Building scalable digital library ingestion pipelines using microservices

Matteo Cancellieri (0000-0002-9558-9772), Nancy Pontika (000-0002-2091-0402), Samuel Pearce (0000-0001-5616-7000), Lucas Anastasiou (0000-0002-1587-5104), and Petr Knoth (0000-0003-1161-7359)

CORE, The Open University, Milton Kenyes, MK7 6AA, UK,
theteam@core.ac.uk,
WWW home page: https://core.ac.uk/

Abstract. CORE, a harvesting service offering access to millions of open access research papers from around the world, has shifted its harvesting process from following a monolithic approach to the adoption of a microservices infrastructure. In this paper, we explain how we rearranged and re-scheduled our old ingestion pipeline, present CORE's move to managing microservices and outline the tools we use in a new and optimised ingestion system. In addition, we discuss the inefficiencies of our old harvesting process, the advantages, and challenges of our new ingestion system and our future plans. We conclude that via the adoption of microservices architecture we managed to achieve a scalable and distributed system that would assist with CORE's future performance and evolution.

Keywords: Harvesting •Repositories •Microservices •Infrastructure •Software Architecture

1 Introduction

Aggregators are being used in many aspects of everyday life, from newspapers to traveling websites and from movies' reviews to social networking services. In the constantly growing scholarly communications environment, plenty of aggregators were developed due to the large amount of scientific knowledge published each year and the scholars' need to discover and extract the knowledge included in them. Moreover, the recent shift of sciences to interdisciplinarity created the need for a seamless tool that would make the retrieval of scientific literature from various subject fields possible.

Aggregators can collect, enrich and clean metadata to harmonise their access, allow a uniform search across a variety of platforms increasing the content's visibility, and bring to the end user an advanced discovering experience, by showcasing new trends in sciences and growth [1]. Due to their role, aggregators must be capable of processing large amounts of data and be developed in a scalable and sustainable infrastructure over time.

The transition process to microservices is described via a real life scenario, the CORE project, a global harvesting service aggregating millions of open access research papers. In the past, CORE's harvesting infrastructure was designed in a rather monolithic approach. Even though the scaling up and the continuation of its use was possible, the architecture suffered from complexity and maintenance issues, especially when facing a larger and extended amount of data, and had strong interdependent components that challenged its sustainability. Being in a need to restructure the current system and make it easier to scale, we introduced a microservices architecture, which is defined as small, autonomous components in a larger infrastructure that harmoniously work together [2]. This technology is lately widely used due to the advancement of the available software and resources. A benefit of the microservices is that programmers are able to efficiently focus on the implementation of the components performed in a single brief task.

In this paper we describe the evolution of a monolithic harvesting infrastructure to the creation of a microservices oriented architecture. Although migrating from a monolithic architecture to a microservices one is already covered in the literature, we believe that our contribution is worth to be described; our specific use case is applied to a combination of metadata harvesting, full text collection, crawling and enrichments of research outputs. The main contributions of this paper are outlined as follows:

- Define the requirements for designing a scalable aggregation infrastructure.
- Guided by the requirements, we propose a design based on microservices that can be applied to any aggregation and digital library infrastructure.
- Describe our experience migrating from a monolithic architecture to a microservices orientend one, in a real life scenario of the CORE project.

1.1 Related work

The design of an aggregation system is usually conducted from square one, due to the variety of the application scenario per each project (or use case), which is defined both by the availability of resources and the flexibility of the distribution [3]. So far a number of architectural solutions have been released; some of them focusing on specific use cases, while others aim to address and solve the lack of reusable infrastructures. The D-NET component infrastructure, which is developed and implemented by the OpenAIRE project [4], addresses similar complexities to the ones we are encountering in our harvesting process and their proposed solution is very similar to CORE's infrastructure. The harvesting is accomplished using the Open Archives Initiative Metadata Harvesting Protocol (OAI-PMH) [5] and the harvested metadata are converted into XML files, creating a graph model information structure. Nonetheless, OpenAIRE's aim is to connect publications, data set repositories and Current Research Information Systems (CRIS) [6], therefore its main interest is to collect and enrich metadata only. On the other hand, CORE's focus lies both in crawling metadata pages and discovering the full text.

Another aggregation infrastructure has been implemented by the SHARE project [7]. Although the project focuses more on aggregating information about

research activity metadata, its pipeline and the technologies used have main similarities with the processes we are describing in this paper. For example, the SHARE infrastructure uses RabbitMQ [8] as a messaging system and a Celery scheduler[9], which arranges how the workers will be collecting, normalising and making the content searchable via the ElasticSearch [10] index. Nonetheless, SHARE's architecture does not extend to providing tools for crawling and enriching full text documents as well. The BASE project [11], is another metadata aggregator facilitating search via repositories. The harvesting infrastructure implemented by BASE is similar to the aforementioned aggregation infrastructures; data is harvested via the OAI-PMH endpoint and the metadata are exposed and made searchable through the SOLR [12] index.

Commercial solutions, such as Google Scholar and Microsoft Academic Graph, use search engine crawl facilities to recognise full text and index it. Nonetheless, both of them limit user access at the granularity level with no access to raw data[13]. CiteSeerX [14], a metadata and full text harvesting service, defines three components of a scalable system that is capable of crawling citations from full text; harvester, document archive and search interface. With this system, the issue of scalability is approached horizontally, mirroring the servers and the locations that scale up the system.

All the aforementioned projects describe scalable architectures, but, there is no specific reference explaining a transition from a monolithic architecture to a microservices approach. In addition, CORE's case is unique, because it combines a) metadata harvesting, b) full text crawling and c) enrichments in the same infrastructure while the other projects architectures are mostly oriented on aggregating metadata and enriching metadata. Furthermore, no previous work refers to scalability from the aggregation point of view of harvesting and enriching the content; the aforementioned literature describes the scalability approach following the front end availability only.

1.2 Real-life scenario: CORE

CORE harvests open access (OA) journals and repositories, institutional and disciplinary, from all over the world and provides seamless access to millions of OA research papers. (OA is defined as content offered digital, online, free of charge and free of most copyright restrictions [15]). While the OA movement is gradually growing, the amount of available scientific information follows this growth. CORE attempts to address this flourishing availability of OA content by aggregating large volumes of OA research papers and offers them via its search engine and other services, e.g. API and Dataset. As of March 2017, CORE harvested content from 6,000 OA journals and 2,437 repositories and offered access to 70 million metadata records and over 6 million full text PDFs. In addition, it provides its content via three access levels [13] and demonstrates the granulated structure of the raw data, the scientific papers and the collections as a whole. By taking into consideration the various research stakeholders who can benefit from CORE, the service offers 1) raw data access, 2) transactional and 3) analytical information access, addressing the needs of 1) text miners, digital

libraries, developers, 2) researchers, students, general public and 3) funders and repository managers, respectively.

To collect its metadata, CORE uses the OAI-PMH, a widely supported protocol for collecting and exposing metadata records. In most cases, these metadata are mostly formatted using the Dublin Core schema [16]. In addition, CORE expands its use by supporting other standards as well, such as the Metadata Encoding and Transmission Standard [17], the RIOXX Metadata Application Profile [18], and the OpenAIRE Guidelines for Literature Repositories [4].

CORE is not merely a metadata aggregation service, but it expands also to aggregating and caching the full text. In an effort to solve the lack of an existing standard for harvesting a paper's full text, CORE has established a methodology and follows various procedures: a) recognising the full-text links in the metadata record, b) performing a crawling technique from an online resource to derive to the link of a specific paper, c) composing the full-text link by using recognised patterns, which are obtained by analysing the structure of our data providers and d) following other custom built approaches, whereby the machine readable structures of specific content providers are being examined.

Until recently, CORE was operating in a rather monolithic approach. Even though it was in position to process simultaneously several repositories, the harvesting pipeline was centralised around each repository. All these years CORE has progressed and custom solutions were applied to: a) improve the harvesting process and growth of our collection, b) address technology advancement issues, such as repository software updates and the establishment of new metadata profiles like RIOXX, and c) assist with the integration of new and emerging services. The accumulation of all these lines of code and the unavoidable high code coupling, would often prove to be difficult to manage, require an extra effort of constant fixes and affect the capabilities and strengths of our performance. As a result, we experienced challenges in efficiently managing all the existing services and fear that the quality of our services would decline. Availability was also an issue; the services designed for public use were strongly coupled to the harvesting infrastructure, thus an overload of the system while harvesting would impact directly the overall availability of the service.

1.3 Ingestion pipeline

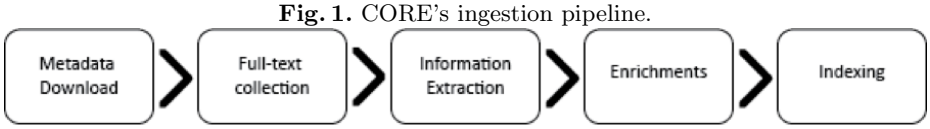
CORE's ingestion pipeline currently consists of five tasks, but is in principle extensible to more tasks. The process can be described as a pipeline, a concept first introduced in computer science by the UNIX operating system and then extended to software architectures models as for example in Doberkat [19], and on a context similar to our use case by Abrams et al. [20]. Each task performs a certain action, while the result of each one of them creates the corpus of a full record, including of metadata and full text, available in the CORE collection (Figure 1).

1. Metadata Download

(a) Extraction: The metadata of a journal or repository, institutional or disciplinary, (thereafter called only "repository") are downloaded and extracted in the CORE database for local storage.
(b) Cleaning: The data is being processed for cleaning, such as performing an author name normalisation. The data is also standardised and normalised across the various supported standards.

2. Full-text download: CORE downloads and stores a cached version of the downloaded full-text in its database.
3. Information extraction
   (a) Text extraction: To enable full text searching, the downloaded PDFs are analysed and the full text is extracted into a text file.
   (b) Language detection: Performed to offer a filtering option for advanced searches.
   (c) Citation extraction: CORE cross-matches whether the referenced paper is available and provided by our service. If the paper is available, then the two papers are linked. If it is not, CORE receives the referenced papers Digital Object Identifier (DOI) via the CrossRef service [21].
4. Enrichment
   (a) Duplicates detection: CORE detects duplicate records and marks these files in its database.
   (b) Related Content Identification: With the use of information retrieval procedures, CORE matches the semantically related papers. (CORE's recommender, a plug-in for repositories and open journal systems, is highly dependent on this task.)
5. Indexing: This is the last step in the ingestion pipeline, which empowers the search functionality, while it supports the use of the CORE API and the CORE Dataset.

Fig. 1. CORE's ingestion pipeline.



2 Scalable infrastructure requirements

Abstracting from our real life use case, we have defined a set of requirements that are generic and can be used in any aggregation or digital library scenario. Taking into consideration the features of the CiteSeerX [14] architecture, we built our requirements on a generic distributed system and we extended them focusing on systems that follow specific workflows and need to interact with a number of external services. In the initial phase we developed a set of requirements that the infrastructure had to support:

- **Easy to maintain**: The solution should be easy to manage and the code should be accessible for maintenance, fixes, and improvements.
- **High levels of automatisation**: We should be able to achieve a completely automated harvesting process allowing also a manual interaction.
- **Fail fast**: Items in the pipeline should be validated immediately after a task is performed, instead of having only one and final validation at the end of the pipeline. This has the benefit of recognising issues early in the process and programmers are notified earlier of possible failures and issues.
- **Easy to troubleshoot**: Possible bugs in the code should be easily discerned.
- **Distributed and scalable**: The addition of new nodes in the system should allow scalability, be transparent and replicable.
- **No single point of failure**: A single crash should not affect the whole ingestion pipeline, but tasks should work independently.
- **High availability**: The infrastructure containing services designed for public use should not be attached to the harvesting pipeline and be invariably available.
- **Recoverable**: When a harvesting task stops, either manually or due to a failure, the system should be able to recover and resume the task without intervening manually.

2.1 The CORE HARvesting System (CHARS)

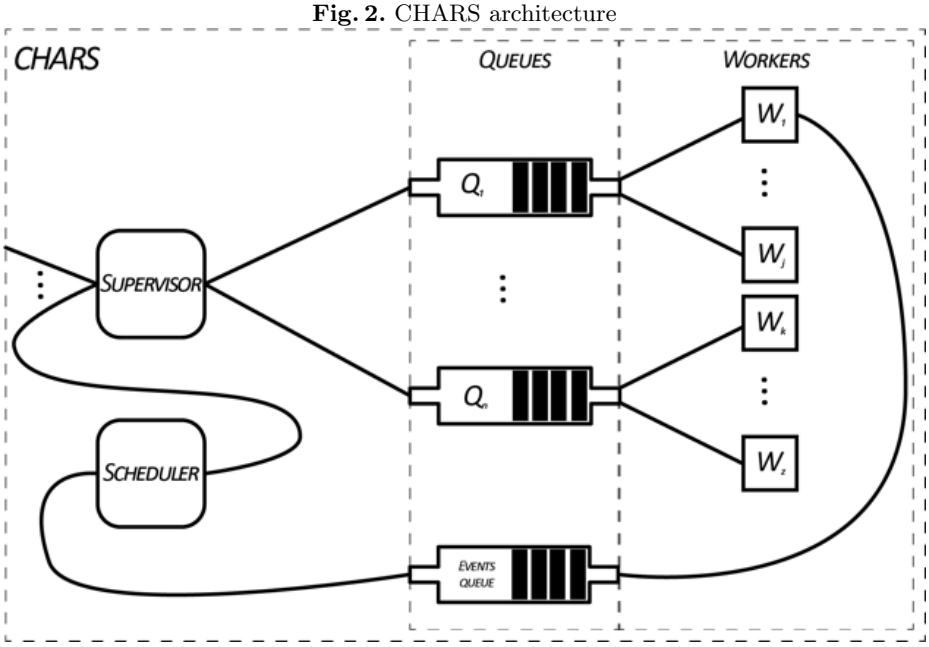
The design of the architecture is based on the following main components (Figure 2):

1. Worker (W_i): an independent and standalone application capable to execute a specific task and communicate via a queue;
2. Task Coordinator (scheduler): it becomes active when a task starts or finishes and coordinates the task workflow in the system;
3. Queue Handling System (Q_n): a messaging system that assists with the communication between the components;
4. Cron Scheduler: lines up tasks periodically;
5. Harvesting endpoint (supervisor): an API endpoint that facilitates the submission of task in the system and therefore works as an entry point for new harvesting requests.

All Workers (1) share the same lifecycle after receiving a task (Table 1). The lifecycle has been designed to be simple, generic and easy to troubleshoot.

The Event Scheduler (2), decides how the tasks should be moving and chooses the next task to run in the ingestion pipeline. If there are enough resources available in the system, i.e. idle Workers, the Scheduler adds new harvesting tasks based on the following policy: first, is adds repositories in the queue with metadata records older than a certain time window and, second, includes new repositories if no other priorities are being met. With this policy, our goal is to efficiently use all the available resources without overloading our system. Since

Fig. 2. CHARS architecture



Item	Description
0	Notify start
1	Collect data to perform task
2	Perform task
3	Finalise execution and collect metrics
4	Assess results (success or failure)
5	Notify end

Table 1. Worker lifecycle

our aim is to keep the content in our collection as fresh as possible, we periodically re-harvest repositories and add new ones without them interfering with the existing procedures.

The communication infrastructure is accomplished via a publish/subscribe pattern [22]. The Workers with the ability to perform a metadata download task express interest in the concept of metadata download and, when a producer submits a message, a Worker will be notified and act on it. A message is an item that is transmitted between components, such as a task definition or a start/finish event. When a CORE staff member wishes to harvest a repository, s/he needs to send a message to the harvesting endpoint by clicking a button in the administrator console. The task will then be created and submitted to the queue system.

2.2 Implementation of CHARS

In deciding the software and tools to use in the implementation of CHARS, we investigated software that fulfilled the following two requirements; out-of-the box solution and ease of integration with our system.

We implemented CHARS following an iterative approach; in the first iteration, we introduced the new architecture and attempted to limit the changes to our existing codebase. During the second iteration, we focused on modifying the harvesting pipeline and moved some tasks, such as information extraction, enrichment and indexing, from a repository level to an article level. With this change we managed to increase the depth of parallelism in our system. For the third iteration, we looked into detail at the mechanics of each task and investigated possible inefficiencies.

The backbone of our infrastructure is written in Java [23] and the Spring Framework [24]; nonetheless, the system equally supports a variety of different technologies for implementing workers. The persistent state of the system was already implemented using MySQL [25] and ElasticSearch and has been migrated as it is from the old infrastructure, since it was already meeting our requirements.

The infrastructure is built and managed through the SupervisorD software [26], which is able to restart the workers when they fail and allows us to define event based actions. SupervisorD uses a static configuration per server, meaning that we are able to define the amount of workers running on a single server based on its hardware specifications. As an alternative to SupervisorD, we explored the possibility of adopting container-based microservices using Docker and Kubernetes [27]. A container-based approach would allow a reusable microservices implementation with a dynamic and fine-grained management of the workers based on the available resources. One disadvantage, though, is that their configuration and usage would have increased the level of complexity of our infrastructure. In the end, we decided to postpone this approach and adopt a container-based microservices approach in a future iteration of the CHARS infrastructure.

2.3 Queuing system

For the queue message system, we explored two solutions, RabbitMQ [8] and Kafka [28]. Even though the latter has superior performance in terms of message rate[29], nonetheless it offers less out-of-the-box functionalities. Although Kafka performed better in the message rate, our use case did not need a system that could process an extremely high number, for example millions, of messages per second. Therefore, we decided to adopt RabbitMQ, which was easier than Kafka to configure and could be integrated with our existing infrastructure and code base. An additional functionality of RabbitMQ was the out-of-the-box support for message priorities. Prioritising in harvesting is necessary, since we often receive requests from our data providers to re-harvest repository collections in order to capture record updates, metadata and full text. Similarly, while fixing and resolving technical harvesting issues, we need to re-harvest a repository more than once, and we need to skip the existing queue.

Year	Metadata	Full text
2015 Apr	23,006,000	2,091,334
2016 (Jan - Dec)	66,137,655	4,626,215
2017 (Jan - Mar)	68,387,703	5,852,274

Table 2. CORE's metadata and full text volume

3 Discussion

CORE has been using the new microservices architecture for the past year. During that time, our collection doubled; we now have 70 million metadata records and 6 million full text PDFs (Table 2). Although we cannot correlate the content increase with the introduction of the new microservices infrastructure, nonetheless we realised that we achieved a reduction in the consumption of staff time with regards to maintenance tasks, which favored the uptake of new duties.

One of the lessons we have learnt is that, by dividing the monolithic code into small components with specific operations, we were able to maintain and troubleshoot our system easier and more efficiently comparing to the past. In addition, when an issue arises, programmers can focus on small problematic units of code instead of going through the whole monolithic code base. Another improvement in the system's efficiency relates to recoverability. CORE harvests some large repositories and a task can run for a long time, even days for large data sets. In our new infrastructure cases of failure or re-deployment of an improved version of the code are treated differently; the task is not lost and it automatically resumes without any manual intervention.

By moving the monolithic harvesting approach to a microservices application, we were able to focus on the quality and performance of each single task establishing valid measures, such as success or failure. For example, in our earlier infrastructure it was problematic to focus on issues, such as delays and quality assurance control, whereas, currently, we are in position to decide whether we should direct more effort and resources in specific tasks.

The introduction of microservices allowed us to work in a more scalable and distributed environment. Scaling up in the old infrastructure required the addition of new resources or a new server. With the new architecture we are able to shuffle our services efficiently within the existing hardware infrastructure and transform it according to our needs.

While introducing microservices in CORE, we also separated our harvesting infrastructure from our publicly available services. Right now our public services are connected to the harvesting back-end only through our ElasticSearch index. The advantages of this distinction, between our front-end and back-end services, resulted in an increase in the amount of uptime of our services.

Despite the aforementioned advantages, new challenges have emerged. The first relates to the difficulty of estimating the optimal number of workers in our system to efficiently run. While the worker allocation is still largely done using a trial and error approach, we are investigating more sophisticated approaches based on formal models of distributed computation, such as Petri Nets [30].

For example, we are looking into formally modeling our system to find valid heuristics for dynamically allocating or launching workers to optimise the usage of our resources.

The designed architecture has been built with an evolutionary approach slowly removing the dependencies from the monolithic system. This introduced a complexity in the production of a formal evaluation framework of our architecture, one that would not require plenty of time and effort. However, with building the aforementioned formal model we will be also able to validate the quality of our approach in an experimental way.

This architecture highlighted an issue of cost-effective resource allocation in our system. Our previous architecture was designed for a monolothic approach, where servers were allocated in full for the harvesting process. With the introduction microservices we were able to fine-grain resource allocation and implement different ways of collecting hardware resources, such as using cloud services for storage and computational power.

With regards to CORE's internal infrastructure, we need to improve the overall performance of our system and, thus, we are now collecting metrics; CPU, memory and network usage, freshness of content and quality of full-text crawling. These metrics will help us define new performance key indicators and improve our services. Moreover, we are exploring ways of optimising the use of the resources in an highly efficient way. Even though, our new infrastructure enabled us to scale up faster than expected, nonetheless we are now facing other issues, such as error detection in the harvesting process, full text crawling efficiency and deduplication improvements. All these are currently highlighted in our system and need to be addressed in future work.

4 Conclusion

In this paper we outlined the requirements of a scalable aggregation system and by following them we designed a microservices architecture that could be applied to any aggregation service or digital library. We presented how CORE's harvesting process migrated from a monolithic to a microservices approach, explained the harvesting workflow, and the technology used in the real life implementation of the architecture. Finally, we discussed the advantages and disadvantages of our new infrastructure and presented future work.

References

1. Knoth, P., Anastasiou, L., Pearce, S.: My repository is being aggregated: a blessing or a curse? Open Repositories 2014.
2. Newman, S.: Building microservices: Designing fine-grained systems. O'Reilly Media, Inc. (2015).
3. Manghi, P., Artini, M., Atzori, C., Pegano, P., Bardi, A., Mannocci, A., La Bruzzo, S., Candela, L., Castelli, D., Pagano, P.: The D-NET Software toolkit:

A framework for the realization, maintenance, and operation of aggregative infrastructures. Program Electronic library and Information Systems, 48, 4. (2014). doi:10.1108/PROG=08-2013-0045
4. OpenAIRE Guidelines. https://guidelines.openaire.eu/en/latest/
5. Open Archives Initiative Protocol for Metadata Harvesting. https://www.openarchives.org/pmh/
6. Joint, N.: Current Research Information Systems, open access repositories and libraries: ANTANEUS. Library Review, 57, 8, (2008). doi: http://dx.doi.org/10.1108/00242530810899559
7. SHARE 2.0 Documentation http://share-research.readthedocs.io/en/latest/
8. Rabbit MQ. https://www.rabbitmq.com/
9. Celery period tasks. http://docs.celeryproject.org/en/latest/userguide/periodic-tasks.html
10. Elastic. https://www.elastic.co/
11. Lösch, M.: A Multidisciplinary Search Engine for Scientific Open Access Documents. In: Depping, R. and Christiane, S. (eds.) Elektronische Schriftenreihe der Universitäts- und Stadtbibliothek Köln. 2, p. 11-15. (2011).
12. Indexing and basic data operations https://cwiki.apache.org/confluence/display/solr/Introduction+to+Solr+Indexing
13. Knoth, P., Zdrahal, Z.: CORE: Three access levels to underpin Open Access. D-Lib Magazine, 18, 11–12 (2012)
14. Li, H., Councill, I., Bolelli, L., Zhou, D., Song, Y., Lee W-G, Sivasubramaniam, A., Giles, L.: CiteSeerX: A scalable autonomous scientific digital library. INFOSCALE'06: Proceedings of the First International Conference on Scalable Information Systems. May 29- June 1. (2016)
15. Suber, P.: Open Access. MIT Press Essential Knowledge Series (2012)
16. Dublin CORE Metadata Initiative. http://dublincore.org/
17. Metadata Encoding and Transmission Standard. http://www.loc.gov/standards/mets/
18. The RIOXX Metadata Profile and Guidelines. http://rioxx.net/
19. Doberkat, E.: Pipelines: Modelling software architecture through relations. Acta Informatica, 40, 1, (2003) doi:10.1007/s00236-003-0121-z
20. Abrams, S., Cruse, P., Kunze, J., Minor, D.: Curation Micro-services: A pipeline metaphor for repositories. Journal of Digital Information, 12, 2, (2011).
21. Crossref: Metadata Enables Connections. https://www.crossref.org/
22. Birman, K., Joseph, T.: Exploiting visual synchrony in distributed systems. SIGOPS Operating Systems Principles. 123-138, (1987). doi: 10.1145/41457.37515
23. Java Software https://www.oracle.com/java/index.html
24. Spring Framework https://spring.io/
25. MySQL https://www.mysql.com/
26. Supervisor: A process control system. http://supervisord.org/
27. Kubernetes. https://kubernetes.io/
28. Apache Kafka. https://kafka.apache.org/
29. Kreps, J., Narkhede, N., Rao, J.: Kafka: A distributed messaging system for log processing. NetDB:6th Workshop on Networking meetsDatabases, (2011).
30. Petri, C..: Communication With Automata: Volume 1 Supplement 1. DTIC Research Report AD0630125, (1966).