngularJS.

There are many back-end **web frameworks** that ease the development of web services. The one used in this project was Django [33]. It was chosen among the alternatives because it is easy to use and promotes fast development (Django's slogan is "The web framework for perfectionists with deadlines"). Additionally, it abstracts away a lot of the concerns that need to be addressed when building applications for the Web. For example, automated testing of a web service requires various components, such as an instance of the application and a database, to be running at once because the interaction between them is usually verified (i.e. it is a form of Integration testing [108]). Django comes packed with a test execution module which automatically spins up (and subsequently tears down) dummy instances of the various components comprising the web application. This allows the developer to focus on writing tests, as opposed to building a suitable test infrastructure. Furthermore, Django is written in Python, which implies that the wide community support for the language is available to the developer. Admittedly, Python's reference implementation is not as fast as Java and C++ (as was discussed in Section 4.2.1), which for certain applications is a problem (e.g. the calculation of next trigger times for the scheduler). However, in the case of web services, scalability is most often achieved through engineering a suitable architecture (e.g. employing replication, load balancing, caching, etc.), as opposed to focusing on the performance of a single instance (take a look at this Reddit post for Flask [56]). On account of this, Django was more than appropriate for the task at hand. Last, but not least, there is a package built on top of Django - Django Rest Framework (DRF) [34], that wraps over a lot of the concerns when building REST APIs in particular. Those include content negotiation, authentication policies, serialization of objects, and many more. Django and DRF together made the development of the System API a very simple and pleasant process.

4.3.2 API Endpoints

A RESTful endpoint is a Unique Resource Identifier (URI) that pinpoints a piece of functionality exposed by a given API. The URI is a pattern that can incorporate a hierarchical structure to denote similar endpoints (e.g. **/users/view-all-users** and **/users/create-new-user/**<**username**>). This, in fact, is the prescribed way of identifying REST resources (refer to the 5th chapter in Fielding's dissertation) and the one followed in the design of the Distributed Scheduler's API. The endpoints, along with a brief overview of the expected content types, format for each request, example output and more are given in Appendix A.

Details of how all API endpoints are implemented is not provided since most of them literally perform CRUD[8] actions on the underlying database. It is interesting to note that updating a task involves issuing an **UPDATE** SQL query that will block if a scheduler replica currently operates on the same row in the database. In other words, the problem with consistency of the database is already handled in the scheduler, as was discussed in Section 4.2.6. Furthermore, the second paragraph of Section 4.2.4 mentioned that the System API notifies a scheduler replica through the ESB (in some cases) when a new task is submitted so that big delays in task triggering are avoided. The place this happens is in the implementation of the **/tasks/submit-task** API call, immediately after the task information is saved in the database. In addition, **/tasks/get-tasks** adopts the use of server-side pagination to reduce response time whenever there are many tasks in the system [118]. A final point to mention w.r.t. the API endpoints is that JSON is used as the data-interchange format between the clients and the server [27]. The reason for this is that JSON is very lightweight (e.g. compared to XML), proved to be sufficient for representing the data in the system, and the Python Standard Library has a very simple and efficient module for encoding/decoding JSON [93].

[7]Remote Method Invocation.

[8]Create, Read, Update, Delete - the 4 basic functions of persistent storage.

4.3.3 Load Balancing

It has been stressed in previous sections that there must be a way for the system workload to be balanced between the scheduler replicas so that each of them is responsible for a disjoint set of the task specifications. Section 4.2.3 already outlined how each replica uses its unique ID when querying the database to ensure isolation from others. To complete the picture, it remains to be discussed how those IDs are associated with each tasks. This is actually a very simple process whose description follows: The System API component knows the number of schedulers in the system - N. It also keeps track of the ID of the scheduler that was last assigned a task - id_{last}. When a new task is submitted, the API determines the scheduler that would responsible for it by adding 1 to id_{last} and computing mod N of the result. This implements a very easy to reason about Round Robin (RR) load-balancer. It is also worth mentioning that N and id_{last} are stored in the database. This means that any conflicting accesses (e.g. if there are multiple instances of the System API service running for load balancing purposes) can be synchronized using appropriate locking mechanisms provided by the database server [42, Chapter 22].

While working on the load balancing it was acknowledged that RR might not be the best task allocation strategy. To exemplify this, consider the scenario where there are 2 deployed scheduler replicas. It could then be the case that tasks are submitted in such a way that one of the replicas gets tasks whose trigger interval[9] is very small, while the other - extremely large. In this scenario, although the tasks are evenly balanced, one of the schedulers will have to do more work than the other. A possible amendment to the load balancing strategy would be for the System API to try to assign tasks to replicas in a way that takes into account the actual amount of work a scheduler needs to do. This would make the system imprevious to the aforementioned problem. However, time did not permit to properly conceive and evaluate another task allocation strategy. As a result, this improvement fell into the scope of future work to be carried out on the Distributed Scheduler.

4.3.4 Message Routing Model

During the Sprint where the System API was built, the way users would interact with the system was considered in greater depth. A problem that became apparent then was that nothing stopped 2 users from interfering with each other's workflow when communicating with the ESB. Before explaining exactly in what way this could cause issues and how the problem was resolved, a quick overview of the routing capabilities of RabbitMQ is given in the below paragraph.

An **exchange** in RabbitMQ is a named resource to which producers send messages. The job of an exchange is to route a message it receives to a (set of) RabbitMQ **queue(s)**. In other words, it could be the case that a single message sent by a producer is delivered to more than one queues. The exact way this is done is imposed entirely by the application logic built on top of the broker. Every message has associated with it, among other attributes, an exchange name and a routing key. The former signifies the exchange to which the message will be sent. The latter is used in actually determining which queues the exchange will down link the message to. In addition, a queue binds to one or more exchanges via a binding key. To clarify how all these RabbitMQ concepts work together, it is best to give an example. Suppose a message **m** with routing key **k1** is sent to to an exchange **e**. The set of queues that will receive **m** - **Q**, comprises those queues that are bound to **e** via a binding key that matches **k1**. Figures 4.8 and 4.9 provide an interactive walk-through of the example so as to clear any confusions. Note that consumers listen on queues and there can be more than one consumer listening on the same queue (e.g. for load balancing reasons).

Now, to illustrate the problem, consider the case where 2 different users of the system both specify a message that will be sent to a queue **q**. In other words, they both provide the same (exchange name, routing key) tuple when they submit their respective task specifications. It is assumed for simplicity that the queue has already been bound to the exchange via a binding key that matches the routing key of the 2 messages (in the case of the

[9]The interval between 2 consecutive triggers of the task.

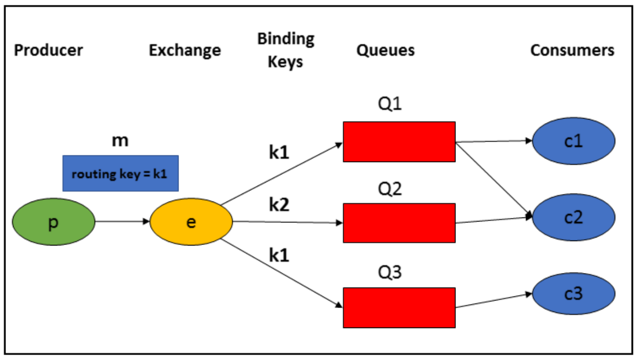


Figure 4.8: RMQ Routing - Before message is sent.

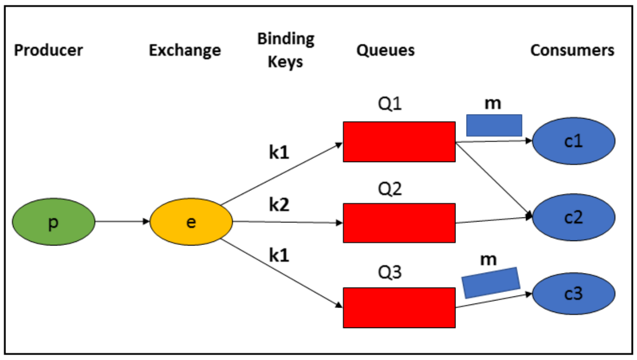


Figure 4.9: RMQ Routing - Message delivered.

scheduler this also happens when a message is sent). Let's also assume that the 2 users have consumers listening on that queue. If this is the case, the message broker will load balance the messages onto the consumers and from the point of view of the users it might appear as if some messages are omitted. Equivalently, the users might observe some unknown messages they never specified themselves. This is only a single example of an unintentional interference that could arise between users of the system. A malicious user could also exploit this lack of authorization by, for example, sending a message to a group of consumers listening on a specific queue. All that user needs to do is, again, submit a task with the message in question and an (exchange name, routing key) tuple that identifies the queue where the target consumers listen. The 2 scenarios mentioned above are probably not exhaustive of the ways things can go wrong. Even so, without loss of generality, it can be stated that the system would not be safe to use, unless the issue was addressed.

The very first idea for solving the problem described above was to try to make sure that the names of queues, exchanges and routing/binding keys are unique and known only to the intended user when a task is submitted. For this approach to work, it also had to be the case that those names are hard to reproduce. Apparently, it was the Distributed Scheduler's responsibility for ensuring the 2 conditions above. This is because making the users come up with hard-to-guess unique names is very bad user experience, which, in fact, is demanded by some web services (e.g. the AddItem call in the eBay's Trading API [1]). On account of this, the perfect place for generating these was the System API component. A simplified workflow of how this would happen is described below:

- A user sends a request for the submission of a new task.
- On receiving and validating the request data, the System API creates a set of unique names for an exchange, a queue and a routing key.
- The API then saves the task data into the database and returns to the user with the 3 names.
- At this point, the user knows where messages will be sent to and can start consumers that listen on the specified queue.

Given the above steps, one is expected to ask how it can be ensured that the generated names are unique. RFC 4122 gives a set of specifications for generating Universally Unique IDentifiers (UUIDs), which fortunately enough, are implemented in Python's Standard Library under the **uuid** module [52, Section 12.8]. Those identifiers are 128 bits long and actually do guarantee uniqueness across space and time. With this in place, it could be seen that the problems described above are actually solved. For example, it will be impossible for 2 users to end up sending messages to the same queue. Moreover, the effort a malicious user needs to put into uncovering the queue of another user would be comparable to that of cracking his/her password (of course if the appropriate safe communication mechanisms are in place when interacting with the API - e.g. TLS).

Although the solution described thus far solves the issues with user authorization, it was not the one used in the end because it carried some unfavourable consequences. First, it was now impossible for a user to submit 2 tasks whose messages are to be sent onto the same queue (since the System API generates new resource names on every task submission). This meant that every time a new task went into the system, the user had to start another consumer listening on the queue where messages will be delivered, which is with little doubt cumbersome. Furthermore, creating new exchanges and queues on every task submission, meant that the broker will be overloaded as it needs to keep track of all these entities [116, Chapter 2].

After some reflection on the first solution and research on the Internet, a new idea was considered that did not involve UUIDs. It turned out that RabbitMQ comes with the ability to set Access Control Rules (ACR) for the broker resources (e.g. exchanges and queues). This StackOverflow article [96] suggests a way to utilize ACRs that served well in solving the authorization problem. The concrete idea is that every user is allowed to access only resources whose name start with their username, appended with a ”-” (e.g. for ”georgi” that would be ”georgi-”). Once the ACRs are set for the specific user, they can submit tasks with any name for the RabbitMQ resources. It would be the System API's task to prepend the username and the dash to each of those resource names. In other words, this approach implements a very simple (yet effective) namespace for every user. This means that even if 2 users want to send messages to a queue called ”A”, there will be no clashes as 2 separate RabbitMQ entities will be created. This additionally solves one of the problems in the first solution that was discussed. Namely, a user is now capable of submitting 2 tasks whose message are sent to the same queue (because with this approach the user is responsible for specifying resource names). As the solution was further considered, it was decided that instead of allowing users to create multiple exchanges, there would be a single exchange per user (whose name is again determined by the username). In this way not only can users focus only on providing queue names and binding keys for a task (reducing the cognitive load on them), but also the broker would not be overloaded with many exchanges. The latter solves the second problem with the first solution.

The solution described above was the one chosen for the system as it seemed the better way to tackle the problem. It only remained to be determined when those ACRs will be set. It was decided that this would happen when a user registered to the system via the System API. Thus, when a registration takes place, the API would send an HTTP request to the RabbitMQ management plugin [116, Section 8.1] with the ACRs for the new user.

4.4 Deployment

4.4.1 Choice of Technologies

In previous sections (e.g. 4.2.3 and 3.3.4), when fault tolerance of the system was discussed, the terms ”monitoring infrastructure” and ”self-healing cluster” were used quite often. This hopefully implied that the system was built with using such a tool in mind and there would not be attempts to implement custom fault tolerance as a deliverable of the project (which was actually done before the technologies discussed in the paragraphs below came about - e.g. in HDFS[10] [105]). Indeed, there are software tools out there that support the automation of deployment and maintenance of application stacks, such as the Distributed Scheduler. Before outlining some of them, however, a quick introduction is given to a concept they all utilize in order to provide their services - operating-system-level virtualization, also known as **containers** [84].

In a nutshell, containers are very similar to traditional virtual machines (VMs) in that they provide an isolated environment that acts as an execution unit. The difference between VMs and containers is that the former require a software component, called a hypervisor, that acts as an intermediary between the VM environment and the underlying hardware. Containers, on the other hand, use certain features in the OS kernel (e.g. Linux namespaces) to provide isolation between processes. Due to the fact that containers run directly on top of the

[10]Hadoop Distributed File System.

underlying OS, they exhibit little-to-none extra overhead (in comparison to host OS code execution) [78, Chapter 1] [66, Chapter 28]. With VMs this depends on the particular virtualization strategy (e.g. type 1 hypervisors are faster than type 2 hypervisors) [107, Chapter 16]. In any case, containers are very lightweight and are characterized by much faster launch times than VMs [29]. Docker has become the de facto container technology in recent years and was the one adopted in the project [36]. It is important to note that even though the Docker Engine is platform independent, the containers managed by it are built for the Linux kernel. Docker on Windows has to spawn a Linux VM in order to run containers. As a result, Windows is used mostly only for development of Docker applications [78, Chapter 2]. On account of this, Linux was used as the operating system for deployment of the Distributed Scheduler.

The abstraction containers provide are leveraged by what is known as **container orchestrators** which are the concrete examples of the self-healing clusters mentioned above. These can be viewed as a set of distributed processes that run on a cluster of nodes, to which the client submits a description of the desired state of an application stack[11]. These processes then have several tasks. First, they need to schedule the application containers on the cluster according to a specific placement scheme [22]. In addition, they do their best to ensure that the desired state of the application is met at all times. This means that if a node where a container runs crashes, the container will be rescheduled on another node in the cluster. This is precisely the reason why it was stressed that it is important the scheduler is a stateless service - because if such a failure happens, all the data persisted directly in the crashed container where the service runs will be lost for good. The crash discovery and recovery functions are realized by a central component in the orchestrator that periodically executes health checks on every container running in the cluster and performs the necessary actions if a failure is noticed.

The 2 orchestrators that were considered for the project are **Docker Swarm** [112] and **Kubernetes** (K8s) [68]. The benefits of the former are that it enables native clustering for Docker. In a sense, knowing how to use Docker implies that you know how to use Swarm because the latter is part of the Docker Engine's API (as of v. 1.12.0). In addition, setting up a Docker Swarm cluster is very easy provided that all participating nodes have Docker installed. Application stacks are described in a **docker-compose** file, which is basically a yml file that needs to follow a certain format [85]. Once a cluster is set-up and a docker-compose file for the application is produced, deploying the stack is as simple as executing a single command. K8s, on the other hand, also exposes a similar API that expects services to be described in yml files. Similarly to Docker Swarm, once a cluster is up and running, deploying stacks onto it is very simple. K8s is indisputably the orchestrator with more features. For example, it has the concept of **Pods** that allows for 2 or more containers to be scheduled on the same node in the cluster [88]. This makes sense, for example, if an application is to be deployed alongside a caching server (e.g. Redis) - ideally one would want both containers to be on the same node to avoid network overheads. The way to do this with Swarm is to run 2 processes in the same container, which defeats the purpose of Docker - to enhance the development of loosely-coupled and highly-cohesive microservices that have well-defined responsibilities [78, Chapter 1]. Despite the many features, K8s (and especially v. 1.7.*) had some disadvantages that made it unsuitable for the project. To begin with, setting up a cluster is no way near as easy as with Docker Swarm. This could have been resolved by using Google Kubernetes Engine (GKE) which is a PaaS[12], built on top of Google Cloud Platform (GCP), that provides pre-bootstrapped K8s clusters [47]. Unfortunately, a student grant for GKE usage was not secured, so another alternative had to be considered. In addition, K8s treats node failures much differently than Docker Swarm. The latter responds very quickly and reschedules all containers on another machine within seconds. On the other hand, when K8s misses a health-check from a node it does not immediately assume a crash. Rather, it acknowledges the fact that there might be a network partition and keeps performing health-checks for a fixed period of 5 minutes [83]. Only after the last expires are the Pods running on the dead node rescheduled. Since 5 minutes was considered a big delay for the Distributed Scheduler, a decent amount of research was made for whether this eviction policy can be adjusted. Unfortunately, it turned out that the K8s version considered at the time did not provide the users with this facility. As a result of those configuration problems with K8s, Docker Swarm was chosen as the orchestrator for the Distributed Scheduler.

[11]An application stack is a set of components that act together to fulfill an application goal.

[12]Platform as a Service.

The **cloud provider** used for the project was **AWS** [4]. One of the reasons for this is that it was the only provider for which a student grant was secured. Additionally, some research was conducted on whether there is a simple way to set-up Docker Swarm clusters on AWS. Surprisingly, it turned out that Docker itself provides easy way to bootstrap AWS Swarm clusters through Docker Cloud [37]. This provided a very simple solution to what was thought would be the biggest problem in the Deployment milestone - automating the provisioning of Docker Swarm. The following sections describe how the technologies described thus far were used to realize some of the non-functional requirements mentioned in Section 2.3.

4.4.2 Fault Tolerance

With the technologies above in place, having a fault-tolerant deployment of the Distributed Scheduler was an easy task (taking into account that the system itself was built with fault tolerance in mind). Before providing an overview of the process, 2 assumptions are made that would simplify the explanation. First, the stateful services in the system (Zookeeper, PostgreSQL and RabbitMQ) are running somewhere in the cloud and are configured for fault tolerance and high availability (e.g. with redundancy of the instances, replication of data, etc.). This is so that focus can be placed on the components that are built while working on the project. Second, assume that there is a Docker Swam on a cluster of AWS EC2 instances that has been set-up a priori (e.g. through Docker Cloud). In other words, it only remains to deploy the replicated scheduler service and the System API on the Swarm.

The first step in achieving the goal is to build the Docker images for the scheduler and the API, and push them to a Docker registry which can be accessed from the AWS EC2 instances (e.g. the official Docker Hub [39]). Thereafter, the application stack has to be described in a docker-compose file. The file would contain an entry for the 2 services (i.e. the scheduler and the System API). Each such entry would have (among other things) the docker image of the service, a command to be executed at the respective container's entry point, and a field denoting the number of replicas to be deployed for the specific service. A simplified yml file for the Distributed Scheduler is provided in Appendix B for reference. Finally, the application stack is deployed onto the cluster in question via the **docker stack deploy** command (run on a manager node of the Swarm) [40]. It is important to note that most of the development efforts in this milestone were focused on writing deployment scripts that automate the process described above.

Once launched inside its respective Docker container, each scheduler replica begins following the steps on the Life Cycle Activity Diagram - Figure 3.3. If a replica crashes (e.g. due to a node restart), the failure will be noticed by the Swarm infrastructure. As a result, a new scheduler instance will be started on another EC2 node inside the cluster. The crash would cause the release of the Zookeeper lock that corresponds to the replica's ID. The newly started replica will then be able to acquire the same lock, and consequently - the same ID (refer to Section 4.2.3). Note that a focus is primarily placed on the scheduler service in the application stack. The failure model and recovery mechanisms are identical for the System API.

The last paragraph of Section 4.2.4 briefly mentioned a case where the monitoring infrastructure needs collaboration from the developer to spot certain process omission failures (e.g. infinite loops and deadlocks). The way this is achieved in Docker Swarm is through custom health checks that can be specified for a service in the docker-compose file. A specification for a health check consists primarily of (1) a command to be executed inside the container where a service replica runs, (2) interval between 2 health checks, and (3) number of retries before a health check is considered unsuccessful. The return status of the command is used to determine the status of the health check - 0 is for success, and anything else - for failure. The above parameters for the specific example, mentioned in the last paragraph of Section 4.2.4, could be:

- (1) - a script, **check_scheduler_liveness.py**, that notes the current system time and the time the scheduler replica has last ticked. If the latter is more than 20 seconds in the past, the script exits with status code 1.

Otherwise, the status code is 0.

- (2) - the interval between 2 consecutive health checks is 20 seconds.
- (3) - the number of retries on failure is 2. This means that for the health check to fail, the command must exit with a non-zero code 3 times in a row.

Given this setting, if a scheduler replica has not performed a tick in the last minute, it could safely be assumed that it is unhealthy. As a result, the monitoring infrastructure would terminate the target container. In addition, it will notify the developers of the failure and/or (in the Distributed Scheduler's case) simply restart the replica. The logs of the crashed container can then be examined and the causes of the failure narrowed down. Hence, provided that an application can be built in a way that allows for reasoning about its run time execution, the custom health check feature of Docker Swarm can provide a great deal of transparency to the developers and help them uncover some of the hardest to debug software failures.

4.4.3 Scalability of The System

Distributed systems are often associated with the notion of scalability. This, in essence, is the system's ability to respond to increased demand without the need to change the underlying design [43]. There are 2 types of scalability that are considered when building a distributed system - horizontal and vertical. The former means that an application scales by adding instances that run on new machines. Vertical scalability, on the other hand, involves adding more resources to a single machine (such as RAM, vCPUs, etc.). It is the less preferred option in cloud environments as it is usually more expensive and less flexible [102, Chapter 2]. As a result, distributed systems are often built with horizontal scalability in mind.

Docker Swarm's API includes the option to scale a given service by increasing the number of its replicas [101]. The scheduling algorithms in the orchestrator would always try to evenly balance the replicas of a given service across the nodes in the cluster. This worked perfectly for the Distributed Scheduler since each replica performs intensive operations that are expected to fully utilize the resources of the node on which it runs. For example, multiple messages are sent to the ESB concurrently by multiple threads and it is therefore expected that all the network interfaces of the node will be busy. As with the other Swarm features used in the project, scaling up or down the number of replicas of a service is achieved with a single command (refer to the previous citation). In the case of the Distributed Scheduler, 2 additional fairly simple steps had to precede that. The reason, as was mentioned in previous sections, is the value of the new number of schedulers - N - needs to be known to several entities. Firstly, the System API has to know N in order to include the new instance in the load balancing logic (refer to Section 4.3.3). Since the old value is stored in the database, a single SQL **UPDATE** query needs to be executed. Also, Section 4.2.3 outlined the Identity Discovery process where it was implied that each scheduler replica needs to know the value of N (so that it can try to acquire one of the N locks that would give its ID). This is achieved through updating an environment variable (**NUM_SCHEDULERS** from Appendix B) in the container where the new replica will run. That environment variable is persisted for the Swarm, so that in case a replica crashes, its latest value is available to the subsequently restarted instances. This is yet another feature available in Docker Swarm.

4.4.4 Scheduler Delivery Guarantees

A problem that was noticed while working on the fault tolerance of the system was that it was possible for a scheduled message to be sent more than once. To exemplify how this could happen, assume that a scheduler replica performs a tick where **task A** is to be triggered. The problem would occur if there is a failure after the message associated with **task A** is sent to the ESB, but before the sync where the new trigger time is written to the database. In that case, when the scheduler replica is restarted, it will re-trigger **task A**, and consequently send the same message to the ESB. With reference to the Computer Science literature, this is an example of at-least-once delivery semantics [24, Chapter 9]. Admittedly, in the use case of the Distributed Scheduler, exactly-once semantics are the ideal scenario. Unfortunately, however, after a lot of research of how this might be achieved, the conclusion was reached that this would not be possible. The reason for this is that in case of a failure, the newly started scheduler replica must have knowledge of which (if any) messages managed to get sent before the crash (so as not to send them again). For this to work, the fact that a message has been successfully delivered to the correct endpoint had to be recorded in a non-volatile store - an activity referred to as Write-Behind Logging [10]. It was noticed, however, that this will not account for the case when a failure happens after a message has been sent, but before a success log entry is recorded.

After some further research, it turned out that exactly-once delivery guarantees, in the general case, are extremely hard to achieve (if not impossible) in distributed systems. The reason for this is that the producer, the messaging system, and the consumer all need to cooperate in order to achieve exactly-once delivery. This breaks the separation of concerns between the 3 entities and becomes a complicated task when the failure models of each is taken into consideration - refer to L. Lamport's Byzantine generals problem [71]. On account of these points, the responsibility of handling duplicates in a pipeline is usually pushed onto the end consumers of messages. The way this works is that messages are tagged with a unique identifier. The consumers exhibit one of 2 characteristics. Either they are able to filter out duplicates based on the unique tags or their processing of a message is an idempotent operation (i.e. one that has the same results no matter how many times it is executed). To support both scenarios, the Distributed Scheduler sends, alongside the message payload, the task identifier (unique and stored in the database) and a sequence number that denotes how many times the task has been triggered. The latter is simply an integer which is read from the database when a task is due, incremented, and written back on a sync. The combination of the 2 uniquely identifies a trigger for a given task. The realizations mentioned in this paragraph were reached with the help of the following online articles - [113] [54] [81].

The above workaround was against one of the design principles with which the Distributed Scheduler was built - to keep the complexities within the system and to provide an easy to reason about API to its users (refer to Section 3.1). A realization reached while working on the project was that distributed systems are nearly always associated with such trade-offs between increased performance and higher complexity, and the Distributed Scheduler was, unfortunately, not an exception.

4.5 Chapter Summary

This chapter gave a thorough overview of how the implementation stage was carried out. It began by outlining the adopted Software Engineering practices that guided the development process. It then examined the 3 milestones that were reached throughout the project, considering for each the choice of technologies, the achievements, the difficulties faced as well as how they were addressed. Chapter 5 will discuss how the Distributed Scheduler was evaluated and whether the aims and objectives listed in Chapter 1 were met.

Chapter 5

Evaluation

There were 2 reasons for evaluating the Distributed Scheduler. First, it had to be verified that the components of the system exhibit the expected behaviours under various circumstances. This was achieved via Unit and Integration testing that was carried out throughout the project. In addition, Chapter 1 mentioned some of the key drawbacks of related products in the context of large organizations where the number of periodic tasks is likely to be large and the timeliness of their execution - important. Those disadvantages were exemplified through outlining the architectures of 2 alternative solutions - cron and Celery. The Distributed Scheduler was consequently introduced as a system that aims to tackle those problems and provide a reliable service for large-scale systems. As a result, in the light of proving that the Distributed Scheduler succeeds in achieving this, the chapter also conducts comparative evaluation against both cron and Celery.

5.1 Testing

Automated testing was considered to be an integral part of the project and was carried out at the end of every Sprint where a component of the Distributed Scheduler was built. **Unit testing** aimed to verify that the service under development behaved correctly as a self-contained unit. This means that interaction with external resources was not considered. Instead, importance was placed only on the pieces of functionality that are local to the component in question. **Integration tests**, on the other hand, involved demonstrating that a component interacts with external services as expected. This included both verifying the state of external components (e.g. checking that a scheduler sync wrote the correct data to the database) and ensuring exceptions resulting from interactions are handled correctly (e.g. making sure that a scheduler replica exits with an error code after noticing that the ESB is down).

It is important to note that automated testing provided a great deal of transparency into the behaviour of the Distributed Scheduler and helped uncover and fix a lot of bugs. The below 2 sections describe how testing was carried out for the 2 components that have been built as part of the project.

5.1.1 Testing the Scheduler

To begin with, it is worth mentioning that both Unit and Integration tests for the scheduler service were written using JUnit as a test framework. JUnit provided a lot of flexibility and allowed for the clear separation of the 2 types of testing strategies (through the use of JUnit Categories [99]). Also, JUnit turned out to be target of many open-source tools whose purpose is to further ease the testing process. One of those was used in the project, as it will become apparent in the following paragraphs.

Unit testing in the case of the scheduler service involved verifying mostly that the different stages of the tick operation are working as expected. For example, there is a test case that checks if (at the end of a tick) the newly computed trigger times (for the set of due tasks) were correct. Even though one would expect this to be a simple exercise, conducting unit testing for this component was not trivial since a scheduler replica would interact with many external resources (refer to Section 4.2.1). As a result, most of the functionality, local to the scheduler, was not possible to use without having to interface with, for example, the database. To tackle this problem, a Software Engineering technique, called **mocking** was used. Mock objects provide the same interface as external objects, but only simulate the latter's functionality [108, Section 8.1.1]. For example, a mock object that simulates the scheduler's database, could return a list of due tasks, and thus eliminate the need for data to be transferred over the network. There is a mocking framework, called Mockito, built for Java, that provides this functionality [77]. Using Mockito, it was possible to test the smallest pieces of functionality within the scheduler without having to spin up instances of the external resources. This is usually desirable as it significantly reduces the time the test suite takes to run, which has in turn proven to prompt developers to execute their tests more often [104, p. 300].

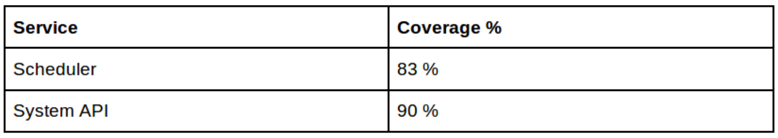
Integration testing was a problem initially as an automated way of bringing up the required entities had to be found. At first, some articles on the Internet suggested using the production environment for this. For example, at the start of testing, all the required tables are created in the production database, and dropped at the end [122]. This approach quickly went out of consideration for 2 reasons. First, it was considered to be unclean and could cause trouble if something went wrong (e.g. if the process does not complete due to an error, the tables used for testing might not be dropped). Second, there was really no persistent production environment for the system - the cluster set-up described in Section 4.4 was only used for test deployments and was after that torn down as it would incur costs. After some further research, an open-source JUnit Rule for Docker developed by Palantir was discovered [38]. It provided precisely the functionality that was required for Integration testing. To begin with, it would spin-up the required external entities in Docker containers. Additionally, it allowed for a self-defined readiness check on a given component that would determine if it is ready to be used. Only after all such checks have passed, will the test suite execution begin. This was needed because, for example, in order to test the interaction between a scheduler replica and the database, the latter had to be properly launched (i.e. be in a state in which it can accept connections). Furthermore, the JUnit Rule recorded logs from containers, which greatly helped the debugging of test failures. Last, but not least, it was easy to use and integrate into the project. With this in place, Integration tests could be ran on any machine with a single command (provided that Docker and Gradle are installed). This was viewed as a huge achievement, given the complexity of the system in terms of number of services that need to be present in order for it to function properly.

5.1.2 Testing the System API

The System API was much easier to test because the web framework used - Django, provided a lot of facilities that simplified the process (as was mentioned in Section 4.3.1). Unit testing consisted of verifying that the underlying utilities produce the expected output. For example, it had to be ensured that the functions that parse user input notice and correctly flag malformed fields (e.g. a wrong cron expression when submitting a task, a time stamp without time zone information, etc.). Integration tests, on the other hand, checked the API's behaviour when used by clients of the system. In other words, this involved sending HTTP requests to a running instance of the System API and inspecting if the returned responses are as expected and/or the state of the database reflects the change that was made.

5.1.3 Code Coverage

Code coverage is a measure used to denote the portion of the code that has been executed during automated testing [104]. M. Fowler [45] and B. Marick [74] warn that code coverage should not be misused as 'high coverage numbers are too easy to reach with low quality testing'. Even so, they both agree that, when used



Service	Coverage %
Scheduler	83 %
System API	90 %

Figure 5.1: Code Coverage Statistics.

thoughtfully, the metric can be a good indication of a reliable system. They advise that developers aim at 80-90% coverage as having around 100 'would smell of someone writing tests to make the coverage numbers happy, but not thinking about what they are doing'. Figure 5.1 shows the coverage for the 2 services in the system. In addition, Appendix D includes more detailed Screenshots of the particular tools used to capture the metric and gives more in-depth analysis of the results. Those figures (referring again to the 2 Engineers) imply that the Distributed Scheduler has indeed been tested sufficiently and promote some degree of trust in the system.

5.2 Comparative Evaluation

This section provides comparative evaluation against cron and Celery with the purpose to justify the design decisions made when working on the project.

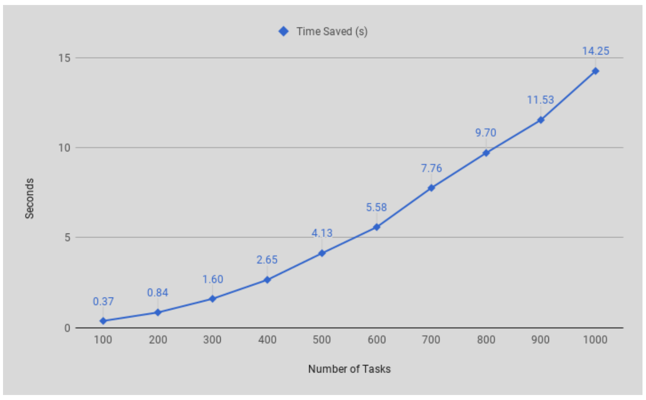


Figure 5.2: Time saved from avoiding connection establishment.

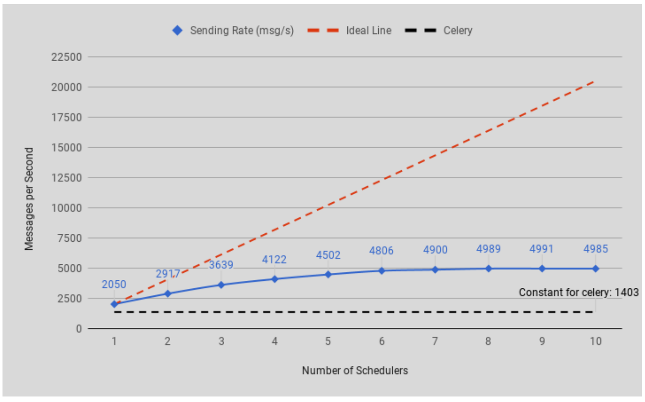


Figure 5.3: Sending rate improvement in comparison to Celery.

5.2.1 Against cron

As it was discussed previously in the dissertation (refer to Section 1.2.1), one of the main disadvantages of cron was that periodic jobs interacting with external resources, such as a database, would need to set up and tear down connections every time they execute. Here, cron and the Distributed Scheduler are compared on precisely this criterion. The aim is to show that the latter is to be the preferred choice in such use cases. The experimental setting included a periodic task whose purpose is to perform some cleansing activities on a database server and then send a notification of the successful completion of the operation to a remote service. The task specification could be expressed in the world of cron through a Python script that (1) sets up the necessary connections and (2) performs the application logic. Assume that the script is called **cron.py**. In order to see how the Distributed Scheduler could be used here, it is worth going back to Section 1.2.2. It hinted that a client system based on an ESB, where consumers listen for events on specific queues, could set up connection to services once and then reuse them for as long as they run. Taking this into consideration, the consumer for the experiment was, again, a Python script (so that the same set of technologies can be used for the comparison of the 2 systems), which initially sets up connections to a RabbitMQ instance, the database server, and the remote service that is to be notified. It then begins listening for triggers on a RabbitMQ queue, and when one comes, performs the application logic. Let the name of this script be **distr_scheduler.py**. With both scripts in place, the workings of cron and the Distributed Scheduler can, respectively, be simulated in the following ways:

- To execute the task in cron's case, a new process has to be spawned where **cron.py** is executed. In other words, running **python cron.py** in a Unix shell would be equivalent to what crond would do (refer to Section 1.2.1).
- With the Distributed Scheduler, **distr_scheduler.py** will first need to be run. Subsequently, triggers could be sent to that consumer (e.g. via another RabbitMQ client). In the end, an execution of the task in question involves (1) receiving of the trigger message and (2) acting on it (i.e. performing the application logic).

Appendix C provides the reader with the pseudo code of the 2 scripts along with a more in-depth discussion on how they are used to represent cron and the Distributed Scheduler.

The experiment per se measured how much time was saved with **distr_scheduler.py** (due to avoiding repetitive connection establishment) with the increase of the number of times the periodic task was executed. The results are shown in Figure 5.2. It can be observed that 1000 executions of the task with both variants lead to the Distributed Scheduler saving 14 seconds from connection establishment. This is more than 3 times as much as for 500 executions. This trend is to verify that the paradigm adopted by the scheduler would result in significantly reduced overheads due to its flexibility, which allows for connection pooling. It is also important to note that all components in the experiments (i.e. the scripts, the database server, and the remote service) were run on the same machine. This implies that if they were placed on separate nodes, the overheads of connection establishment would be even higher. This is the right place to say that the princples the Distributed Scheduler preaches are actually adopted in the implementation of the system itself. Section 4.2.2 outlined the use of pooling (in the implementation of the scheduler service) as a way of achieving lower latency when interacting with the scheduler's database and the ESB.

5.2.2 Against Celery

The Distributed Scheduler and the scheduler component of Celery were compared on the sending rate (SR) the 2 systems adopt under high load. This was achieved by submitting a high number of tasks - say, 40k to both systems, all of which trigger at the same time and recur every 5 seconds. In other words, the sending rate here is expected to be 8k messages per second. The reason these numbers were chosen was not to provide a target benchmark for the Distributed Scheduler. Instead, it was because they were high enough so as to saturate the SRs of the systems under evaluation. This implies in itself that either of the 2 systems may not be able to achieve the rate, required to send all messages on time. To get some intuition as to why this might be the case, consider how a scheduler replica works (the following explanation is valid for Celery too). At time t_0 - the initial trigger time of all 40k tasks, it would wake up, and begin performing a tick. Unless this operation finishes at time t_1, $t_0 \leq t_1 \leq t_0 + 5s$, (which is the next time the 40k tasks are due), the next tick would not begin on time, and consequently some messages would be delivered with delay. A corollary effect would be precisely a lower SR than what is needed (e.g. if only 30k messages were sent for the 5 seconds, the SR would be 6k msg/s).

As a result of the above, the purpose of this evaluation was in fact twofold. Firstly, and primarily, it meant to show that the Distributed Scheduler had higher SR than Celery. However, it also aimed to showcase that increasing the number of scheduler replicas in the system would correspond to increase in the SR. The latter would justify the load balancing of the tasks in the system across the replicas and all the complexities (accompanying this design choice) that had to be taken care of.

Before moving onto the actual metrics gathered, a quick overview of how the 2 systems were set up for evaluation is given. Both Celery and the Distributed Scheduler were deployed on EC2 **t2.xlarge** instances running a Docker Swarm [3]. The latter was deployed 10 times in a row such that every new deployment added an extra scheduler replica. Each replica ran on a node of its own so that it could fully utilize the node's resources (Section 4.4.3 gives a more detailed explanation of why this is needed). In addition, the stateful services utilized by the Distributed Scheduler - Zookeeper, PostgreSQL, and RabbitMQ, were run inside Docker containers, which were deployed in the Swarm as well (each occupying a node of its own). This was not ideal because, as was mentioned in Section 4.4.2, in order for those services to perform optimally, they require special configuration. Regardless of this, the deployment proved to be sufficient for the experiments and it also had the enormous advantage of being reproducible with a single command. Furthermore, Celery and the Distributed Scheduler alike were instrumented with the help of Prometheus [92] and Grafana [49], which extremely eased the gathering of metrics. Last, but not least, each deployment was ran for 15 minutes. As a result, the SR was calculated as the ratio of the total number of messages sent to the amount of time the deployment was running (i.e. 900 seconds).

Figure 5.3 depicts the results of the experiment. The single deployment of Celery achieves SR of 1403 msg/s (note that a horizontal dashed line is shown on the graph, so that Celery's performance can be more easily compared with that of the Distributed Scheduler). On the other hand, a Distributed Scheduler deployment with a single replica manages to send 2050 msg/s. The reason for this is both that the system is powered by the JVM's highly optimizing JIT compiler (as opposed to the Python's reference implementation in Celery's case) and the optimizations that were introduced in the implementation, discussed in Section 4.2.4. This very first observation per se fulfills the first purpose of the evaluation mentioned above.

The figure also shows a result that was initially not anticipated. The dashed red line shows how the system should ideally scale as the number of replicas is increased. This is derived from the performance of a single replica. For example, if there are N scheduler instances in the system, the ideal SR would be N multiplied by the SR of a 1-replica deployment. Even though it was not expected that the actual performance of the system will fully match the red line, it was very surprising to find out that its growth rate is much lower. Even worse, when there are more than 7 schedulers, there is virtually no improvement in the SR. Several days were spent researching why this might be the case. In the end, 3 possible reasons were put forward. First, PostgreSQL and RabbitMQ both have a limit on the amount of concurrent work they can carry out in parallel. Adding more replicas was likely to have resulted in one (or both) of these services' capacity being saturated. Second, it was mentioned in the previous paragraph that the stateful services were not configured for production use. If this was actually done (i.e. configure replication, load balancing, partitioning, etc.), it would very likely result in better response to the increased load and in turn - higher growh of SR. The least likely possibility for the plateau region (but still one that cannot be excluded) was that the limited shared network bandwidth of the EC2 instances had been saturated by the constant flow of data. Regardless of which reason did cause the non-linearity in SR increase, this was a typical example of an application of Amdahl's law, which warns that if a task is not 100 % parallelizable, there is a limit to the theoretical speed-up that can be achieved with parallel computing [5]. This indeed came as a disappointment, but was a valuable lesson taken from the project. In any case, the fact that 7 scheduler replicas were more than twice faster than a single one was irrefutable. This, in essence, fulfilled the second purpose of comparing the Distributed Scheduler with Celery.

5.3 Chapter Summary

This chapter gave an overview of how the Distributed Scheduler was evaluated. Initially, it presented the testing strategy used in the project. Afterwards, it showcased the results of the comparative evaluation that was carried out with cron and Celery. The final chapter quickly summarizes what has been talked about in the previous 38 pages, provides a short discussion on future work that can be carried out on the Distributed Scheduler and concludes the dissertation.

Chapter 6

Conclusion

6.1 Summary

This document presented my work on the distributed event-triggering scheduler I built in the final year of my undergraduate studies at the University of Glasgow. The report began by introducing the context in which the system would be useful and giving background discussion on related products. Afterwards, the functional and non-functional requirements of the Distributed Scheduler were listed and prioritized. Subsequently, the Design chapter provided the reader with a high-level overview of the system, which helped identify the key components and the interaction between them. The document then discussed the implementation of the Distributed Scheduler: Chapter 4 first listed the Software Engineering practices adopted throughout the academic year and then went on to thoroughly review the key milestones reached while working on the system. Finally, the dissertation outlined how evaluation of the Distributed Scheduler was carried out.

6.2 Future Work

As it has been mentioned in some of the chapters, not all of the features that were initially planned for the Distributed Scheduler were implemented. This was primarily due to the big scope of the project and the many unknowns at the beginning. As a result, any work on the system after the end of the Level 4 Individual Project course would indisputably involve realizing the features that had to be left behind. Those are listed below:

1. A Web UI for non-programatic interaction with the system (Section 2.2.3).
2. Calculating per-user utilization of the system (Section 2.2.4).
3. Real-time operation of the Distributed Scheduler. This was probably the least understood requirement due to the unclear definition of 'real time', which was the main reason why it was dropped at such an early stage of the project (refer to Section 2.3.4). However, if it does turn out to be possible to introduce a set of real-time constrains and actually implement them in the system, its value will with little doubt increase a lot due to the added guarantees and transparency. On account of this, future research looking into this is definitely worth being carried out.

In addition to the requirements that did not make it in the final version of the Distributed Scheduler, there were also some areas of improvement that the dissertation mentioned. To begin with, the second paragraph in

Section 4.3.3 acknowledged that the load balancing strategy used in the system might not be the optimal one. As a result, coming up with a better algorithm than RR (in its simplest form) for distributing tasks to scheduler replicas is definitely something to be considered in the future. Additionally, the results from the comparative evaluation against Celery (refer to Figure 5.3) showed that the increase of the sending rate of the system (as more replicas are added) reaches a plateau region at 7 scheduler replicas. The last paragraph in Section 5.2.2 outlined 3 possible reasons as to why this might be the case. Given this, it would be very interesting to review those in greater detail and conclude if it is possible to increase the number of replicas after which there is no improvement (e.g. by deploying the stateful services with the appropriate production configurations).

Furthermore, even though the system was tested extensively, it is definitely the case that the QA could be strengthen even further. What Figure 5.1 showed was code coverage of the Distributed Scheduler in number of statements executed while the test suite runs. Although this is the metric that is used most often in the industry, there are other coverage criteria that are more effective. For example, branch coverage and condition coverage are supesets[1] of statement coverage [79, Chapter 4]. Admittedly, they are not as efficient as the latter due to the increased overheads of testing (primarily when it comes to time it takes to produce the test suite). Nevertheless, scoring higher on them would correspond to higher perceived reliability of the system. What is more, Continuous Integration (CI) is an Agile technique that demands that the code is built and tests are run every time before a set of changes are pushed into the central code repository (GitLab in the case of the Distributed Scheduler) [26, Chapter 14]. In practice, there are CI tools that automate this process and using one is recommended in any software development effort. Even though GitLab has a built-in CI runner that is easy to use, time did not permit to set it up early in the development process (it did not make much sense to do that at the very end as CI is expected to be an on-going practice). Regardless, if any future work is carried out on the system (especially if other developers join), configuring GitLab CI would definitely be on the to-do list.

Finally, the Distributed Scheduler was never tested in a real-world scenario. Even if it can be assumed that in its current state the system meets its specification (taking into account the discussions in the Evaluation chapter), it is never until actual users come into the picture that a given software product is truly tested. As a result, it would definitely be interesting to observe how the system performs if it is deployed in a production context.

6.3 Final Reflections

This Level 4 Individual Project was the most significant undertaking during my time as an undergraduate student at the University of Glasgow. It allowed me to apply a lot of the concepts I have learned about in my courses and was undoubtedly the biggest measure of how much I have grown as a Software Engineer over the last 4 years. In addition, the project allowed me to acknowledge the importance of conducting evaluation of a newly-built system, which was something I had not done in the past. Furthermore, while I worked on the project I employed many technologies that I have not used before. As a result, I feel that my ability to quickly take up new tools has improved a great deal, which is a very important skill to have. Last, but far from being least, despite the fact that I managed to deliver most of the requirements, my work on the Distributed Scheduler did not go without encountering some challenges that tested my abilities to the limit. Some of those I have thoroughly described in the dissertation, others I had to omit as otherwise the 40-page bound would have been exceeded by a long way. Regardless, I believe that it is those hurdles that provided the most valuable experience as they not only improved my problem-solving skills and critical thinking, but even more importantly, made me realize I am capable of tackling non-trivial challenges in a systematic and efficient manner. As a result of this, I am leaving the University of Glasgow confident that I am prepared for whatever life in the industry has to offer.

[1]In other words, having 100 % branch and/or condition coverage implies 100 % statement coverage, but not vice versa.

Appendices

Appendix A

System API Endpoints

Note that all endpoints require that the HTTP **Content-Type** header be set to **application/json**. In addition, endpoints A.3-A.6 require an **Authorization** HTTP header to be set to the auth token for the specific user making the request. As a result, for the sake of simplifying the explanation, we assume that the appropriate headers are set a priori.

Also **note** that only examples are given where the requests to the System API are successful. It is therefore important to say that all API endpoints include extensive validation and return error messages in the case of malformed input.

A.1 Registering a User

Endpoint

This endpoint registers a new user into the system. To achieve this, we submit an HTTP POST request to **/users/register**.

Details

Alongside the request, we must submit a JSON payload that conforms to the format on Figure A.1.

Example

An example request is shown on Figure A.2. The response is shown on Figure A.3

A.2 Authenticating a User

Endpoint

Once a user is registered, they can get their auth token by sending a POST request to **/users/auth**. This auth token will subsequently be needed for accessing the further API endpoints.

Details

We must submit a JSON payload that conforms to the format on Figure A.4.

Example

An example request is shown on Figure A.5. The response is shown on Figure A.6.

A.3 Submitting a New Task

Endpoint

To submit a new task into the system, we must send an HTTP POST request to **/tasks/submit-task**.

Details

We must submit a JSON payload that conforms to the format on Figure A.7. The object attributes surrounded by square brackets are optional.

Example

An example request is shown on Figure A.8. The response is shown on Figure A.9. Note that the System API has assigned a unique identifier to the task. This is used when updating or getting information about the task.

A.4 Updating an Existent Task

Endpoint

The endpoint for updating an existent task is **/tasks/update-task/"task-name"** where "task-name" is replaced by the unique identifier of the task. This is done via a POST request.

Details

We must submit a JSON payload that conforms to the format on Figure A.10. Note that all object attributes are optional. However, we must specify at least one.

Example

An example request is shown on Figure A.11. The response is shown on Figure A.12.

A.5 Getting Information for a Single Task

Endpoint

To get information about a single task, we submit a GET request to **/tasks/get-task-by-name/”task-name”** where ”task-name” is replaced by the unique identifier of the task.

Example

An example request is shown on Figure A.13. The response is shown on Figure A.14.

A.6 Getting Information for a List of Tasks

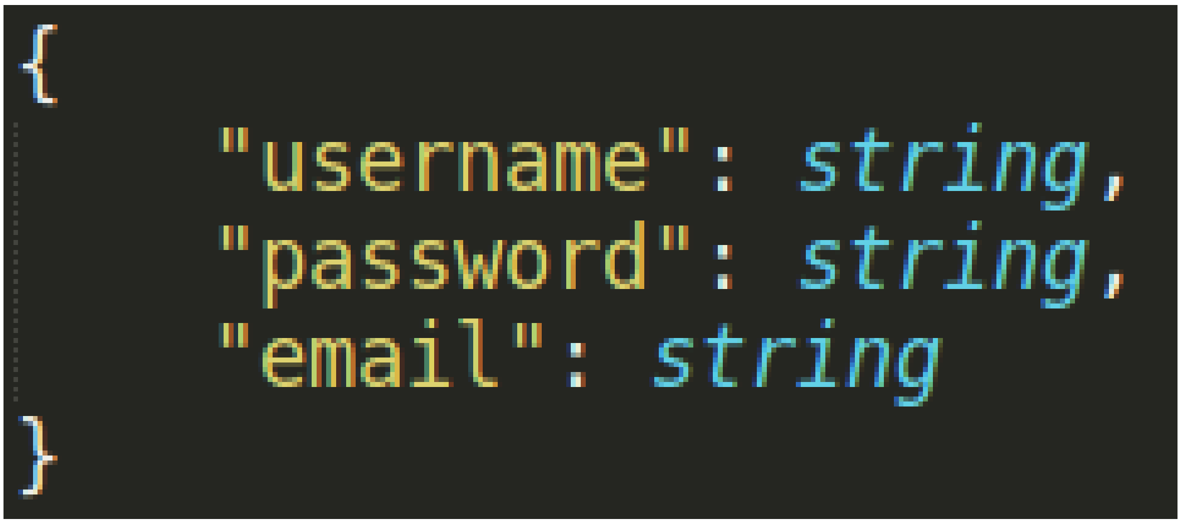
Endpoint

To get information about a list of tasks, we submit a GET request to **/tasks/get-tasks?page=”page-number”** where ”page-number” is replaced by an integer. Note that server-side pagination is used here to reduce load time.

Example

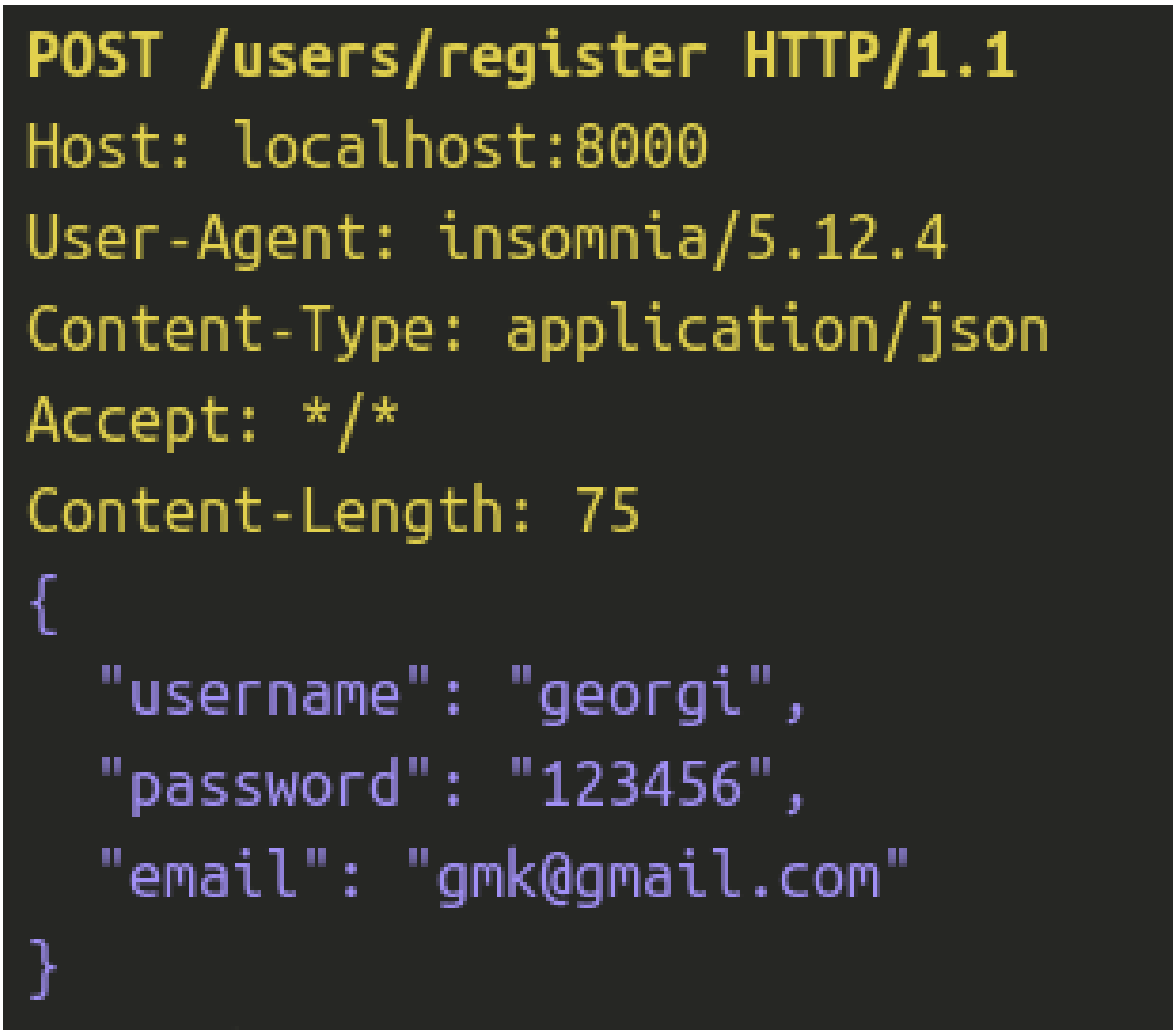
An example request is shown on Figure A.15. The response is shown on Figure A.16. The **results** JSON attribute is a list of task data, each element of which has the exact same format as the one shown on Figure A.14. In addition, the **next** attribute is a link to the next page of tasks if such exists, which is not the case in the example. Identical is the case for **previous**.

A.7 Screenshots



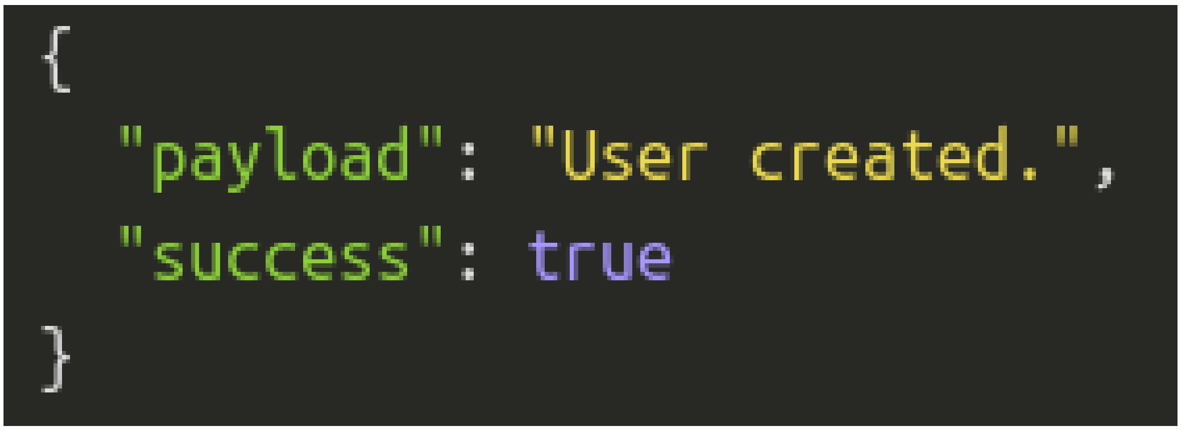
```
{
    "username": string,
    "password": string,
    "email": string
}
```

Figure A.1: JSON payload format for registering a user.



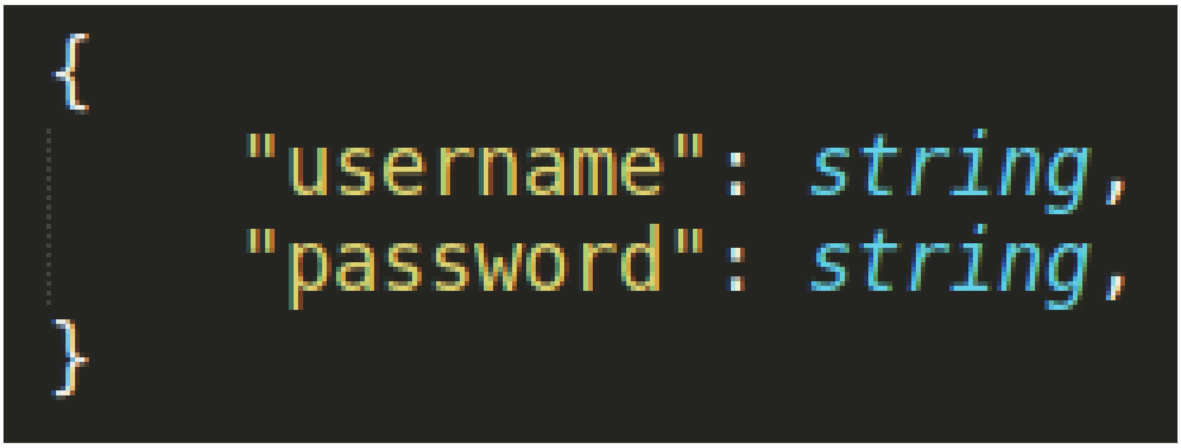
```
POST /users/register HTTP/1.1
Host: localhost:8000
User-Agent: insomnia/5.12.4
Content-Type: application/json
Accept: */*
Content-Length: 75
{
  "username": "georgi",
  "password": "123456",
  "email": "gmk@gmail.com"
}
```

Figure A.2: Example request for registering a user.



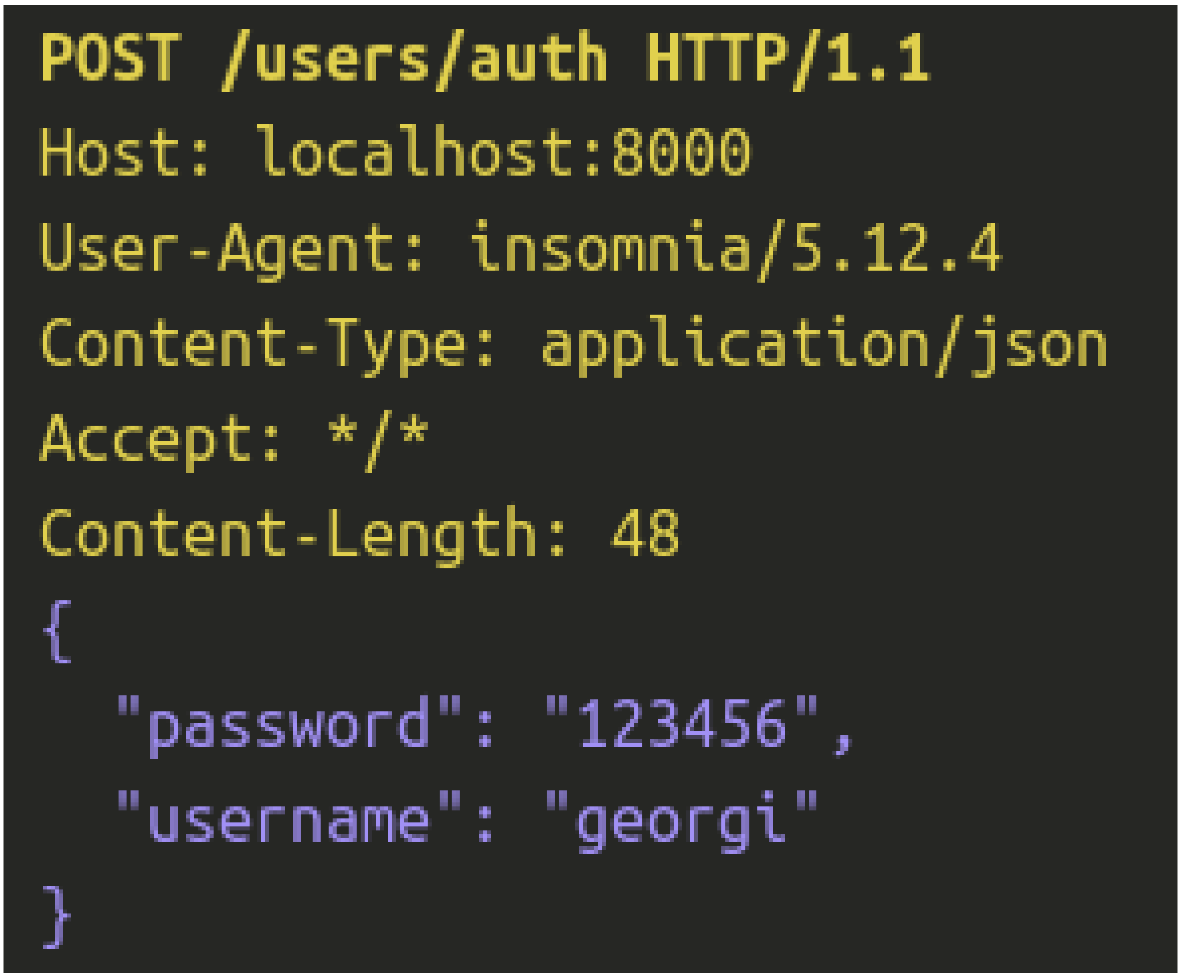
```
{
  "payload": "User created.",
  "success": true
}
```

Figure A.3: Example response on successful registration.



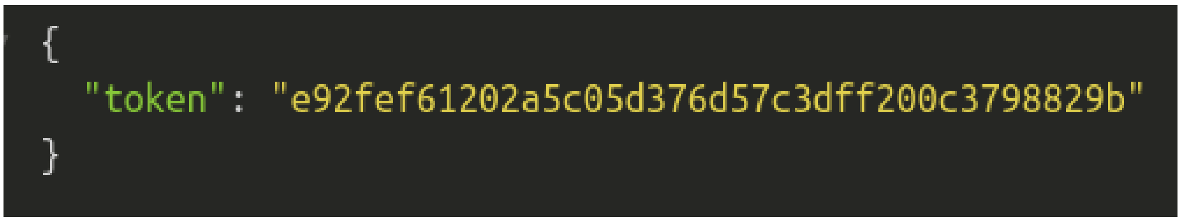
```
{
    "username": string,
    "password": string,
}
```

Figure A.4: JSON payload format for authenticating a user.



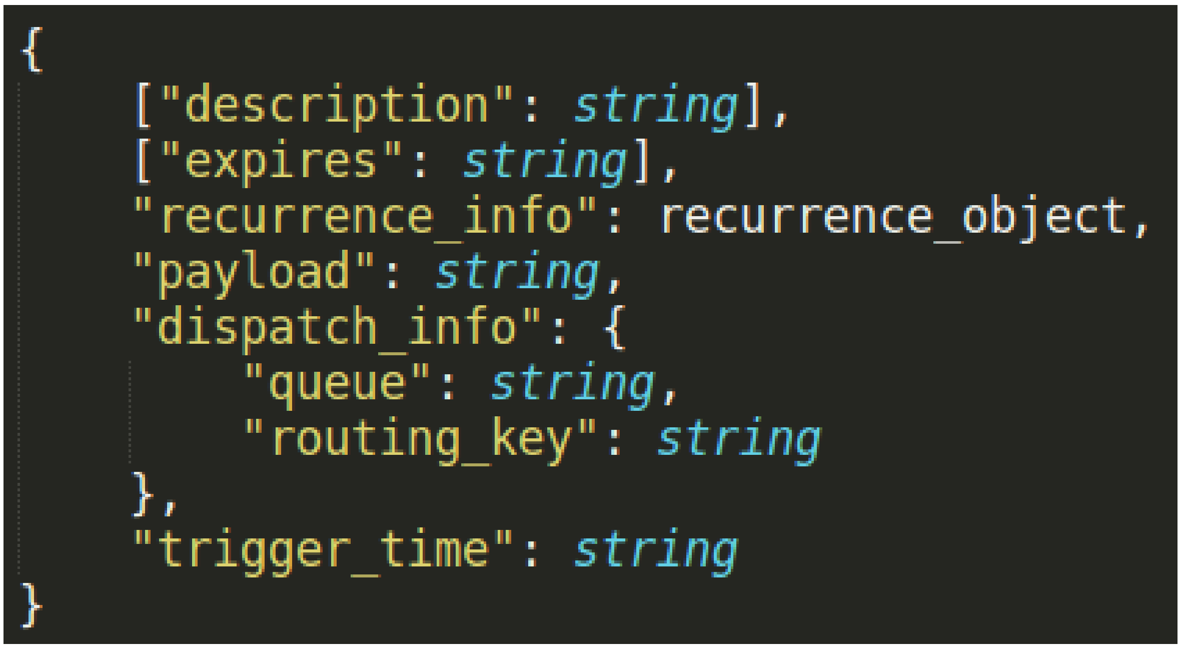
```
POST /users/auth HTTP/1.1
Host: localhost:8000
User-Agent: insomnia/5.12.4
Content-Type: application/json
Accept: */*
Content-Length: 48
{
  "password": "123456",
  "username": "georgi"
}
```

Figure A.5: Example request for authenticating a user.



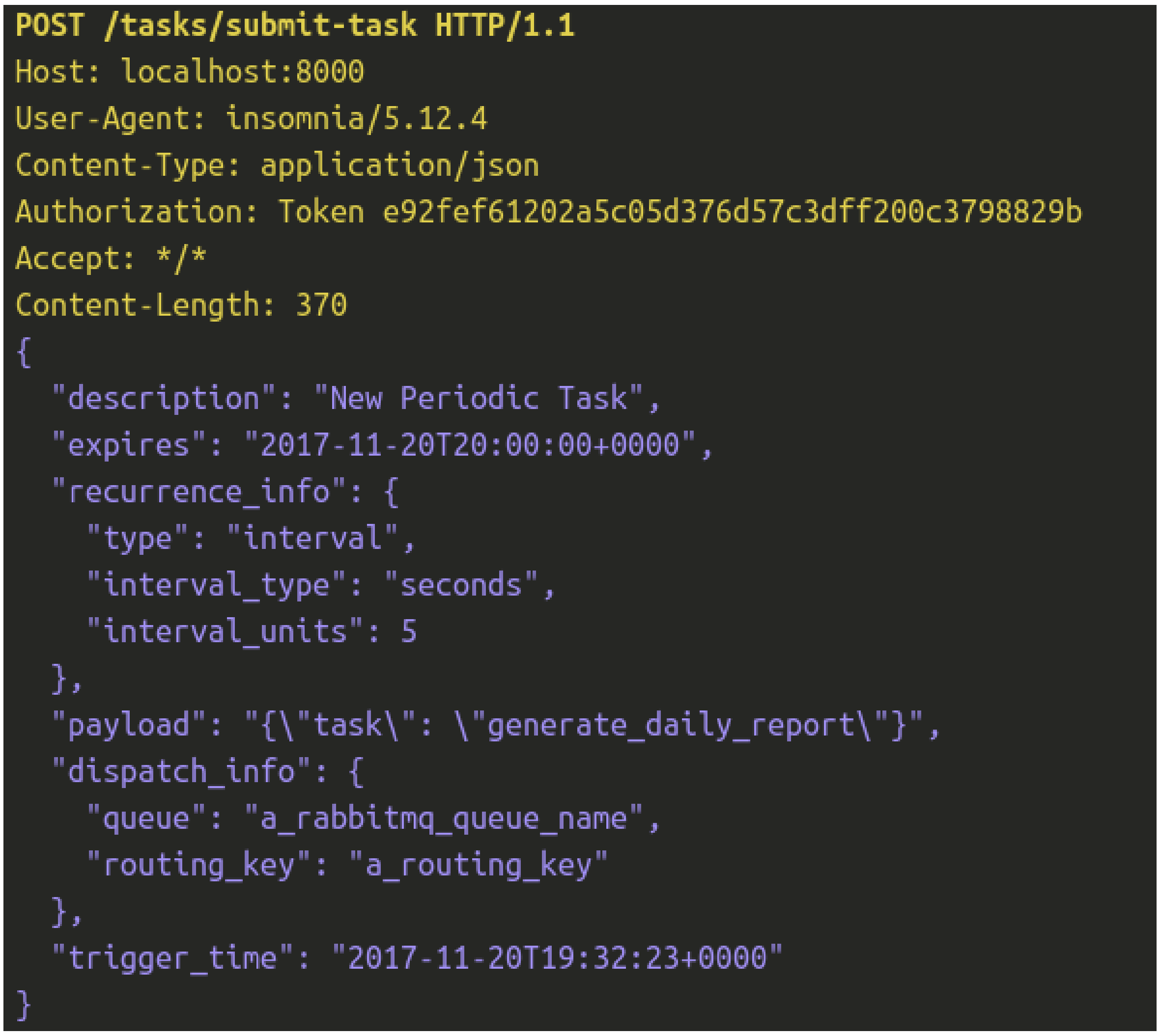
```
{
  "token": "e92fef61202a5c05d376d57c3dff200c3798829b"
}
```

Figure A.6: Example response on successful authentication.



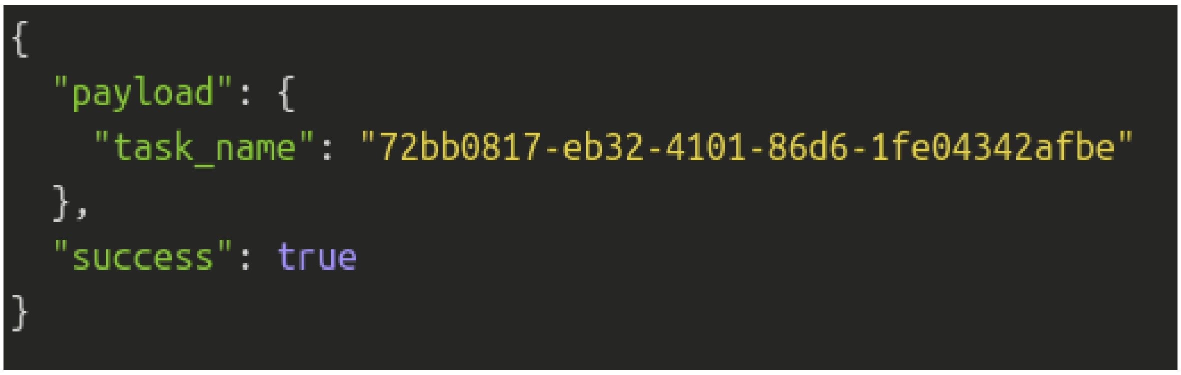
```
{
    ["description": string],
    ["expires": string],
    "recurrence_info": recurrence_object,
    "payload": string,
    "dispatch_info": {
        "queue": string,
        "routing_key": string
    },
    "trigger_time": string
}
```

Figure A.7: JSON payload format for submitting a new task.



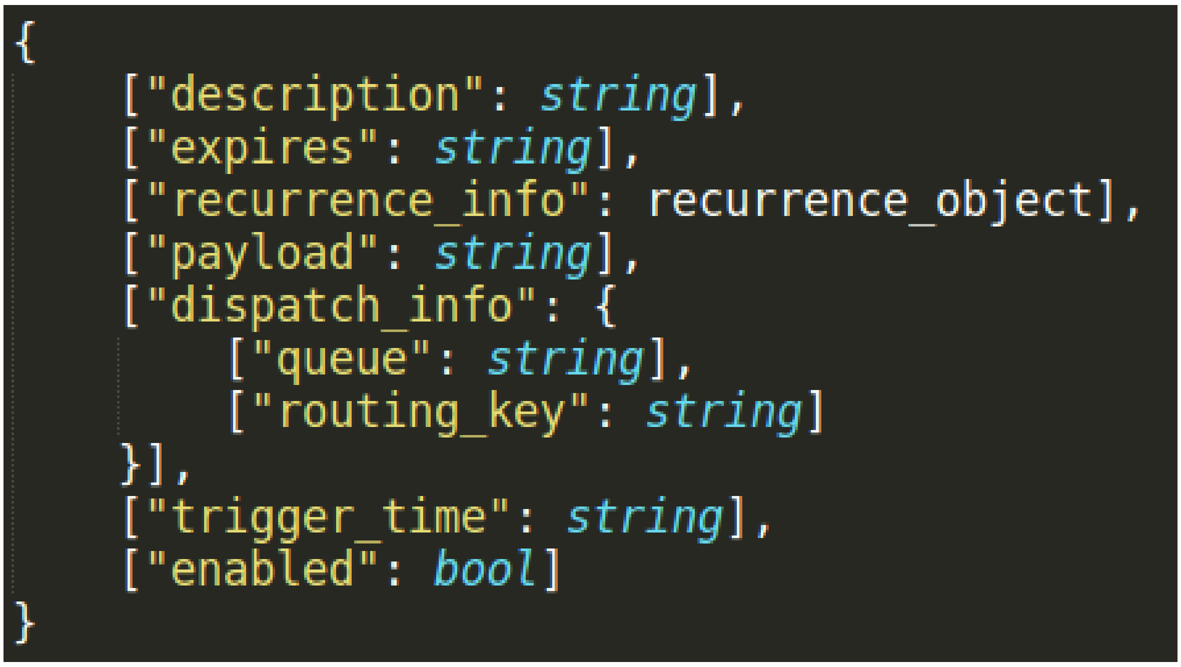
```
POST /tasks/submit-task HTTP/1.1
Host: localhost:8000
User-Agent: insomnia/5.12.4
Content-Type: application/json
Authorization: Token e92fef61202a5c05d376d57c3dff200c3798829b
Accept: */*
Content-Length: 370
{
  "description": "New Periodic Task",
  "expires": "2017-11-20T20:00:00+0000",
  "recurrence_info": {
    "type": "interval",
    "interval_type": "seconds",
    "interval_units": 5
  },
  "payload": "{\"task\": \"generate_daily_report\"}",
  "dispatch_info": {
    "queue": "a_rabbitmq_queue_name",
    "routing_key": "a_routing_key"
  },
  "trigger_time": "2017-11-20T19:32:23+0000"
}
```

Figure A.8: Example request for submitting a new task.



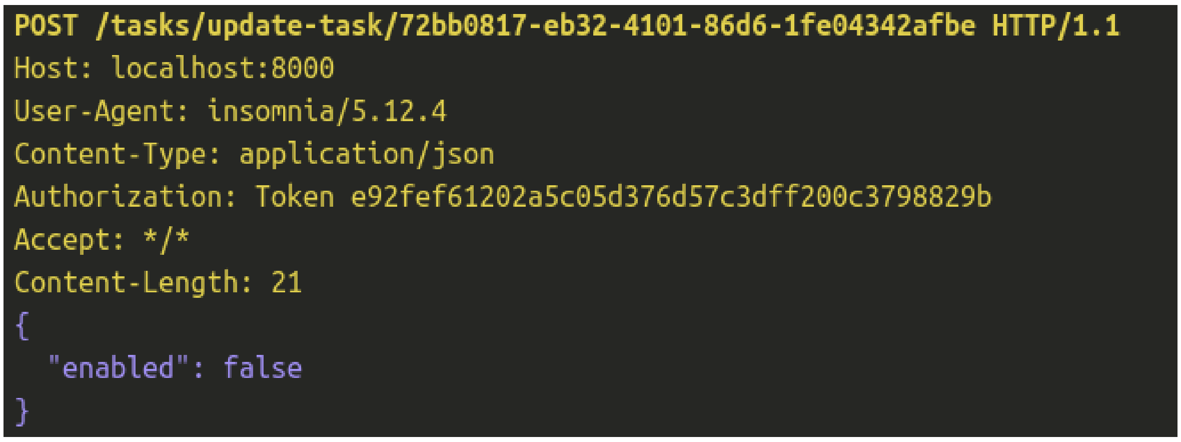
```
{
  "payload": {
    "task_name": "72bb0817-eb32-4101-86d6-1fe04342afbe"
  },
  "success": true
}
```

Figure A.9: Example response on successful task submission.



```
{
     ["description": string],
     ["expires": string],
     ["recurrence_info": recurrence_object],
     ["payload": string],
     ["dispatch_info": {
         ["queue": string],
         ["routing_key": string]
     }],
     ["trigger_time": string],
     ["enabled": bool]
}
```

Figure A.10: JSON payload format for updating an existent task.



```
POST /tasks/update-task/72bb0817-eb32-4101-86d6-1fe04342afbe HTTP/1.1
Host: localhost:8000
User-Agent: insomnia/5.12.4
Content-Type: application/json
Authorization: Token e92fef61202a5c05d376d57c3dff200c3798829b
Accept: */*
Content-Length: 21
{
  "enabled": false
}
```

Figure A.11: Example request for updating task.

```
{
  "payload": "Task successfully updated.",
  "success": true
}
```

Figure A.12: Example response on successful task update.

```
GET /tasks/get-task-by-name/72bb0817-eb32-4101-86d6-1fe04342afbe HTTP/1.1
Host: localhost:8000
User-Agent: insomnia/5.12.4
Authorization: Token e92fef61202a5c05d376d57c3dff200c3798829b
Accept: */*
```

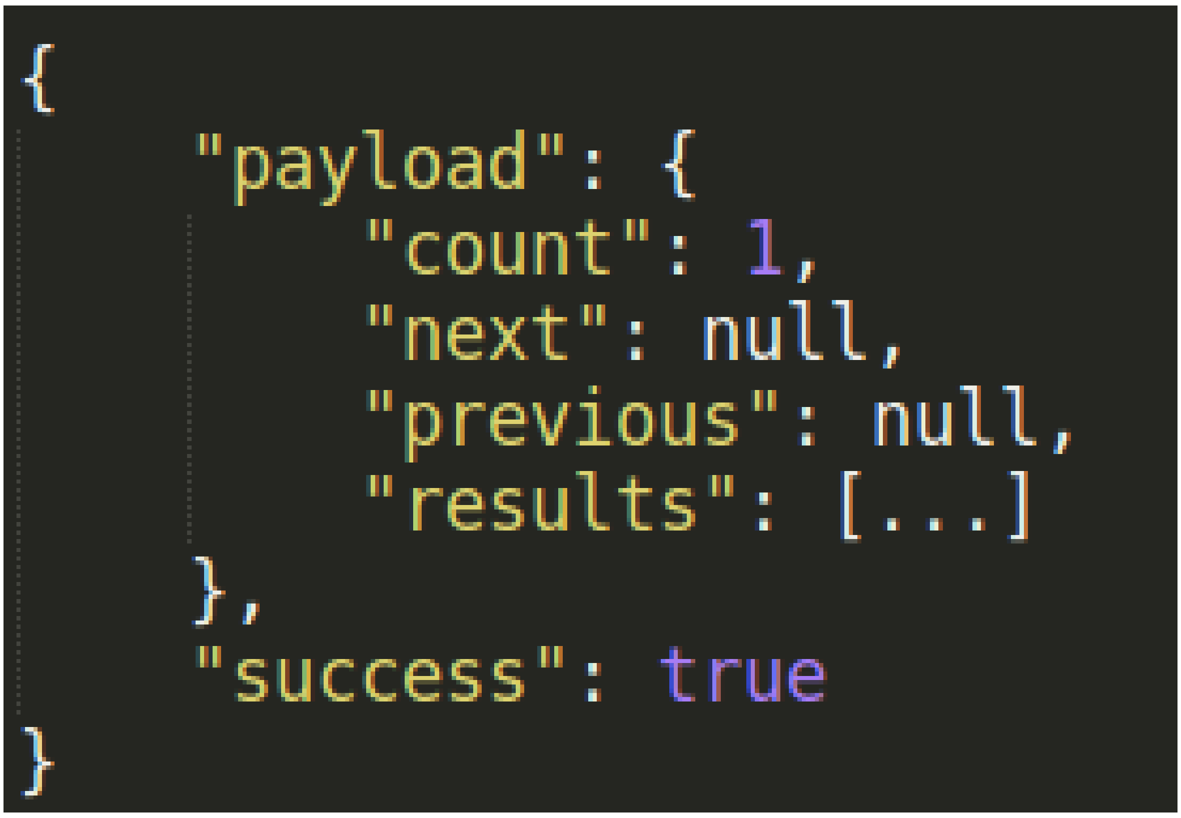
Figure A.13: Example request for getting task information by task name.

```
{
  "payload": {
    "name": "72bb0817-eb32-4101-86d6-1fe04342afbe",
    "cron_expr": null,
    "interval_type": "seconds",
    "interval_units": 5,
    "enabled": false,
    "trigger_time": "2017-11-20T19:32:23Z",
    "payload": "{\"task\": \"{\\\"task\\\": \\\"generate_daily_report\\\"}\", \"total_run_count\":
\"%1$d\", \"trigger_time\": \"%2$s\", \"task_name\": \"%3$s\"}",
    "expires": "2017-11-20T20:00:00Z",
    "last_run_at": null,
    "total_run_count": 0,
    "description": "New Periodic Task",
    "scheduler_id": 5,
    "run_once": false,
    "queue": "georgi-a_rabbitmq_queue_name",
    "exchange": "georgi-exchange",
    "routing_key": "georgi-a_routing_key",
    "recurrence": 24,
    "user": 4
  },
  "success": true
}
```

Figure A.14: Example response for getting task information by task name.

```
GET /tasks/get-tasks?page=1 HTTP/1.1
Host: localhost:8000
User-Agent: insomnia/5.12.4
Authorization: Token e92fef61202a5c05d376d57c3dff200c3798829b
Accept: */*
```

Figure A.15: Example request for getting task information for many tasks.



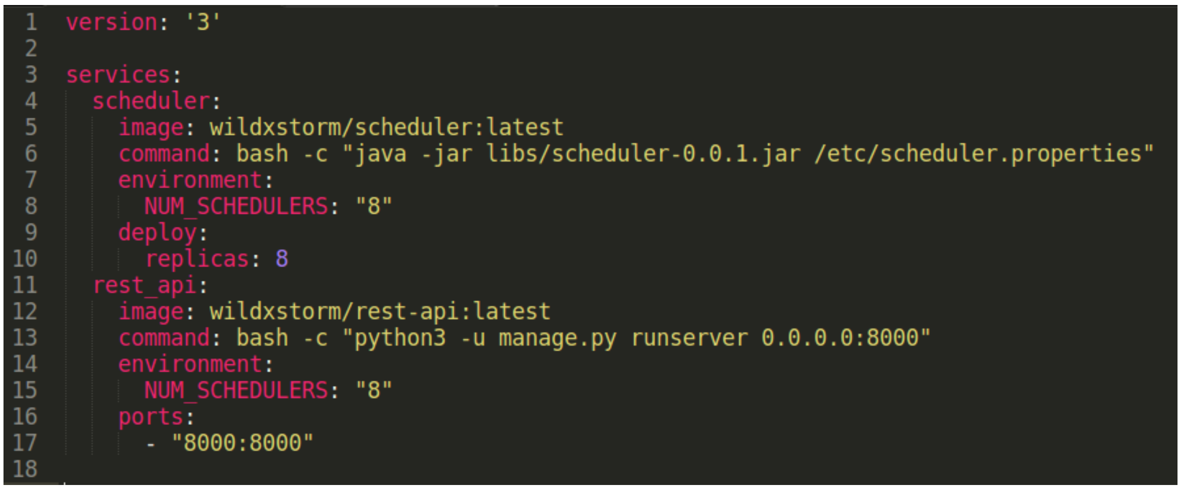
```
{
    "payload": {
        "count": 1,
        "next": null,
        "previous": null,
        "results": [...]
    },
    "success": true
}
```

Figure A.16: Example response for getting task information for many tasks.

Appendix B

Sample Docker-Compose File

The sample docker-compose file is shown on Figure B.1. There are 2 services in the application stack - **scheduler** and **rest_api**. The **image** parameter specifies the Docker image that will be pulled from the Docker Hub on the cluster nodes. **command** denotes the command to be executed in a container (that runs a given service) when it is spun-up. With **environment** it is possible to specify a set of environmental variables that will be accessible by the process running inside the container. In the example, both the scheduler and the System API will have access to **NUM_SCHEDULERS**, which denotes the number of replicas in the system. The **scheduler** service uses a **deploy** parameter that tells the orchestrator the number of replicas to deploy to the Swarm. Each replica is a single docker container that will run the **command** for the service. A final note is that the **ports** option that is used in the **rest_api** service. It is used to specify a port mapping between the cluster and the container. The example on Figure B.1 declares that any traffic sent to port 8000 on any node participating in the Swarm will be forwarded to port 8000 of the container in which the **rest_api** service is running.



```
version: '3'

services:
  scheduler:
    image: wildxstorm/scheduler:latest
    command: bash -c "java -jar libs/scheduler-0.0.1.jar /etc/scheduler.properties"
    environment:
      NUM_SCHEDULERS: "8"
    deploy:
      replicas: 8
  rest_api:
    image: wildxstorm/rest-api:latest
    command: bash -c "python3 -u manage.py runserver 0.0.0.0:8000"
    environment:
      NUM_SCHEDULERS: "8"
    ports:
      - "8000:8000"

```

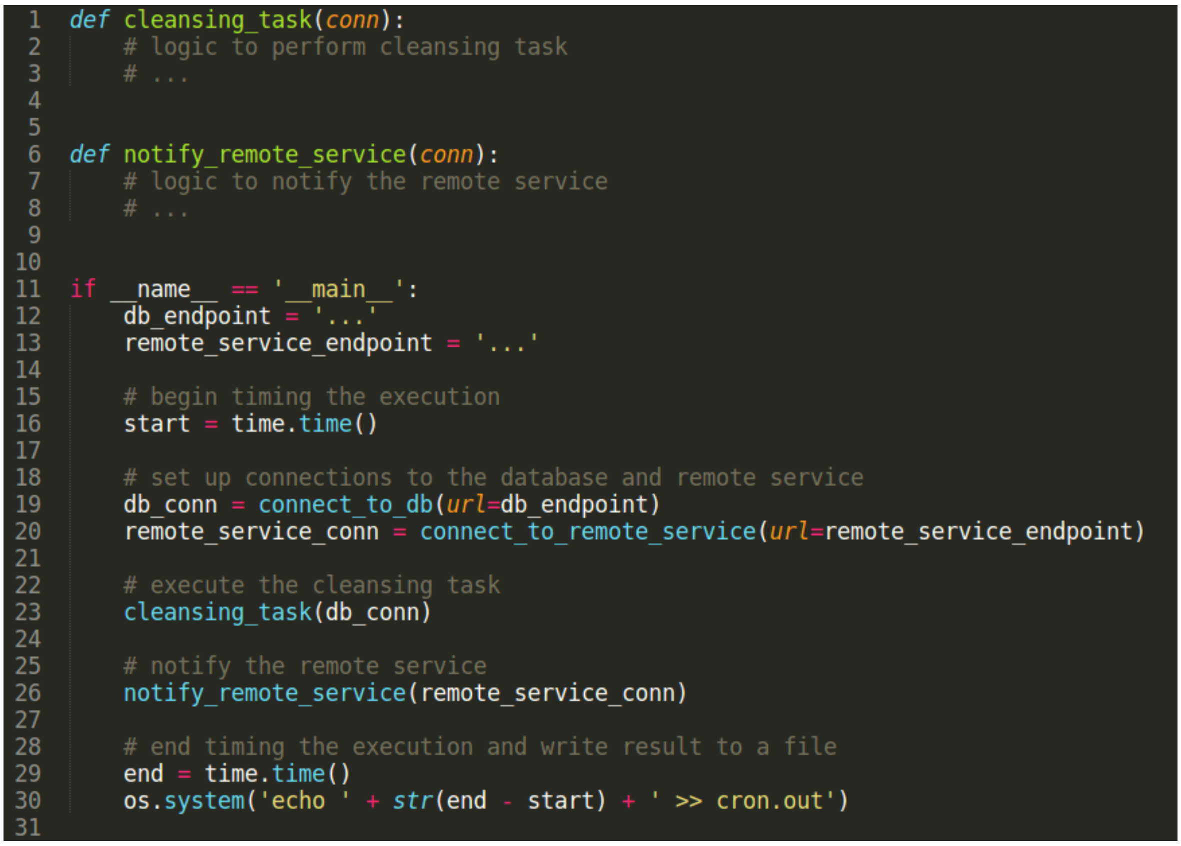
Figure B.1: Simplified Docker Compose File for the Application Stack.

Appendix C

Against cron

C.1 Simulating cron

Figure C.1 shows the pseudo code for the **cron.py** script. Before explaining how cron can be simulated through it, let's first take a look at what the script does. It defines 2 variables that specify the endpoint where the database and the remote service are running (respectively, **dp_endpoint** and **remote_service_endpoint**). Afterwards, lines 19 and 20 are responsible for setting up connections to the 2 external resources. The script then consecutively performs the cleansing task (line 23) and notifies the remote service of the successful completion (line 26). In the end, the time the whole operation took (including setting up the connections) is appended to a file (**cron.out**).



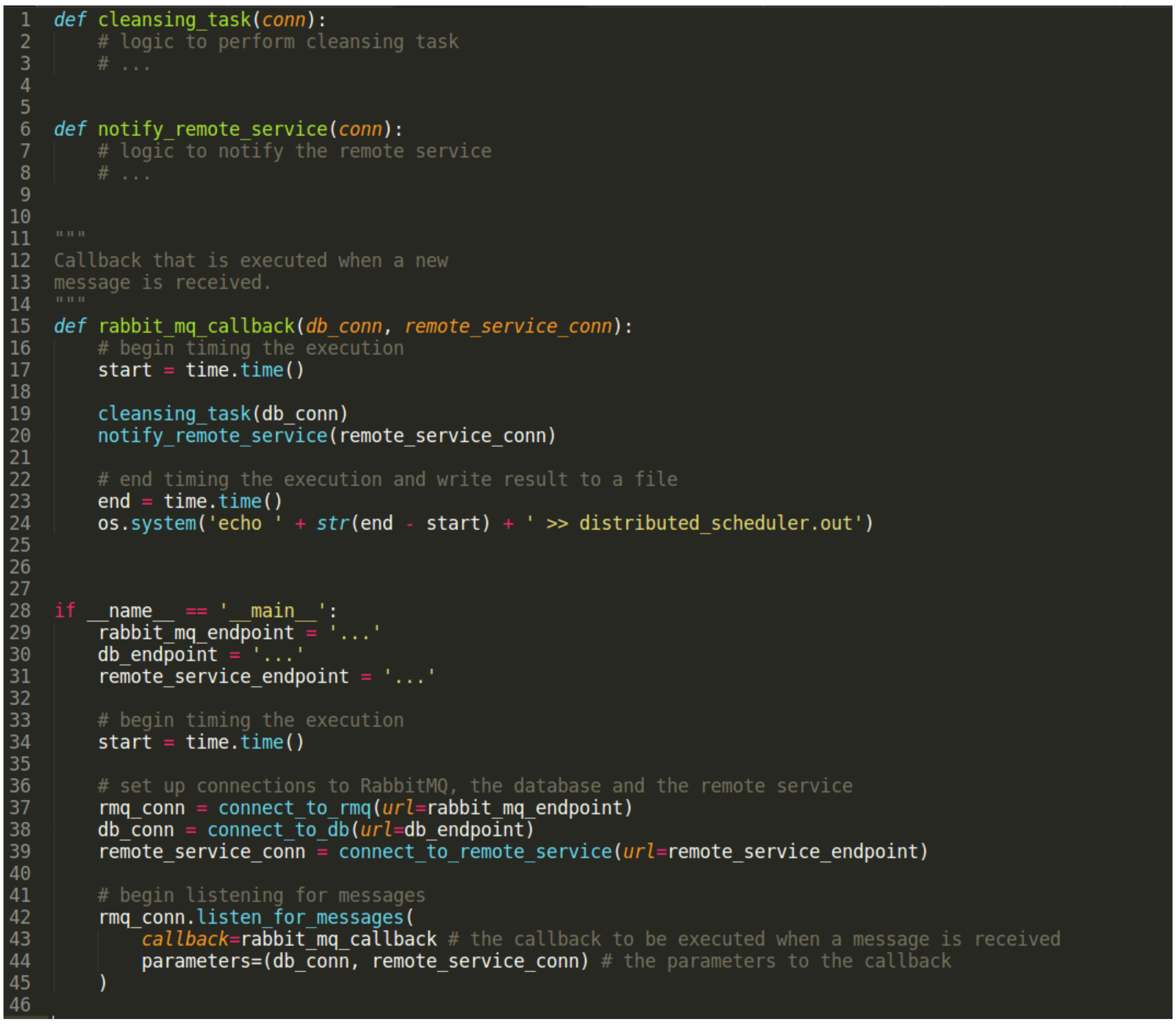
```
 1  def cleansing_task(conn):
 2      # logic to perform cleansing task
 3      # ...
 4
 5
 6  def notify_remote_service(conn):
 7      # logic to notify the remote service
 8      # ...
 9
10
11  if __name__ == '__main__':
12      db_endpoint = '...'
13      remote_service_endpoint = '...'
14
15      # begin timing the execution
16      start = time.time()
17
18      # set up connections to the database and remote service
19      db_conn = connect_to_db(url=db_endpoint)
20      remote_service_conn = connect_to_remote_service(url=remote_service_endpoint)
21
22      # execute the cleansing task
23      cleansing_task(db_conn)
24
25      # notify the remote service
26      notify_remote_service(remote_service_conn)
27
28      # end timing the execution and write result to a file
29      end = time.time()
30      os.system('echo ' + str(end - start) + ' >> cron.out')
31
```

Figure C.1: Pseudo Code for cron.py.

Considering the explanation of how cron operates given in Section 1.2.1, it can be seen that if **cron.py** is set-up as a cron job, crond will spawn a new process (when the job is due) in which to execute the script. Given this, for the simplicity of evaluation, instead of creating an actual cron job, we could simply run the script a given number of times. For example, if we want to see the total amount of time 100 executions of the job would take, we can simply run **python cron.py** 100 times (using a wrapper script for convenience). Afterwards, we can sum all the numbers in **cron.out** to get the desired value.

C.2 Simulating the Distributed Scheduler

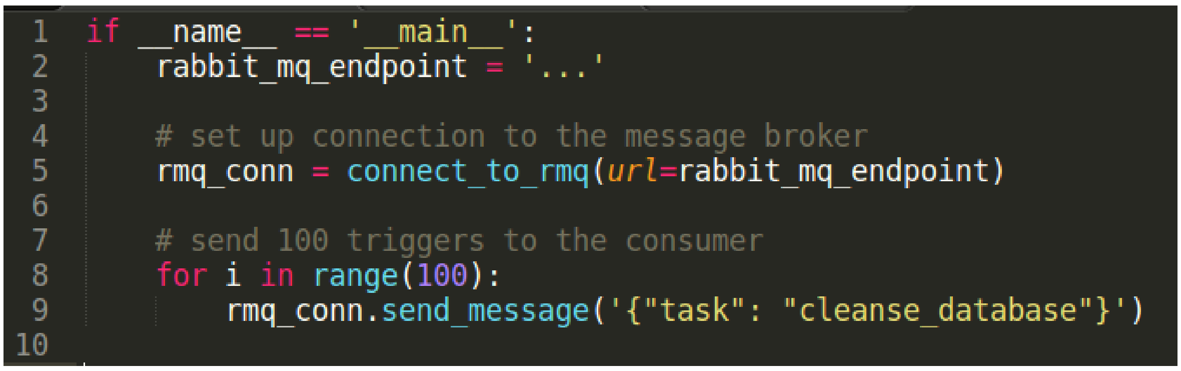
The pseudo code in Figure C.2 is slightly different from that in the previous one. It again starts by setting up connections to the external resources. However, it now begins listening for messages on the message broker (line 42) and specifies a callback to be executed once a message is received. This callback (**rabbit_mq_callback** defined on line 15) takes as parameters the connection objects to the database and the remote service. It then performs the cleansing task and notifies the remote service (lines 19 and 20). In the end, it writes the amount of time the 2 operations took to a file (**distributed_scheduler.out**). It is very important to understand that it is not needed to record the amount of time connection establishment took (unlike with cron.py) simply because the price for this is paid exactly once. To quickly summarize, the script is run once, and then the callback will be executed whenever a new trigger message is received.



```
1   def cleansing_task(conn):
2       # logic to perform cleansing task
3       # ...
4
5
6   def notify_remote_service(conn):
7       # logic to notify the remote service
8       # ...
9
10
11  """
12  Callback that is executed when a new
13  message is received.
14  """
15  def rabbit_mq_callback(db_conn, remote_service_conn):
16      # begin timing the execution
17      start = time.time()
18
19      cleansing_task(db_conn)
20      notify_remote_service(remote_service_conn)
21
22      # end timing the execution and write result to a file
23      end = time.time()
24      os.system('echo ' + str(end - start) + ' >> distributed_scheduler.out')
25
26
27
28  if __name__ == '__main__':
29      rabbit_mq_endpoint = '...'
30      db_endpoint = '...'
31      remote_service_endpoint = '...'
32
33      # begin timing the execution
34      start = time.time()
35
36      # set up connections to RabbitMQ, the database and the remote service
37      rmq_conn = connect_to_rmq(url=rabbit_mq_endpoint)
38      db_conn = connect_to_db(url=db_endpoint)
39      remote_service_conn = connect_to_remote_service(url=remote_service_endpoint)
40
41      # begin listening for messages
42      rmq_conn.listen_for_messages(
43          callback=rabbit_mq_callback # the callback to be executed when a message is received
44          parameters=(db_conn, remote_service_conn) # the parameters to the callback
45      )
46
```

Figure C.2: Pseudo Code for distr_scheduler.py.

The explanation in the previous paragraph describes (in an abstract way) what a consumer of the Distributed Scheduler's messages has to do. Now, for the sake of simplifying the evaluation, we do not really need to have the Distributed Scheduler hooked up, so that it can trigger task executions. Instead, a simple producer script that connects to the message broker and sends messages to the consumer is sufficient. The pseudo code of a sample producer (called **simple_producer.py**) is shown in Figure C.3. Now, if we again want to see how long 100 executions of the periodic job would take, all wee need to do is (1) run **distr_scheduler.py** and (2) send 100 triggers via **simple_producer.py**. In the end, we sum all values in **distributed_scheduler.out**. It is important to note that the metrics on Figure 5.2 were obtained by subtracting (for a given number of tasks) the total amount of time it took cron from the corresponding value in the case of the Distributed Scheduler..



```
if __name__ == '__main__':
    rabbit_mq_endpoint = '...'

    # set up connection to the message broker
    rmq_conn = connect_to_rmq(url=rabbit_mq_endpoint)

    # send 100 triggers to the consumer
    for i in range(100):
        rmq_conn.send_message('{"task": "cleanse_database"}')

```

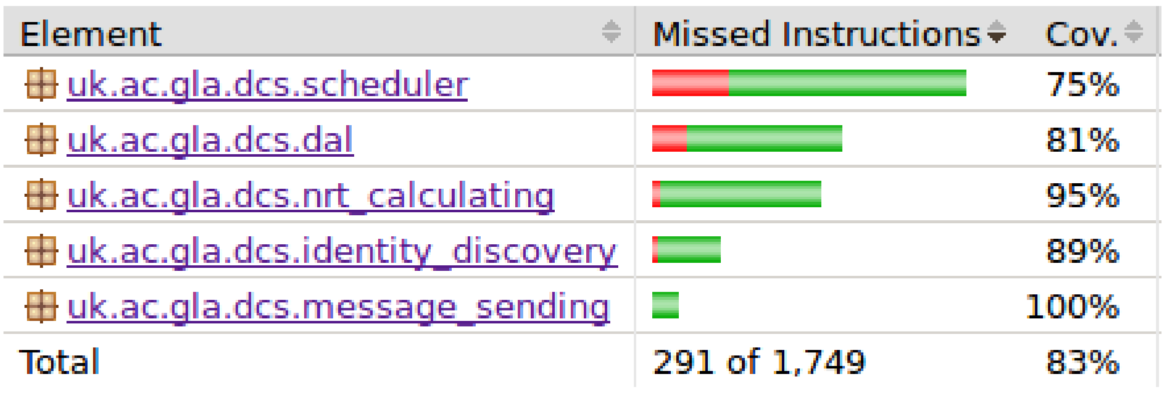
Figure C.3: Pseudo Code for simple_producer.py.

Appendix D

Code Coverage

D.1 Code Coverage for the Scheduler

Figure D.1 shows the code coverage of the scheduler service as reported by **JaCoCo** - the test coverage tool used for the Java project [41]. It can be seen that all, but the **uk.ac.gla.dcs.scheduler** package, meet the goal, set in Section 5.1.3 (80-90 % coverage). The reason for this is that the package contained many methods that would not add value to the test suite (e.g. getters, setters, etc.). Consequently, those were not tested. Other code coverage tools (such as the one used for the System API, as described below) allow the developer to mark such methods as **noqa** - not for QA. This feature, however, was missing in the version of JaCoCo used while working on the Distributed Scheduler.

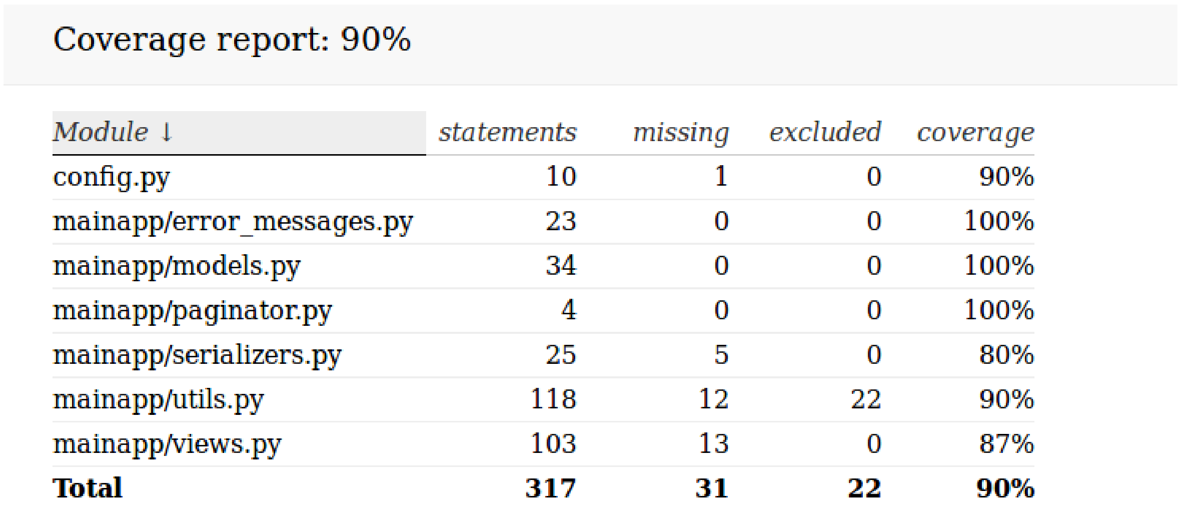


Element	Missed Instructions	Cov.
uk.ac.gla.dcs.scheduler		75%
uk.ac.gla.dcs.dal		81%
uk.ac.gla.dcs.nrt_calculating		95%
uk.ac.gla.dcs.identity_discovery		89%
uk.ac.gla.dcs.message_sending		100%
Total	291 of 1,749	83%

Figure D.1: Scheduler Service Coverage.

D.2 Code Coverage for the System API

The test coverage tool used for the System API was **coverage.py** [25]. Figure D.2 show the results. It can be seen that all modules achieve the goal from Section 5.1.3. One of the main reasons the results for the System API are higher is the finer noqa granularity of coverage.py. It allowed for methods to be excluded from the reports, and as a result utilities that do not provide much functions were not tested.



Coverage report: 90%

Module ↓	*statements*	*missing*	*excluded*	*coverage*
config.py	10	1	0	90%
mainapp/error_messages.py	23	0	0	100%
mainapp/models.py	34	0	0	100%
mainapp/paginator.py	4	0	0	100%
mainapp/serializers.py	25	5	0	80%
mainapp/utils.py	118	12	22	90%
mainapp/views.py	103	13	0	87%
Total	**317**	**31**	**22**	**90%**

Figure D.2: System API Coverage.

Bibliography

[1] AddItem - API Reference - Trading API. `https://developer.ebay.com/devzone/xml/docs/reference/ebay/additem.html`. [Online; Accessed on 20.02.2018].

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition*. John Wiley & Sons.

[3] Amazon EC2 Instance Types - Amazon Web Services (AWS). `https://aws.amazon.com/ec2/instance-types/`. [Online; Accessed on 25.02.2018].

[4] Amazon Web Services (AWS) - Cloud Computing Services. `https://aws.amazon.com/`. [Online; Accessed on 20.02.2018].

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, n.j., apr. 18 x2013;20), afips press, reston, va., 1967, pp. 483 x2013;485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, Summer 2007.

[6] AngularJS - Superheroic JavaScript MVW Framework. `https://angularjs.org/`. [Online; Accessed on 29.01.2018].

[7] Apache Curator. `https://curator.apache.org/`. [Online; Accessed on 08.02.2018].

[8] Apache Kafka. `https://kafka.apache.org/`. [Online; Accessed on 05.02.2018].

[9] Apache Zookeeper. `https://zookeeper.apache.org/`. [Online; Accessed on 06.02.2018].

[10] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 10(4):337–348, November 2016.

[11] Mike Ashworth. The potential of java for high performance applications. 02 1970.

[12] A. J. C. Bik, M. Girkar, and M. R. Haghighat. Innovative architecture for future generation high-performance processors and systems. pages 87–94, Oct 1998.

[13] Boost C++ Libraries. `http://www.boost.org/`. [Online; Accessed on 10.03.2018].

[14] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, RFC Editor, 1997.

[15] E. Brewer. Towards robust distributed systems, 2000.

[16] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. 2006.

[17] Celery: Distributed Task Queue. `http://www.celeryproject.org/`. [Online; Accessed on 28.01.2018].

[18] Narciso Cerpa and June M. Verner. Why did your project fail? *Commun. ACM*, 52(12):130–134, December 2009.

[19] F. Chang, J. Dean, S. Ghemewat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. 2006.

[20] D. Chappell. *Enterprise Service Bus: Theory in Practice*. O'Reilly Media.

[21] M. Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley.

[22] Container distribution strategies - Docker Documentation. `https://docs.docker.com/docker-cloud/infrastructure/deployment-strategies/`. [Online; Accessed on 20.02.2018].

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press.

[24] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design, Fifth Edition*. Addison-Wesley.

[25] Coverage.py. `https://coverage.readthedocs.io/en/coverage-4.5.1/`. [Online; Accessed on 08.03.2018].

[26] L. Crispin and J. Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley, 2009.

[27] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, July 2006.

[28] Cron - Wikipedia. `https://en.wikipedia.org/wiki/Cron`. [Online; Accessed on 28.01.2018].

[29] R. Cziva and D. P. Pezaros. Container network functions: Bringing nfv to the network edge. *IEEE Communications Magazine*, 55(6):24–31, 2017.

[30] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 186–189, New York, NY, USA, 2002. ACM.

[31] Database-as-IPC - Wikipedia. `https://en.wikipedia.org/wiki/Database-as-IPC`. [Online; Accessed on 29.01.2018].

[32] Dependency hell - Wikipedia. `https://en.wikipedia.org/wiki/Dependency_hell`. [Online; Accessed on 28.02.2018].

[33] Django Official Website. `https://www.djangoproject.com/`. [Online; Accessed on 20.02.2018].

[34] Django REST Framework. `http://www.django-rest-framework.org/`. [Online; Accessed on 20.02.2018].

[35] Django Templates Documentation. `https://docs.djangoproject.com/en/2.0/topics/templates/`. [Online; Accessed on 29.01.2018].

[36] Docker - Build, Ship, and Run Any App, Anywhere. `https://www.docker.com/`. [Online; Accessed on 20.02.2018].

[37] Docker Cloud - Build, Ship and Run any App, Anywhere. `https://cloud.docker.com/`. [Online; Accessed on 20.02.2018].

[38] Docker Compose JUnit Rule. `https://github.com/palantir/docker-compose-rule`. [Online; Accessed on 25.02.2018].

[39] Docker Hub. `https://hub.docker.com/`. [Online; Accessed on 20.02.2018].

[40] docker stack deploy - Docker Documentation. https://docs.docker.com/engine/reference/commandline/stack_deploy/. [Online; Accessed on 20.02.2018].

[41] EclEmma - JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco/. [Online; Accessed on 08.03.2018].

[42] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems, Sixth Edition*. O. Reilly Media.

[43] L. Farias. The Importance of Scalability In Software Design. https://conceptainc.com/blog/importance-of-scalability-in-software-design/. [Online; Accessed on 28.02.2018].

[44] R.T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. 2000.

[45] M. Fowler. TestCoverage. https://martinfowler.com/bliki/TestCoverage.html. [Online; Accessed on 08.03.2018].

[46] GitLab Official Website. https://about.gitlab.com/. [Online; Accessed on 05.02.2018].

[47] Google Kubernetes Engine - Google Cloud Platform. https://cloud.google.com/kubernetes-engine/. [Online; Accessed on 20.02.2018].

[48] Gradle Build Tool. https://gradle.org/. [Online; Accessed on 06.02.2018].

[49] Grafana - The open platform for analytics and monitoring. https://grafana.com/. [Online; Accessed on 25.02.2018].

[50] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing, Second Edition*. Addison Wesley.

[51] S. Haloi. *Apache Zookeeper Essentials*. Packt Publishing.

[52] D. Hellmann. *The Python Standard Library by Example*. Addison-Wesley.

[53] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. O. Reilly Media.

[54] J. Hugg. Recipe: Combine At-Least-Once Delivery with Idempotent Processing to Achieve Exactly-Once Semantics. https://www.linkedin.com/pulse/recipe-combine-at-least-once-delivery-idempotent-processing-john-hugg/. [Online; Accessed on 20.02.2018].

[55] Inner-platform effect - Wikipedia. https://en.wikipedia.org/wiki/Inner-platform_effect. [Online; Accessed on 15.03.2018].

[56] Is Flask good enough to develop large applications? https://www.reddit.com/r/Python/comments/2jja20/is_flask_good_enough_to_develop_large_applications/. [Online; Accessed on 20.02.2018].

[57] D. Janzen and H. Saiedian. Does test-driven development really improve software design quality? *IEEE Software*, 25(2):77–84, March 2008.

[58] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, June 1984.

[59] Java Callable Interface. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html. [Online; Accessed on 08.02.2018].

[60] Java Executor Service. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html. [Online; Accessed on 08.02.2018].

[61] Java JDBC Official Documentation. `https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html`. [Online; Accessed on 08.02.2018].

[62] Java Thread. `https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html`. [Online; Accessed on 06.03.2018].

[63] Java Timer. `https://docs.oracle.com/javase/8/docs/api/java/util/Timer.html`. [Online; Accessed on 06.03.2018].

[64] A. Jesse. Grok the gil: How to write fast and thread-safe python. PyCon 2017, 2017.

[65] M. Jones. What is the inner-platform effect? `https://www.exceptionnotfound.net/the-inner-platform-effect-anti-pattern-primers`. [Online; Accessed on 15.03.2018].

[66] M. Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. W. Pollock.

[67] Dounia Khaldi, Pierre Jouvelot, Corinne Ancourt, and François Irigoin. Task parallelism and data distribution: An overview of explicit parallel programming languages. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, pages 174–189, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[68] Kubernetes - Production-Grade Container Orchestraton. `https://kubernetes.io/`. [Online; Accessed on 20.02.2018].

[69] S. D. Label and E. Steegmans. On the promises and challenges of event-driven service oriented architectures.

[70] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[71] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4/3:382–401, July 1982.

[72] R. Love. *Linux System Programming: Talking Directly to the Kernel and C Library, First Edition*. O'Reilly Media.

[73] Manifesto for Agile Software Development. `http://agilemanifesto.org/`. [Online; Accessed on 05.02.2018].

[74] B. Marick. How to Misuse Code Coverage. 1999.

[75] S.N. Mehmood, N. Haron, V. Akhtar, and Y. Javed. Implementation and Experimentation of Producer - Consumer Synchronization Problem. *International Journal of Computer Applications*, 14(3), 01 2011.

[76] M. Mikowski and J. Powell. *Single Page Web Applications: JavaScript end-to-end*. Manning Publications.

[77] Mockito framework site. `http://site.mockito.org/`. [Online; Accessed on 25.02.2018].

[78] A. Mouat. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly.

[79] G. J. Myers. *The Art of Software Testing, Second Edition*. Willey, 2004.

[80] MySQL. `https://www.mysql.com/`. [Online; Accessed on 28.01.2018].

[81] N. Nerkhede. Exactly-once Semantics are Possible: Heres How Kafka Does it. `https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/`. [Online; Accessed on 20.02.2018].

[82] Network Interface Controller - Wikipedia. https://en.wikipedia.org/wiki/Network_interface_controller. [Online; Accessed on 30.01.2018].

[83] Nodes - Kubernetes. https://kubernetes.io/docs/concepts/architecture/nodes/. [Online; Accessed on 20.02.2018].

[84] Operating-system-level virtualization - Wikipedia. https://en.wikipedia.org/wiki/Operating-system-level_virtualization. [Online; Accessed on 20.02.2018].

[85] Overview of Docker Compose - Docker Documentation. https://docs.docker.com/compose/overview/. [Online; Accessed on 20.02.2018].

[86] Monica Peshave and Kamyar Dezhgosha. HOW SEARCH ENGINES WORK AND A WEB CRAWLER APPLICATION. 01 2018.

[87] Plans & Pricing - CloudAMQP. https://www.cloudamqp.com/plans.html. [Online; Accessed on 29.01.2018].

[88] Pods - Kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/pod/. [Online; Accessed on 20.02.2018].

[89] PostgreSQL - Row Level Locks. https://www.postgresql.org/docs/9.5/static/explicit-locking.html#LOCKING-ROWS. [Online; Accessed on 08.02.2018].

[90] PostgreSQL Official Website. https://www.postgresql.org/. [Online; Accessed on 28.01.2018].

[91] Principles behind the Agile Manifesto. http://agilemanifesto.org/principles.html. [Online; Accessed on 05.02.2018].

[92] Prometheus - Monitoring system & time series database. https://prometheus.io/. [Online; Accessed on 25.02.2018].

[93] Python json module. https://docs.python.org/3/library/json.html. [Online; Accessed on 20.02.2018].

[94] Quartz Enterprise Job Scheduler. http://www.quartz-scheduler.org/. [Online; Accessed on 28.01.2018].

[95] RabbitMQ - Messaging that just works. https://www.rabbitmq.com/. [Online; Accessed on 05.02.2018].

[96] RabbitMQ user permission to pub/sub on a pre-created queue. https://stackoverflow.com/questions/32379506/rabbitmq-user-permission-to-pub-sub-on-a-pre-created-queue. [Online; Accessed on 20.02.2018].

[97] React - A JavaScript library for building user interfaces. https://reactjs.org/. [Online; Accessed on 29.01.2018].

[98] Redis. https://redis.io/. [Online; Accessed on 08.02.2018].

[99] A. Ruiz. A Closer Look at JUnit Categories. https://dzone.com/articles/closer-look-junit-categories. [Online; Accessed on 25.02.2018].

[100] P.J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley.

[101] Scale the service in the swarm - Docker Documentation. https://docs.docker.com/engine/swarm/swarm-tutorial/scale-service/. [Online; Accessed on 20.02.2018].

[102] B. Scholl, T. Swanson, and D. Fernandez. *Microservices with Docker on Microsoft Azure*. Addison-Wesley.

[103] B. Schwartz. 5 subtle ways you're using MySQL as a queue, and why it'll bite you. `https://www.engineyard.com/blog/5-subtle-ways-youre-using-mysql-as-a-queue-and-why-itll-bite-you`, 2011. [Online; Accessed on 29.01.2018].

[104] J. Shore and S. Warden. *The Art of Agile Development*. O'Reilly.

[105] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. 2004.

[106] A. Shvets. *Design Patterns Explained Simply*.

[107] A. Silberschatz, P.B. Galvin, and G.Gagne. *Operating System Concepts, Ninth Edition*. Willey.

[108] I. Sommerville. *Software Engineering, Ninth Edition*. Pearson Education.

[109] A. Stellman and J. Greene. *Learning Agile: Understanding Scrum, XP, Lean, and Kanban*. O'Reilly Media.

[110] W.R. Stevens, B.Fenner, and A.M.Rudoff. *Unix Network Programming, Volume 1: The Sockets Networking API, Third Edition*. Addison-Wesley.

[111] T. Storer and J. Singer. *Lecture Notes on Software Engineering*.

[112] Swarm mode overview - Docker Documentation. `https://docs.docker.com/engine/swarm/`. [Online; Accessed on 20.02.2018].

[113] T. Treat. You Cannot Have Exactly-Once Delivery - Brave New Geek. `https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/`. [Online; Accessed on 20.02.2018].

[114] Understanding JIT Compilation and Optimizations. `https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/underst_jit.html`. [Online; Accessed on 05.02.2018].

[115] Updating Multiple Database Rows Quickly and Easily. `http://techblog.net-a-porter.com/2013/08/dbixmultirow-updating-multiple-database-rows-quickly-and-easily/`. [Online; Accessed on 08.02.2018].

[116] A. Videla and J.J.W. Williams. *RabbitMQ in Action: Distributed Messaging For Everyone*. Manning Publications.

[117] D. A. Watt and W. Findlay. *Programming Languages Design Concepts*. John Wiley & Sons.

[118] What exactly is server and client side pagination?? `https://stackoverflow.com/questions/2807755/what-exactly-is-server-and-client-side-pagination`. [Online; Accessed on 15.03.2018].

[119] What is a Sprint Backlog? `https://www.scrum.org/resources/what-is-a-sprint-backlog`. [Online; Accessed on 15.03.2018].

[120] What is a Sprint in Scrum? `https://www.scrum.org/resources/what-is-a-sprint-in-scrum`. [Online; Accessed on 15.03.2018].

[121] What is Sprint Planning? `https://www.scrum.org/resources/what-is-sprint-planning`. [Online; Accessed on 15.03.2018].

[122] T. Wrobel. Run your tests on production! `https://blog.arkency.com/2017/01/run-your-tests-on-production/`. [Online; Accessed on 25.02.2018].