NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DIOGO RAMOS PAULICO

BSc in Computer Science

METRICS AND REAL-TIME MONITORING IN DISTRIBUTED SYSTEMS

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2023

DEPARTMENT OF
COMPUTER SCIENCE

METRICS AND REAL-TIME MONITORING IN DISTRIBUTED SYSTEMS

DIOGO RAMOS PAULICO
BSc in Computer Science

Adviser: João Carlos Antunes Leitão
Assistant Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2023

Abstract

With the increase in the availability of useful Internet services and the affordability of devices that allow access to the Internet, services have had to adopt a distributed model, to be able to provide service to a large number of geographically distributed users. This has led to an increasing need for tools which make the tasks of prototyping, implementing and evaluating distributed applications and protocols more manageable. While there are many tools which facilitate the implementation and evaluation of distributed applications and protocols, such as Babel and Kollaps, there is a lack of solutions for monitoring and assessing the performance of distributed systems. To achieve this, the developers of those applications must implement their own solutions for collecting metrics.

This work plans to address this shortcoming by developing a solution that allows developers of distributed applications and protocols to obtain information about the hardware resource utilization, generic metrics for protocols and applications, and user-defined metrics for their developed solutions, without making large modifications to their code. To this end, our solution will supply developers with a library which they can use to capture and expose relevant metrics for their implemented distributed applications and protocols. Moreover, this solution will provide a monitor that can collect those metrics and display them to the developer, using relevant visualizations.

Keywords: distributed systems and protocols, monitoring, metrics, performance assessment

Resumo

Com o aumento da disponibilidade dos serviços da Internet e com a accessibilidade a dispositivos com acesso à Internet, os serviços tiveram de adotar modelos distribuídos, de modo a conseguir servir um grande número de utilizadores geograficamente distribuídos. Isto levou ao aumento da necessidade de ter ferramentas que simplifiquem as tarefas de implementação e avaliação de aplicações e protocolos distribuídos. Apesar de existirem várias ferramentas que facilitam a implementação e avaliação de aplicações e protocolos distribuídos, como o Babel e o Kollaps, há escassez de soluções para monitorização e avaliação do desempenho de sistemas distribuídos. Para alcançar isto, os desenvolvedores devem implementar a suas próprias soluções para coleção de métricas.

Este projeto planeia resolver esta falha, desenvolvendo uma solução quer permita aos desenvolvedores de aplicações e protocolos distribuídos obter informação sobre os recursos de hardware utilizados, métricas genéricas sobre protocolos e aplicações e métricas definidas pelo utilizador para a sua solução, sem ter de fazer alterações significativas no seu código. Para este fim, a nossa solução providenciará aos desenvolvedores uma biblioteca que estes podem utilizar para recolher e expôr métricas relevantes para as aplicações e protocolos distribuídos que implementaram. Além disso, a presente solução disponibilizará uma componente capaz de coletar métricas e apresentá-las ao desenvolvedor, por meio de visualizações relevantes.

Palavras-chave: sistemas e protocolos distribuídos, monitorização, métricas, avaliação de desempenho

Contents

List of Figures

1 Introduction

The increase in the number of Internet users [32] is explained not only by the fact that there are several services useful to everyday life readily accessible, such as those related to communication, navigation, entertainment, and education, but also because computer systems have gotten more powerful over time whilst getting smaller and more affordable [31]. In addition, technological improvements made in networking allow for easier access to the Internet.

Due to this increase in usefulness and accessibility, Internet users are now geographically dispersed, thus Internet services can no longer afford to not be distributed since for this they would need a very large amount of computing power to be able to provide service to all users while still providing poor service to users distant from the location where the service is hosted due to latency increasing with distance [33].

This means there is an ever-increasing need for easing the process of prototyping, developing and evaluating new distributed protocols or applications, that can operate on a large scale setting and provide adequate performance, to provide adequate service to a large and spread out user base.

Babel [15, 16] is a framework developed at NOVA LINCS, that provides valuable support in the task of prototyping and developing new distributed protocols by allowing developers to focus on the logic of their solutions, instead of expending time on the repetitive and error-prone low level aspects of such development.

Nonetheless, this framework does not feature a metrics collection system, that could allow developers of distributed protocols and applications to evaluate their complete solutions and even test the impact of small changes based on the observations of some generic hardware metrics and application/prototype-specific metrics. Thus relying on each developer to put effort into developing their own systems to export and collect relevant metrics, which not only costs precious development time, it also opens the door for bugs to be introduced, that may result in inaccurate evaluation results, which may go unnoticed causing an under-performing or incorrect solution to be deployed.

1.1 Objective

This work intends to address the aforementioned shortcoming by exploring relevant metrics for each type of distributed protocol but also generic metrics, such as those related to computational resource usage, in order to supply developers with these in a streamlined way allowing them to have a more complete view of the inner workings of their developed stack of distributed protocols and/or distributed system.

One of the challenges this presents is how to collect the protocol-specific metrics, while asking the developer to make minimal modifications to their code and still allowing the collection of as many relevant metrics as possible, without noticeable overhead being introduced during the testing stage.

Another challenge relates to performance, in order to not take away resources from the running protocol, the retrieval of metrics should incur in as little overhead as possible while still providing enough insight into the functioning of the application so as to be useful for the developer.

While the solutions to be pursued in the context of this work should be generic, in this work, we will apply them to the Babel framework.

1.2 Expected Contributions

The main contributions that are expected to result from conduction of the work discussed in this document are the following:

1. The implementation of a metrics system that allows protocol developers to monitor general hardware, protocol specific, and arbitrary metrics related to their protocols or applications by the means of simplified, non-intrusive and performant interface for data collection.

2. The development of a monitor that collects these metrics in real-time and presents them to the developer in a readable form by using, for example, suitable dashboard or plots, while also enabling this data to be exported.

3. Integration of the two previous contributions into the Babel framework, resulting in a new release of the framework.

4. Experimental assessment of the proposed solution in the context of Babel by exploring two different use cases.

1.3 Document Structure

The remainder of this document is organized as follows:

Chapter 2 Presents relevant and related concepts for the development of this work. Namely, it offers a small introduction on distributed systems, explores the models for distribution of these systems, and presents some protocols and services that serve as building blocks for distributed systems. We then present metrics that are made available by the runtime environments of distributed applications, some of the more relevant tools used for collecting metrics and monitoring computer systems and finally, some systems used for designing and testing distributed systems.

Chapter 3 Presents the future work to be carried out for this project. The strategy to fulfil the proposed objectives and a schedule to achieve the purposed objectives.

2 Related Work

In this chapter, we provide an introduction into concepts relevant for the elaboration of this dissertation. Section 2.1 provides an introduction into fundamental concepts of Distributed Systems, looking more in depth into the different models for distribution and for each of them provide examples of systems that apply that distribution model and relevant metrics. Section 2.2 presents environments where these applications are deployed and the hardware utilization metrics that each of these environments exposes. Section 2.3 introduces some systems used for collecting metrics and monitoring systems, for each, we present their capabilities, architecture and discuss how they can be leveraged in this project. Section 2.4 presents existing systems for assisting in the task of developing and evaluating distributed applications/protocols, for each, introducing their features, architecture, but also their shortcomings.

2.1 Distributed Systems and Performance Assessment

Distributed systems are systems that are spread across machines, connected by a network, allowing for greater concurrency, fault-tolerance and potentially reduced latency for users as some of the machines that make up the system may be placed closer to them.

2.1.1 Distribution models

These systems follow one of three alternative models for distribution: Client-Server model presented in Section 2.1.1.1, Grid/Cloud model presented in Section 2.1.1.2, and Decentralized model presented in Section 2.1.1.3.

In the following sections, we introduce each of these distributions models in turn, and provide examples of their application in practice alongside relevant metrics and performance indicators that can be used to characterize them.

2.1.1.1 Client - Server Model

This is the simplest model for distributed systems, in this model the processes belong to one of two groups, clients, those who request a service/or resource, and servers, those who provide a service or resources.

This simple distribution model is used in web-based systems that serve static web pages. The server is implemented by a process, that has access to a file system that stores the HTML files, which answers the client's request with the desired file. The client requests these documents through a web browser, which is not only responsible for requesting the documents using a protocol common to both the client and server, which in this case is either HTTP or HTTPS, but also responsible for displaying the content of the HTML document correctly to the user.

Some relevant metrics for distributed applications that follow this distribution model are:

Average Response Time This metric refers to how much time it takes for the server to reply to a request once it has received it. A higher average response time than expected may imply that this server instance is trying to handle more requests than it should be, which may result in a degraded experience for the clients.

Number of active threads Modern servers have a queue, that hold incoming requests while they are waiting to be processed, and a pool of threads for satisfying these requests. Thus, the number of active threads provides an insight into the load our server instance is under, if the number of active threads is very close to the number of total threads in the pool, requests may be left in the queue longer resulting in a longer response time for clients.

Number of requests The number of requests received by a certain server instance may be compared with other instances to evaluate if the load is being divided by all instances as expected.

Availability The availability of a server can be measured by the success rate of requests.

Generic hardware utilization metrics Hardware resource usage metrics, such as CPU and memory usage, may allow us to identify a fault in a certain server instance through a historic difference in resource usage, or a difference between the resource usage of other server instances.

2.1.1.2 Grid/Cloud Model

Whereas in Cluster computing we have a system that comprises mostly similar computers, all connected by the same network that is used to run computation intensive applications taking advantage of parallel execution across the different computers, Grid computing is concerned with solving the Grid problem, defined by [11] as the problem of "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations".

To this effect, we have machines from different organizations, which may have different characteristics, that are brought together through the creation of virtual organizations,

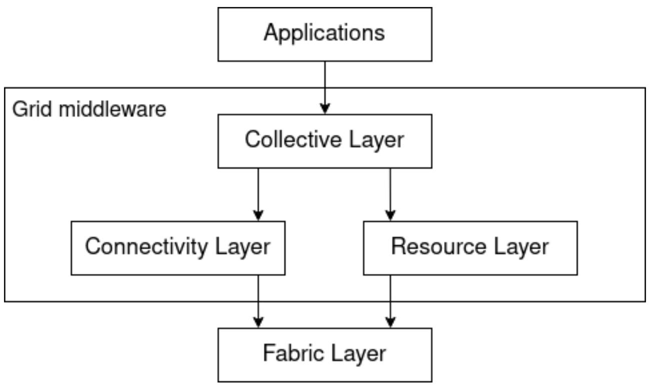

Figure 2.1: Layered grid architecture

comprised of many real organizations, whose members share the computational resources encompassed by the formed virtual organization.

While there is no standard architecture for Grid computing platforms, these are usually composed of the following layers:

Applications Layer Comprised of the applications that take advantage of the computation resources offered by the virtual organization.

Collective Layer This layer discloses the services offered by the virtual organization. To this effect, it features services to allocate, schedule, and discover available computational resources.

Connectivity Layer Features the communication protocols needed to support the operation of the grid infrastructure. These are necessary to provide a way for data to be transferred between the machines that comprise the system, for accessing resources in a remote location (in another machine that is part of the virtual organization platform) and, other services such as authentication.

Fabric Layer Provides interfaces to access specific resources belonging to the system. These interfaces are designed to facilitate the sharing of resources, which in the case of Grid computing are computational resources, providing ways to query the state, get the capabilities, and locking of these resources.

Resource Layer Responsible for managing resources individually, takes advantage of the functions offered by the connectivity layer and calls functions from the fabric layer. It provides functions to obtain information or interact with a specific computational resource.

The aforementioned collective, connectivity, and resource layer form what is known as the Grid middleware, which gives access to and manages resources that are part of the virtual organization.

An overview of this architecture is presented in Figure 2.1.

Cloud computing originated from the desire of organizations, such as Amazon, which at the time was only an online retailer, running large pools of computational resources, such as data centers, to monetize the computing capacity which was not actively needed, but existed to support moments of high system utilization, such as periods of sales in the case of online retailers. Moreover, this also allowed smaller organizations to take advantage of powerful computation infrastructures without the need to hire qualified personnel to manage them. This desire resulted in the precursor of cloud computing, utility computing, in which customers paid for the resources necessary to run the tasks they needed on these organizations' infrastructure.

Nowadays, Cloud computing is characterized by an available and easy to use pool of resources, that can be dynamically and easily scaled and configured depending on the customer's needs at that moment.

These cloud computing infrastructures are organized into four layers:

Hardware Comprised of the physical components of the systems, these are placed in data centers and the customer never interacts with these directly.

Infrastructure This layer forms the foundation of cloud platforms. Virtualization is used to provide customers virtualized resources, such as storage and computing capacity.

Platform The platform layer provides users a base where they can deploy their applications. Similarly to an operating system's libraries, these platforms also offer several service abstractions to interact with the offered resources, such as storage.

Application Applications executed on top of their own infrastructure are made available to the customer by the cloud vendor are offered in this layer.

Cloud vendors expose these layers as services to users. The Infrastructure layer is offered as Infrastructure-as-a-Service, this service allows clients to get access to virtual machines with different combinations of computational and storage resources. The Platform layer is offered as Platform-as-a-Service, giving users the ability to deploy their own application without having to worry about setting up an operating system and libraries on a provided virtual machine. The Application layer is offered as Software-as-a-Service, where users can get access to the applications developed by cloud vendors which are deployed in their own infrastructure.

Some of the relevant metrics for cloud deployed distributed applications are common to those presented in the client-server model, such as average response time, number of requests received, availability, and hardware utilization metrics as these are generally relevant for all distributed applications, as they allow us to identify any performance issues, faults, or poor load distribution that may be present in the system.

In regard to these hardware utilization metrics, cloud providers offer ways to obtain these metrics, in Amazon's case this is Amazon Cloudwatch [3], such as CPU, disk I/O, network I/O and memory usage, without the need for installing any software on the Virtual Machines where they deploy their applications.

2.1.1.3 Decentralized Model

This paradigm has become increasingly interesting as the computational capacity and number of devices continues to increase. Besides the increase in usage of computation devices, there has also been an exceptional increase in the number of people connected to the Internet, according to Our World in Data [32], as of 2020, around 60% of the word population had access to the Internet. Decentralized applications allow us to take advantage of the resources offered by these devices already connected to a common network, to improve quality of service and ensure scalability of popular distributed systems.

Peer-to-peer or P2P computing allows computers participating in the network, normally referred to as nodes, to exchange information or provide services directly among themselves, ideally bypassing the need for a centralized (even if logically) server.

While P2P systems should ideally contain no centralized components, this is not always the case. Napster [8], a P2P file-sharing application, employed an approach where a centralized component operated as a central directory, which had information about what files were made available by which users and this directory was updated every time nodes joined/left the system. Another example, is BOINC [2], whose objective is to make it easy for scientists to take advantage of computational resources made available by the public to perform super computations of scientific interest. Scientific projects that wish to take advantage of these public computational resources make available a central server which contains an application and the work units, which specify the inputs to a computation, that peers will process. After each work unit is processed, the peers send the result to this centralized server.

Typical applications where P2P technologies do, or could, bring advantages are presented in the book *Peer-to-Peer Computing - Principles and Applications* [36] and include, but are not limited to:

Scientific computation The massive availability of computation devices, means a large fraction of them are not being actively used at all times, this means there is a huge amount of computing capacity that is connected to the Internet left unused. Projects, such as the aforementioned BOINC project, allow common people to share computing capacity they do not need with scientific researchers studying, for example, protein dynamics to develop new therapies for various diseases, among other initiatives that require high amounts of computing power. This may allow small research teams, which may work with limited funds, to perform complex and complicated computations without access to large centralized computing clusters.

Collaborative file systems Computers that are part of a P2P system share their disk space, to allow for content to be accesses and replicated in different parts of the network, thus allowing for content to potentially be placed closer to users that need it, reducing access times. An example of this is the InterPlanetary File System, or IPFS [6], a P2P distributed file system where nodes get files from other peers and after getting them they also help distribute them

Distributed Databases Systems like CassandraDB [12] allow organizations with very large amounts of data by scaling their database horizontally (which involves adding new machines rather than by adding more resources to an existing machine), avoiding the need for a large and expensive central database system.

One of the fundamental operations in P2P systems is message routing. It is through the passing of messages between peers that nodes in a network are able to locate the desired resources, thus making the design of these routing protocols the target of various research efforts in the area of P2P systems.

Some metrics that allow us to evaluate different routing protocols are:

Storage Typically in each node of a P2P network some central information, such as data regarding other peers, is kept by peers to enable and accelerate the process of searching for content. While keeping no information means we may need to flood the network, causing a large amount of traffic in large networks, to find the resources we want, storing information about their peers also means that we need to update it, which may cause problems in networks with a lot of churn[1]. Thus, the amount of metadata that each peer has to keep about the network is directly related to how costly it is to keep that data up-to-date

Efficiency The efficiency of a routing protocol is defined by how quickly it can find the requested resource. This is measured by the average query path length, which is the average number of hops it takes for a query to reach the peer which has the requested resource.

Recall This metric refers to whether the search for a certain resource is successful. The higher the coverage, the more useful the system is.

Scalability This is important if we want the system to be functional in large-scale applications. The number of messages that need to be routed to locate resources is an example of a relevant metric to evaluate scalability. A broadcast based system will have a huge amount of messages being routed to locate resources, causing large amounts of network traffic, compromising the scalability of the system.

[1]changes in membership caused by peers joining and leaving the network.

Due to their growing relevance and usefulness, more and more protocols that rely on P2P architectures are being developed, and thus, the importance to obtain metrics for evaluation of these applications in a streamlined way is also growing.

2.1.2 Building blocks for distributed systems

Distributed applications are complex applications that need communication, coordination, agreement between their processes, in order to provide the desired functionality.

Section 2.1.2.1 explores coordination services, namely Zookeeper, a distributed coordination service. Section 2.1.2.2 explores communication services, focusing on Kafka, a distributed messaging system. Section 2.1.2.3 explores membership protocols, presenting the example of the HyParView [25] protocol. Section 2.1.2.4 explores agreement protocols, presenting the example of the Paxos [24] protocol.

2.1.2.1 Coordination services

While processes of distributed applications operate independently, they need to coordinate with each other in order to provide the desired functionality. Examples of relevant coordination include: determining the leader process in a replicated system and ensuring that all components of a system have access to the same global configuration parameters.

Zookeeper [21] is an example of a service that allows distributed applications to execute different coordination tasks. To this effect, it exposes a simple API that supplies its clients an abstraction in the form of a set of hierarchical data nodes (znodes) which are replicated across all Zookeeper servers that compose the service. It allows znodes to be dependent on active TCP connections (and keep alive messages) to the node that created them. Furthermore, it provides a notification mechanism, that allows clients to be notified when there are changes in the znodes. It is commonly used to allow developers to implement their own coordination mechanisms, such as, the aforementioned leader election and configuration management.

Zookeeper's evaluation focused on the following metrics:

Throughput of reads and writes This metric measures the number of reads and writes per second that the system can handle.

Latency This metric measures the time it takes for an operation, such as creating a znode, to the Zookeeper service to be completed.

2.1.2.2 Communication Services

Kafka [23] is a distributed messaging system designed to handle high volumes of data with low latency. Kafka's architecture consists of *topics*, which group messages; *brokers*, the servers that store topics and the messages contained in them; *consumers* which subscribe to topics and receive messages from them, and *producers* that publish messages to topics.

Kafka scales horizontally, meaning brokers can be added to the system dynamically allowing for load balancing, and forming a cluster. Topics are split into partitions, which are distributed across brokers.

The metrics taken into account when evaluating Kafka's performance were the consumer and producer throughout, meaning the number of messages consumed and produced per second.

2.1.2.3 Membership Protocols

For distributed systems to work, they need to communicate and for that to happen they need to know which nodes are part of the system. This is done by using membership protocols, which are responsible for keeping track of the nodes belonging to the system.

The initial assumption is that all nodes in the system know all the other nodes in the system, meaning they would have a global view of the system. However, this assumption presents problems, specially in large scale systems, namely as the number of nodes in the systems changes, with node joins and departures, the membership information needs to be updated which is a costly operation, due to it requiring messages to be delivered to every node whenever the membership changes, moreover information about membership can be heavy in large systems. This initial assumption is thus relaxed, now nodes only have a partial view of the system, knowing only a subset of participants, to this effect, nodes associate with their neighboring nodes, forming an overlay network, a logical network deployed on top of another network, whose links are logical, meaning two direct neighbors in an overlay may be separated by multiple hoops in the underlying network.

HyParView [25] is a membership protocol, based on partial views, which relies on two distinct partial views, an active one and a passive one. The active view of each node contains a small set of nodes to which they will disseminate broadcast messages. The passive view of each node, provides fault-tolerance and contains a larger set of nodes, that are useful to replace nodes in the active view in case of failures of nodes.

Some metrics that allow us to evaluate different membership protocols are:

Accuracy The accuracy of a node is defined as the number of neighbor nodes that have failed divided by the total number of neighbor nodes.

Connectivity The connectivity of the overlay network, dictates whether all nodes have a way of receiving messages. If an overlay is not connected, some nodes are isolated, thus unable to participate in the system.

Graph Diameter Average number of hops to go from a node to every other node.

2.1.2.4 Agreement Protocols

For distributed systems to continue to work despite the failures of individual components, they need to have their state replicated across multiple nodes, to this effect nodes need to execute the same operations in the same order. A way to do this without a centralized component is to have all nodes agree on the order of operations. This leads us to the consensus problem, which is not solvable in real life systems, which are asynchronous and fault prone. Thus, we need to use agreement protocols (an agreement protocol relaxes properties of the consensus problem, so there is a way of realizing a weaker version of the consensus agreement).

The Paxos [24] consensus protocol, relaxes the termination property, stating it only happens if the system behaves synchronously, and its variants [14, 10] have been used as a way to achieve state machine replication, typically by allowing replicas to agree on the order of client operations to be executed.

Some of the metrics used to evaluate Paxos and its variants are:

Throughput The number of operations that can be executed per second.

Average time for reaching agreement The average time it takes for nodes to agree on the same value.

Number of messages needed to reach agreement The number of messages that need to be propagated to reach agreement.

2.1.3 Discussion

In this section we explored different distribution models, and for each identify relevant metrics, we introduced different types of distributed protocols, which serve as building blocks to distributed applications, and the metrics that are used to evaluate each of them. The main goal of this section was to identify common performance metrics and indicators that are of relevance for developers and operators when optimizing or studying the performance of these solutions.

We have identified three types of metrics: hardware resource utilization metrics, generic metrics common for a wide range of distributed applications, and metrics specific to each distributed protocol.

2.2 Metrics from execution environments

Different execution environments provide different ways to obtain information about the usage of hardware resources.

In Section 2.2.1, we introduce the Linux family of operating systems and the featured proc [27] pseudo-file system that allows obtaining information about the per-process usage of hardware resources. Section 2.2.2 provides an introduction to virtualization, focusing on virtual machines and how we can obtain hardware utilization metrics about running virtual machines. Section 2.2.3 introduces containers, focusing on Docker containers due to their widespread adoption [22], presenting an overview of how we can obtain hardware utilization metrics about running containers and an existing solution that allows exporting those same metrics, so they can be analyzed.

2.2.1 Operating System

While this section is titled Operating Systems, we will be focusing on the Linux family of operating systems, as these are the most used operating systems in server applications according to a 2017 report published by Red Hat [35].

The Linux family of operating systems includes a pseudo-file system, /proc [27], that offers information about running processes and the system overall in a hierarchical file structure form.

It contains more general information about the system, such as CPU information in /proc/cpuinfo, and also information about each running process in a subdirectory whose name is the process ID.

For each process, we can get information about their CPU and memory usage, bytes written/read from disk and network metrics (bytes and packages sent and received).

This allows us to extract some blackbox metrics from applications running on bare metal, independently of whether we can modify their source code.

2.2.2 Virtualization

Virtual machines, VMs for short, are a type of virtualization technology, that enables us to run multiple operating systems, known as guest operating systems, on the same (host) hardware, supplying to each operating system its own virtualized hardware resources, such as CPU, memory, disk and network.

The virtual machine monitor, responsible for creating and running virtual machines, can be realized in one of two ways, as a native virtual monitor, which is implemented on top of the hardware and manages the access of its virtual machines to the underlying hardware, or as a hosted virtual machine monitor, which sits on top of a host operating system and relies on it to manage the access to the underlying hardware.

Virtualization takes special importance in the context of clouds, as it allows for isolating applications and their related environments, which means failures in one application do not affect the other applications hosted on the same physical machine.

The virtual machine monitor, due to being in charge of running and monitoring all running virtual machines, allows us to get hardware utilization metrics from all our virtual machines through it without the need to modify them.

2.2.3 Containers

The rise of containerization technologies is explained by the fact that they allow for time and cost savings in the software deployment process, since there is no need to configure hardware and software to host a deployment. Containers include the application code and necessary dependencies while being isolated from each other, allowing multiple containers to run on the same hardware.

Container isolation on Linux relies on control groups (cgroups) [9], a feature that limits and isolates the resource usage of a group of processes. cgroups also expose metrics about the usage of some resources by each cgroup.

In Docker, since each container belongs to its own cgroup, we can collect CPU, memory and I/O metrics for the container from its associated cgroup. Moreover, each container has its own virtual Ethernet adapter from which we can obtain metrics, such as bytes and packets sent and received, by making use of the /proc pseudo-file system.

Google's cAdvisor [17], or Container Advisor, runs either as a Docker container or as a standalone component and takes advantage of the aforementioned proc and cgroups mechanism to collect resource usage and performance metrics for running containers. These metrics are exposed as REST API, meaning they can be retrieved by metrics collection systems, such as Prometheus [28] which we will discuss further ahead.

2.3 Metrics collection and monitoring systems

There are many existing solutions for monitoring and collecting metrics from systems which take different approaches for doing so.

We can have solutions that implement specific protocols for monitoring and management, such as the Simple Network Management Protocol [29] which we describe in section 2.3.1, solutions that allow us to collect some blackbox metrics to monitor systems whose code we can't instrument, such as NAGIOS [5] which we present in section 2.3.2, solutions which collect and aggregate logs generated by systems to monitor and obtain metrics about those systems, such as Elastic Stack [4] and Splunk [7], which we present in sections 2.3.3 and 2.3.4 respectively, and solutions that collect and aggregate metrics from applications where we can instrument their code to export metrics in a format compatible with these systems, such as Prometheus [28] and Graphite [20], introduced in sections 2.3.5 and 2.3.6 respectively.

2.3.1 SNMP

The Simple Network Management Protocol [29], SNMP for short, is a protocol that uses UDP for communication, designed to allow the remote management and monitoring of devices on IP networks.

Its architecture is composed of a collection of managed network devices, which are running management agents that perform network management functions as requested by the manager, which is the device requesting information and issuing commands to the agents. This communication is however not only done in a request/reply fashion, as managers can set "traps" on agents, which allow for the manager to be notified by the agent when a set of conditions is met. For each management agent, there is a collection of managed objects, which must include a set of common objects for all agents implementing this protocol called a Management Information Base specified by RFC1213 [30], that expose the behavior and capabilities of that managed device.

Despite its features, we can not instrument the SNMP protocol to obtain metrics for distributed protocols and applications since it would require us to export unnecessary information due to the need to include the objects specified in the aforementioned Management Information Base.

2.3.2 NAGIOS

NAGIOS [5] is a monitoring system that focuses on the availability of hosts and services. In terms of its architecture, NAGIOS is composed of a central server, which collects and aggregates the results of the checks performed by the means of plugins, these plugins are executed by the central server to perform checks on hosts and services.

To this effect there are a variety of plugins, which are external commands or scripts, that make NAGIOS modular, and allow it to perform probing of hosts for reachability, for example by pinging them, and of services, by testing individual services such as HTTP, SMTP, etc.

The result of these checks indicates if the check was successful or if it failed with an error, if this error is critical, a warning or unknown.

Host checks are only performed if none of the services associated with that host can be reached, otherwise, if a service can be reached the host must be functioning, making that check redundant.

While NAGIOS is suitable for basic monitoring of static systems, it can not be instrumented to obtain real-time metrics for distributed systems as it is designed to check the availability of hosts and services, rather than their performance.

2.3.3 Elastic Stack

Elastic Stack [4] is an open-source software solution, maintained by Elastic, designed for centralized logging and analysis of massive amounts of data whose architecture is composed of three main components: Logstash, Elasticsearch and Kibana.

The Logstash component is a tool that extracts and transforms data from sources such as log files, message queues or streaming platforms, and feeds this data into Elasticsearch.

The Elasticsearch component is a data store, search and analytics engine that takes advantage of Apache Lucene [13] (an open source search engine library). It can scale horizontally, meaning nodes can be added to the cluster allowing the workload to be shared, moreover it also exposes REST APIs allowing for other applications to access its stored and analyzed data.

The Kibana component offers a graphical interface for Elasticsearch users, by taking advantages of its exposed REST APIs allowing them to search and visualize data.

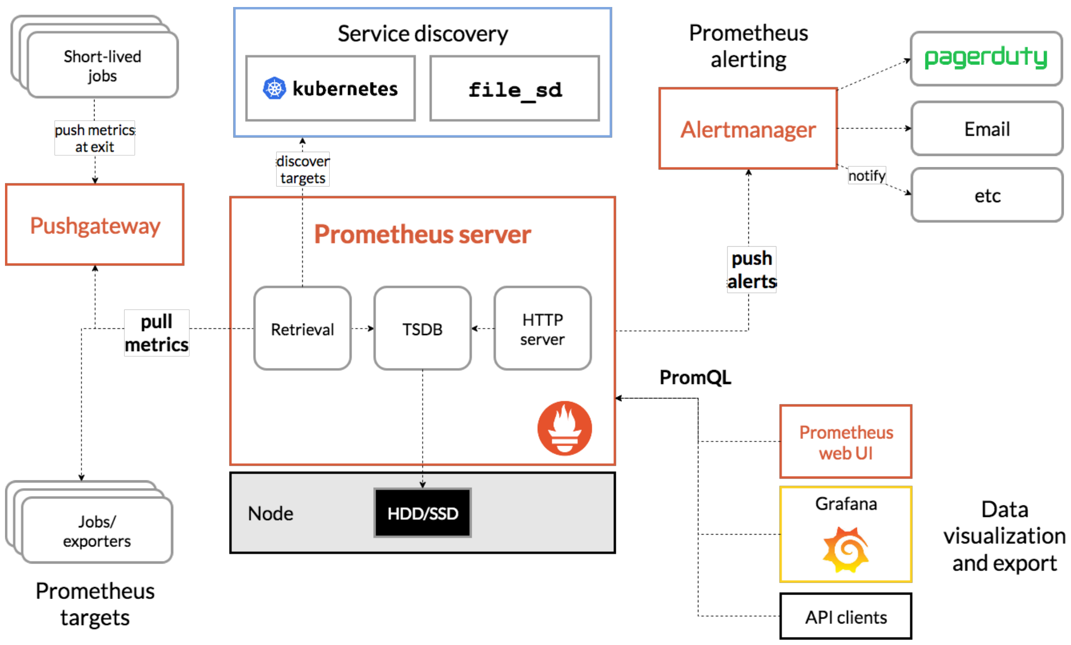

Figure 2.2: Prometheus's Architecture (extracted from [28])

Elastic Stack provides its users with a way to source, analyze and visualize vast amounts of data. This may prove useful to analyze logs that are already being used by protocol developers, we can take those logs and process them in order to provide a better and more concentrated view of the overall operation of their protocols.

2.3.4 Splunk

Splunk [7], similarly to Elastic Stack, is a platform designed to analyze large volumes of machine created data such as systems logs.

Its architecture comprises a forwarder, that collects these machine created information and forwards it to Indexers that will process and store this data, allowing system admins to get some insight into the systems they oversee.

It provides an alternative proprietary solution to Elastic's open-source Elastic Stack, which is not particularly useful for us as the need for licensing would mean it would not be practical to use.

2.3.5 Prometheus

Prometheus [28] is an open-source project, initially developed by SoundCloud, built for collecting and storing metrics for distributed systems. It features the ability to obtain metrics via a pull model, in which the systems from which we wish to obtain metrics, publish an HTTP endpoint which will be scraped by Prometheus. To this effect, Prometheus provides client libraries in various languages.

In short-lived applications, this pull model is replaced by a push model in which the applications from which we want to collect metrics push them to Prometheus's Pushgateway component, which functions as a cache for these metrics from which Prometheus will scrape them.

In addition to the client libraries mentioned before, Prometheus also supplies the developer with Exporters, which allow exporting metrics from supported apps as metrics that can be scraped by Prometheus automatically.

Prometheus stores this metric data along with the timestamp at which it was recorded. However, due to reliability reasons, it will use storage local to the system running Prometheus to store this data, meaning it is not designed for long-term archival of metrics data (unlike Graphite, which we discuss next).

It features its own query language, PromQL, which allows selecting and aggregating metrics. This ability to query stored metrics data paired with its HTTP API, allows for integration with other applications, such as those used to visualize data, a widely known example being Grafana [19].

An overview of this architecture is presented in Figure 2.2.

Prometheus may prove useful in this work due to its ability to collect metrics from various sources, aggregate, and store them, and it further can be leveraged in collecting metrics from Babel applications and, more generally from applications running in a containerized form, while allowing collected data to be queried.

2.3.6 Graphite

Similarly to Prometheus, Graphite [20] is a monitoring tool that allows its users to track metrics about their services/applications. To this effect, it provides a storage solution for this time-series data and offers a way for users to obtain visualizations about the stored data.

Considering its architecture, it consists of one or more instances of the carbon daemon, this component receives the metrics data pushed into it by the application being monitored and reports them into Whisper. Whisper is a database system that provides reliable storage for numeric data over time (such as metrics) which allows higher resolution data, meaning there are more data points per time unit, to be decayed into lower resolution data to allow for more efficient archival.

However, whereas Prometheus provides a solution that can collect metrics for distributed systems, Graphite relies on a push model, meaning the monitored applications need to push the metrics into Graphite, which Prometheus also supports.

This makes Prometheus a more suitable candidate for a metrics collection and storing system as it features both, a pull and push model, and while it doesn't feature Graphite's native ability to archive metrics data efficiently, this is unnecessary for the purpose of evaluating distributed protocols.

2.3.7 Discussion

While some of the solutions presented in this chapter do not provide us any help in developing our metrics collection system, some solutions presented in this chapter may prove useful, namely Prometheus, presented in Section 2.3.5, due to supplying a solution that offers client libraries in various languages, including Java, which enables us to expose our collected metrics in a format that can be scraped by Prometheus, allowing us to direct our attention to the development of a way for developers to add metrics to their applications without much hassle.

2.4 Systems for designing and testing distributed systems

Due to the importance of distributed systems in the development of the Internet, which has an ever-growing number of users, there is a need for tools that can help developers design and test their distributed systems.

In this section, we will discuss systems that are designed to help developers design and test distributed systems. Section 2.4.1 explores SPLAY, a system for designing and evaluating distributed systems. Section 2.4.2 explores Babel, a framework for simplifying the development of production-ready distributed protocols and applications. Section 2.4.3 introduces Kollaps, a network emulator designed for evaluating the effect of the variability of network conditions in large scale applications, making it useful for distributed applications evaluation.

2.4.1 SPLAY

SPLAY [26] is an integrated system that was designed with the purpose of making the development, deployment, and evaluation of complex distributed applications easier, enabling fast prototyping. To this effect it features its very own simple language, similar to pseudocode, a lightweight and sandboxed execution environment, making it suitable to be used not only on dedicated testbeds but also in production systems.

In terms of its architecture, SPLAY consists of a controller, responsible for receiving requests for deploying applications, from now on referred to as jobs, deploying and executing said jobs, and daemons, responsible for starting, stopping, and monitoring SPLAY applications.

The controller is implemented as a set of processes that may be distributed throughout multiple servers but is not decentralized as it relies on a central database to store data regarding running applications and hosts.

Each participating machine, or node, runs a daemon, this daemon restricts the usage of the machine resources by running SPLAY applications based on previous configuration and also receives requests to run jobs from the controller.

Therefore, in the SPLAY architecture, each node, as long as it has enough free resources, can run multiple applications and these are sandboxed from each other.

An important component for evaluation of SPLAY applications is the churn management component. It allows a developer to specify how many, or which percentage, of nodes should join/leave the network, allowing them to assess how their developed solution reacts to changes in membership.

While SPLAY's simple pseudocode like language and architecture allow for simpler prototyping and evaluation of distributed systems, it does not offer its developers a way to obtain real-time metrics regarding their developed protocols' operation.

2.4.2 Babel

Babel [15, 16] is a Java framework created to simplify the process of prototyping and implementing distributed applications and protocols.

It presents a real alternative to simulators as it allows developers to implement production-ready code whereas simulators require implementing the solution twice, with no guarantees that the real implementation will match the logic of the code implemented for the simulator.

Each Babel process can have multiple protocols running, as each protocol will run in a different thread. This framework allows developers to focus on the logic of the protocol by handling for the developer many of the low level complexities associated with the development of distributed protocols, these include interactions between protocols, message passing, handling of timers, and concurrency aspects within and across protocols.

To abstract the network layer, Babel provides *channels*, protocols can interact with these using simple primitives and some channels with different capabilities and guarantees are already offered out of the box, an example being the *TCPChannel* that allow establishing TCP connections to other processes. These channels can be shared by multiple protocols, and each protocol can use any number of channels.

To simplify the development of protocols and abstract the aforementioned low level aspects, Babel provides an event driven programming model, meaning developers can model their protocols as state machines whose state evolves by receiving events where each protocol has its event queue from which events are retrieved. These events can be timers, channel notifications (notifications about the network layer), network messages (messages from other processes) and intra-process events (interactions between protocols running in the same process). Developers only have to register callback functions to handle each type of event, Babel core component handles the delivery of events to each protocol's event queue.

In addition to this, the core component keeps track of timers set up by the different protocols, delivering events to their event queue when they are triggered and also handles communication between protocols, both in the same and different processes.

A more succinct overview of this architecture is illustrated by Figure 2.3

Despite all its advantages, Babel does not offer any way for developers to monitor in real-time the behavior of their protocols which could allow them to better understand the

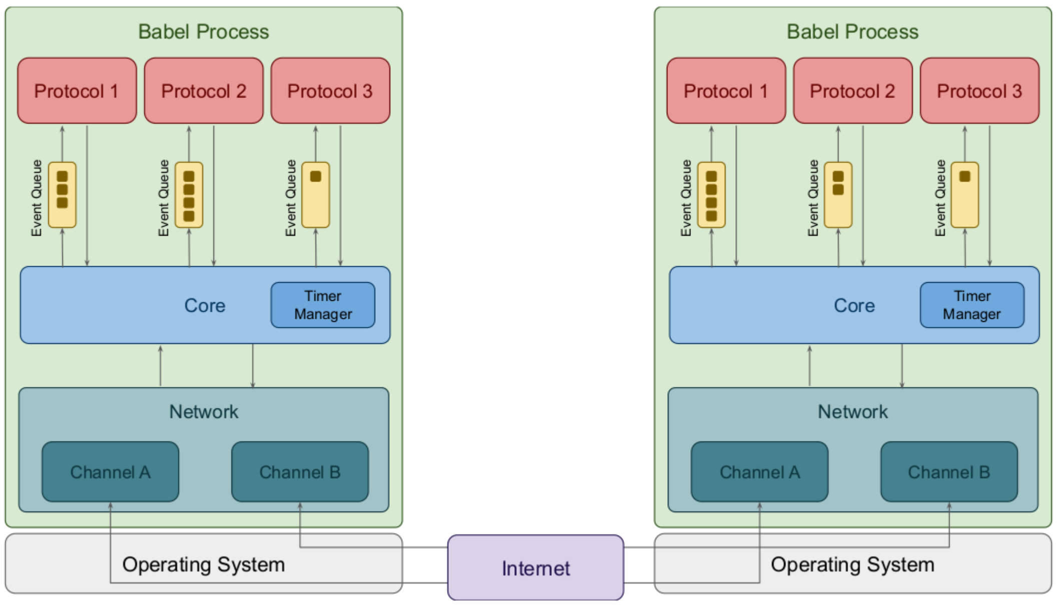

Figure 2.3: Babel's Architecture (extracted from [15])

inner workings of their solution, based on generic and protocol specific metrics, relying on the developer to implement a system to collect these metrics.

2.4.3 Kollaps

Kollaps [18] is a distributed network emulator designed to evaluate the effect of the variability in network properties in large-scale applications, which is agnostic to language, libraries, and transport protocol used as long as they can be containerized, without the need to modify its source code. It is also capable to scale to thousands of processes and is accurate when compared to bare metal approaches that keep the full state of the network.

It aims to fix the shortcomings of research testbeds, which do not always provide consistent conditions, due to overloading at certain times, and do not allow the modelling of dynamic network conditions, which is important for developers to understand the impact this change in conditions has in their application.

It also provides an alternative to simulators, which rely on a simulated model which may not correspond to the real-world implementation, meaning test results may not apply to our real-world deployment.

Kollaps takes a streamlined approach when compared to other network emulators. It does away with emulating the network elements and their internal state while maintaining accuracy. Since applications only care about end-to-end observable properties (such as jitter, bandwidth and packet loss), it collapses the network topology, calculating the properties of the end-to-end links, so they preserve the properties of the original topology. This simplified model has the advantage of allowing the emulation of dynamic network events, such as link removals, in a performant way Finally, to allow for easy scaling the emulation is maintained in a fully distributed fashion.

It has however some limitations. It does not feature the possibility of an interactive testing session, this is due to the dynamic events defined by the user to change the topology while the experiment progresses being computed before the experiment begins as a sequence of graphs as a way to avoid expensive computations while the experiment is running. In addition to this, since it uses a shortest path algorithm to generate the collapsed link between each pair of containers, it is not possible to use multipath routing. It is also not possible to use Kollaps to evaluate applications that run on bare-metal, it has a high CPU usage due to the communication system between its components and due to the retrieval of information from kernel in large-scale deployments, however these problems were solved in Kollaps 2.0 [1]

Kollaps provides a valuable contribution in the evaluation of distributed systems, as it allows developers to understand how the change in network conditions affects their applications. However, it does not provide any way of obtaining metrics about the system operation out of the box.

2.4.4 Discussion

As we saw in this chapter there is a myriad of tools available to developers to ease the process of developing and evaluating their distributed systems, however, none of them provide a way to obtain metrics about the system operation out of the box, relying on the developer to implement a system to collect these metrics. Among these tools Babel takes special importance as it allows developers to implement production-ready code, so it is important to provide a way for developers to obtain metrics that give them an insight into the inner workings of their distributed protocols and applications.

2.5 Summary

In this chapter, we presented the different models for distributed models and provided examples and relevant performance indicators for each of them, an overview of how different execution environments supply metrics about the hardware resource, and an overview of different systems for monitoring and obtaining metrics about the operation of systems.

Finally, we presented a survey of the state of the art in the field of distributed protocols' evaluation, exploring the tools available to developers to ease the process of prototyping, developing and, evaluating their distributed applications. We presented the main characteristics of each tool and discussed their advantages and shortcomings.

In the next chapter we will review our identified challenges, take the insights gained from this chapter and present an initial solution to the problem of supplying the obtaining metrics about the operation of distributed systems, we will also detail our evaluation methodology, present an idea of what our full solution may encompass, and present a roadmap for the future work.

3 Future Work

As introduced in Chapter 1 this work intends to address the lack of a system for collecting performance metrics and indicators for developers of distributed protocols and applications. In particular, in this work, we will focus on providing some support for the Babel framework. To this effect, this work intends to make the following contributions: the implementation of a metrics collection system which allows Babel developers to monitor hardware usage indicators, generic metrics for protocols and applications (e.g., request processing time, number of messages received) and user-defined metrics (e.g., a developer creating a Distributed Hash Table using a finger table approach to lookup keys, such as the one used in Chord [34], where each node of the system keeps a table of the identities of its known successors, may be interested in knowing the occupation of those same tables) related to the operation of their protocols through a simplified, non-intrusive and performant interface for data collection; the development of a monitor that collects this metrics in real-time and presents them to the developer in a readable format, by using a dashboard containing suitable plots, while also allowing it to be exported if the developer so desires and, the experimental evaluation of the solution in the context of Babel by exploring different use cases.

Section 3.1 presents an overview of the initial solution, including a short explanation on how the developer indicates the class of protocol they are developing, what metrics should be collected and how those metrics are then aggregated and displayed. Section 3.2 describes what aspects are relevant for the evaluation of our solution and goes into some detail about how each of these relevant aspects will be evaluated. Finally, Section 3.4 establishes the schedule to carry out the proposed work.

3.1 Initial Solution

The initial solution we plan on pursuing will focus on creating a library for capturing and exposing relevant metrics from distributed protocols and applications. While initially this library will be implemented in Java as this development stage will be focused in the Babel framework, this approach allows us to design a solution that can be used by developers using any programming language. On the other hand, since Java supports the use of annotations to inject code into the compiled classes, we will experiment with those in this initial solution, to see if they can be used to simplify the process of adding metrics to the code of the distributed protocols and applications.

To allow the initial solution to focus on the design and implementation of the mechanism for extracting metrics, we will take advantage of an already existing solution to collect these metrics so they can be displayed to the developer, by the means of Prometheus's Java client library, which allows us to expose metrics to be scraped by Prometheus.

These metrics fit into three types:

Counter A counter is a cumulative metric that only increases and can used to count the number of times an event has happened. For example, the number of messages received by a node.

Gauge Gauges are used to measure a value that can go up and down. For example, current memory usage.

Histogram Used to summarize continuous data measured on an interval of time. For example, how the average response time varies over time.

Prometheus will then store this metrics data, allows us to query it and export it so it can be used by Grafana to supply us with appropriate graphs and plots.

3.2 Evaluation

The suitability of our solution will depend directly to how well it can address the identified key challenges.

In terms of the challenge of incurring in as little overhead as possible, the evaluation consists of comparing the computational resource usage when the developed metrics collection system is disabled and when this system is enabled and collecting relevant metrics for different classes of distributed protocols, we plan to conduct experimentation focusing on consensus and membership protocols.

Regarding the challenge of the developer having to make the least amount of modifications to their code, the evaluation consists of comparing the lines of code that were added to the original developed protocol/application in order to allow the collection of relevant metrics, while also providing enough flexibility to specify how these metrics should be collected. To evaluate this we will conduct user testing of the developed solution by asking some developers with experience in using the Babel framework to evaluate the process of instrumenting the developed solution to collect metrics regarding their previously developed distributed protocols/applications.

3.3 Full Solution with optimization

The final solution will cement the method for extracting metrics, which may or may not be similar to the method presented in the initial solution, depending on the results of the performance and usability evaluation of that system.

Regarding the monitor, responsible for collecting and presenting these metrics to the developer using relevant visualizations, it will be designed to included in the Babel framework, avoiding the need for extra dependencies and large monitoring systems, which may include many unnecessary features, which may result in the introduction of extra overhead.

This full solution will be subjected to an optimization stage, where we will evaluate different models for collecting the metrics, namely comparing the overhead introduced when using a pull vs a push model.

In terms of the evaluation of this full solution, we will use as case studies the implementation of ChainPaxos [14], a high throughput state nachine replication solution that leverages a specialized variant of the Paxos algorithm [24] which organizes the nodes of the systems in a chain reducing the number of messages necessary to reach consensus. Due to lack of space, we omit the details of this solution, the interested reader can find a more detailed explanation of the ChainPaxos in [14].

Additionally, we will also rely on an implementation of the HyParView [25] membership protocol, a gossip based protocol for maintaining a view of the nodes present in a distributed system, both of which are implemented using the Babel framework and encompass different classes of distributed protocols, for each we will use our solution to collect relevant metrics, which we previously identified.

It will also be subjected to a stage of user testing, in which developers with some experience in using the Babel framework for developing distributed applications and protocols will assess the process of using our proposal to add relevant metrics to their previously developed protocols and applications.

After this evaluation process, this full solution will be integrated into Babel, resulting in a new release of the framework.

3.4 Planning

The expected schedule for this work is presented in Figure 3.1. It is split into the following main activities:

Initial Solution Involves the design, implementation, and evaluation of the initial solution of the metrics collection system for distributed protocols and applications.

Final Solution Consists in the design, implementation, evaluation, and optimization of the final solution for the system

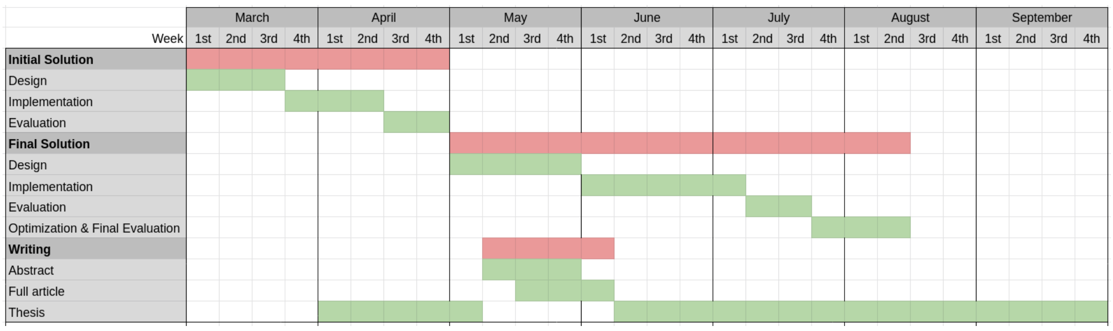

Figure 3.1: Gantt chart of expected tasks and duration

Article Consists in writing an article to be submitted to the INForum 2023 symposium.

Thesis Writing Consists in writing the thesis document which will be a long running activity over most of the time, to document the progress of the work.

Bibliography

[1] S. Amaro. MA thesis. 2021. URL: https://fenix.tecnico.ulisboa.pt/cursos/meic-t/dissertacao/1972678479055152 (cit. on p. 21).

[2] D. Anderson. "BOINC: a system for public-resource computing and storage". In: *Fifth IEEE/ACM International Workshop on Grid Computing*. 2004, pp. 4–10. DOI: 10.1109/GRID.2004.14 (cit. on p. 8).

[3] *Application and infrastructure monitoring - Amazon Cloudwatch*. https://aws.amazon.com/cloudwatch/. Accessed Jan 2023. 2023 (cit. on p. 7).

[4] A. Athick and S. Banon. *Getting Started with Elastic Stack 8.0*. Birmingham, England: Packt Publishing, 2022-03 (cit. on pp. 14, 15).

[5] W. Barth. *Nagios system and network monitoring*. eng. U.S. ed. Munich: Open Source Press, 2006. ISBN: 1-59327-124-7 (cit. on pp. 14, 15).

[6] J. Benet. *IPFS - Content Addressed, Versioned, P2P File System*. 2014. DOI: 10.48550/ARXIV.1407.3561. URL: https://arxiv.org/abs/1407.3561 (cit. on p. 9).

[7] D. Carasso. *Exploring Splunk*. Cito Research, 2012-08 (cit. on pp. 14, 16).

[8] B. Carlsson and R. Gustavsson. "The Rise and Fall of Napster - An Evolutionary Approach". In: *Active Media Technology*. 2001 (cit. on p. 8).

[9] *cgroups(7) Linux User's Manual*. 5.13. 2021-08. URL: https://man7.org/linux/man-pages/man7/cgroups.7.html (cit. on p. 14).

[10] H. Du and D. J. S. Hilaire. "Multi-Paxos : An Implementation and Evaluation". In: 2009 (cit. on p. 12).

[11] I. Foster, C. Kesselman, and S. Tuecke. "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". In: *The International Journal of High Performance Computing Applications* 15.3 (2001), pp. 200–222. DOI: 10.1177/109434200101500302. eprint: https://doi.org/10.1177/109434200101500302. URL: https://doi.org/10.1177/109434200101500302 (cit. on p. 5).

[12] A. S. Foundation. *Apache Cassandra, open-source distributed nosql database*. Accessed Jan. 2023. URL: https://cassandra.apache.org/ (cit. on p. 9).

[13] A. S. Foundation. *Apache Lucene, open-source search engine software library*. Accessed Jan. 2023. URL: https://lucene.apache.org/ (cit. on p. 15).

[14] P. Fouto, N. Preguiça, and J. Leitão. "High Throughput Replication with Integrated Membership Management". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, 2022-07, pp. 575–592. ISBN: 978-1-939133-29-67. URL: https://www.usenix.org/conference/atc22/presentation/fouto (cit. on pp. 12, 24).

[15] P. Fouto et al. *Babel: A Framework for Developing Performant and Dependable Distributed Protocols*. 2022. DOI: 10.48550/ARXIV.2205.02106. URL: https://arxiv.org/abs/2205.02106 (cit. on pp. 1, 19, 20).

[16] P. Fouto et al. "Babel: A Framework for Developing Performant and Dependable Distributed Protocols". In: *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. 2022, pp. 146–155. DOI: 10.1109/SRDS55811.2022.00022 (cit. on pp. 1, 19).

[17] Google. *Google/cadvisor: Analyzes resource usage and performance characteristics of running containers*. Accessed Jan. 2023. URL: https://github.com/google/cadvisor (cit. on p. 14).

[18] P. Gouveia et al. *Kollaps: Decentralized and Dynamic Topology Emulation*. 2020. DOI: 10.48550/ARXIV.2004.02253. URL: https://arxiv.org/abs/2004.02253 (cit. on p. 20).

[19] *Grafana - Query, visualize, alerting observability platform*. https://grafana.com/grafana. Accessed Jan. 2023. 2023 (cit. on p. 17).

[20] *Graphite - enterprise-ready monitoring tool*. https://graphiteapp.org. Accessed Jan. 2023. 2023 (cit. on pp. 14, 17).

[21] P. Hunt et al. "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Boston, MA: USENIX Association, 2010, p. 11 (cit. on p. 10).

[22] J. Kreisa. *Docker index: Dramatic growth in Docker usage affirms the continued rising power of developers*. Accessed Jan 2023. 2020-07. URL: https://www.docker.com/blog/docker-index-dramatic-growth-in-docker-usage-affirms-the-continued-rising-power-of-developers/ (cit. on p. 12).

[23] J. Kreps. "Kafka : a Distributed Messaging System for Log Processing". In: 2011 (cit. on p. 10).

[24] L. Lamport. "Paxos Made Simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001-12), pp. 51–58. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/ (cit. on pp. 10, 12, 24).

[25] J. Leitão, J. Pereira, and L. Rodrigues. "HyParView: a membership protocol for reliable gossip-based broadcast". In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Edinburgh, UK, 2007, pp. 419–429 (cit. on pp. 10, 11, 24).

[26] L. Leonini, É. Rivière, and P. Felber. "SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze)". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, pp. 185–198 (cit. on p. 18).

[27] *proc(5) Linux User's Manual*. 5.13. 2021-08. URL: https://man7.org/linux/man-pages/man5/proc.5.html (cit. on pp. 12, 13).

[28] *Prometheus - Monitoring system & time series database*. https://prometheus.io/. Accessed Jan. 2023. 2023 (cit. on pp. 14, 16).

[29] J. Case et al. *Simple Network Management Protocol (SNMP)*. RFC 1157 (Historic). RFC. Fremont, CA, USA: RFC Editor, 1990-05. DOI: 10.17487/RFC1157. URL: https://www.rfc-editor.org/rfc/rfc1157.txt (cit. on p. 14).

[30] K. McCloghrie and M. Rose. *Management Information Base for Network Management of TCP/IP-based internets: MIB-II*. RFC 1213 (Internet Standard). RFC. Updated by RFCs 2011, 2012, 2013. Fremont, CA, USA: RFC Editor, 1991-03. DOI: 10.17487/RFC1213. URL: https://www.rfc-editor.org/rfc/rfc1213.txt (cit. on p. 15).

[31] M. Roser, H. Ritchie, and E. Mathieu. "Technological Change". In: *Our World in Data* (2013). https://ourworldindata.org/technological-change (cit. on p. 1).

[32] M. Roser, H. Ritchie, and E. Ortiz-Ospina. "Internet". In: *Our World in Data* (2015). https://ourworldindata.org/internet (cit. on pp. 1, 8).

[33] H. Rostami and J. Habibi. "Topology awareness of overlay P2P networks". In: *Concurrency and Computation: Practice and Experience* 19.7 (2007), pp. 999–1021. DOI: https://doi.org/10.1002/cpe.1089. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1089. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1089 (cit. on p. 1).

[34] I. Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications". In: *IEEE/ACM Trans. Netw.* 11.1 (2003-02), pp. 17–32. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.808407. URL: https://doi.org/10.1109/TNET.2002.808407 (cit. on p. 22).

[35] T. R. H. E. L. Team. *Red Hat continues to lead the Linux Server Market*. Accessed Jan. 2023. 2018-08. URL: https://www.redhat.com/en/blog/red-hat-continues-lead-linux-server-market (cit. on p. 13).

[36] Q. Vu, M. Lupu, and B. Ooi. *Peer-to-Peer Computing - Principles and Applications*. 2010-01. ISBN: 978-3-642-03513-5. DOI: 10.1007/978-3-642-03514-2 (cit. on p. 8).