Automatic Scaling of Web Services using an Adaptive Distributed System

David Jaenson

Tuan Nguyen

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6037
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2017

Abstract

In this thesis a Mesos framework is built to enable dynamic scaling of a collection of HTTP application programming interfaces (APIs) in response to a varying request workload. Several scaling algorithms ("policies") that run on top of this Mesos framework are designed and built. Furthermore, a performance evaluation framework is designed and built. The evaluation framework is used to evaluate the scaling algorithms running on top of the Mesos framework. The evaluation is performed by emulating two real world APIs using request data extracted from a real world system's access logs. Based on this evaluation the positive and negative attributes of the design of the Mesos framework and the policies are discussed.

Acknowledgments

We would like to thank our supervisor Johan Eker at the department of Automatic Control at Lund University for his extensive help and his invaluable feedback. We would also like to extend a thank you to our examiner Karl-Erik Årzén at the department of Automatic Control at Lund university for his input and feedback on the thesis. Furthermore, we would also like to thank Per-Gustaf Stenberg, our supervisor at Data Ductus for his help and suggestions regarding the design of the Mesos framework. We would finally like to thank Data Ductus for allowing us to use the workload data from their APIs as well as the descriptions of the APIs running in their data center. We would also like to extend a thank you to everyone else who read our thesis and provided us with feedback.

Contents

1

Introduction

With the advent of cloud computing platforms that enable rapid scaling of a system's resources, the importance of building durable and efficient systems that can take advantage of horizontal scaling has increased. By increasing the number of independent machines used in the system (i.e. horizontal scaling) virtually unlimited scaling of resources is possible. The steadily declining cost of using cloud computing platforms such as Digital Ocean [1] and Amazon Web Services [2] (AWS) make them more and more appealing. For example, the price per hour for running one small Amazon Elastic Cloud instance has declined from $0.085 in 2010 to $0.023 in 2017 [3] [4].

The cloud has not only simplified the process for small-scale operators to scale their systems. Large scale web services, such as Netflix [5] and Quora [6], that previously would have had to run their own data centers are now using these cloud computing platforms [7] [8]. By moving from a private data center to the cloud, organizations can simplify their infrastructure and focus their attention on their core product.

The renting of resources on cloud computing platforms is often priced hourly. Many services hosted in the cloud use static allocation, i.e., they use a fixed number of servers. The number of servers used is chosen such that they are able to handle some fixed peak workload. However, this also means that a statically allocated system is often over-provisioned since at most times the workload is not at the possible maximum. To take full advantage of the possible cost reductions that the cloud enables there is a need to reduce the average provisioned resources of a system. As most web services have a varying work-load during different times, a cost optimized system should decrease the provisioned resources it uses during times when there is a small workload and increase the resource usage during higher workloads. The larger the system and the more variance in the workload the larger the impetus is for taking advantage of the dynamic scaling possibilities of the cloud. By utilizing dynamic scaling instead of static allocation, the average resource usage can in many cases be decreased. This would, in turn, lead to decreased operating costs for such a system.

Even before the advent of cloud computing, there was a need to efficiently use and administer large computational clusters. Demanding computations needed systems that were able to scale horizontally instead of vertically (increasing the computational power of an existing machine) due to the physical limitations of scaling a single machine. Large organizations, such as Google [9], Facebook [10] required a way to efficiently administer and take advantage of the resources of their large data centers. A multitude of distributed applications needed to be able to run concurrently in these clusters to achieve an efficient resource usage. Hence this led to the emergence of cluster managers such as Borg [11], Omega [12] and Mesos [13]. These cluster managers [14] enable developer to build distributed applications on top of them. They have in recent years significantly simplified the process of building distributed frameworks and applications. They abstract the details of the underlying distributed system's resources and make them available to distributed frameworks and applications via an application programming interface (API). A lot of the issues related to large distributed systems, such as fault tolerance and messaging, are also significantly simplified by using these systems.

Data Ductus [15] is a Swedish IT consulting company. The company has its own data center where it hosts a number of web services on behalf of its customers. Currently the services hosted use a fixed amount of server resources at all times. In this thesis we explore the possible resource reductions that could be realized by either moving to the cloud or by using a cluster manager to share the resources of the data center among multiple such services. Since the request rate to different services vary over time and it is difficult beforehand to determine the correct amount of resources needed for each API, we propose to build a distributed framework to dynamically scale these services as needed based on the workload. This distributed system itself can react to and predict the behavior of the incoming workload to each individual service. Thus the system should be able to adjust the resource allocation to each service dynamically and therefore optimize the resource usage. The framework designed in this thesis aims at alleviating the suspected resource over-provisioning.

1.1 Approach

A rudimentary distributed framework will first be implemented. This framework will be able to deploy instances of services to different machines in the system statically (static allocation). By stress testing this framework, a baseline performance for the developed framework will be acquired.

The basic distributed framework will then be extended with a scheduling system in an attempt to minimize resource usage while still maintaining the service level agreement (SLA). The services will also be implemented with different characteristics in an attempt to emulate the real world services of Data Ductus.

Request data extracted from Data Ductus' access logs will be used to evaluate, via data replay, the correctness and performance of the framework. Different policies will be developed to try different strategies to minimize resource usage while maintaining the service level agreements.

1.2 Contributions

The project consisted of a few different components. All components were designed by both authors by discussing what parts were necessary. The base distributed framework which was built on top of Mesos was built primarily by David Jaenson with many contributions by Tuan Nguyen. The component which emulates the properties of the Data Ductus services was built primarily by Tuan Nguyen. To be able to send requests to the framework a workload emulation component was necessary. That component was primarily built by Tuan Nguyen. To be able to distribute the incoming requests to all the active service deployments a load balancer configuration was needed. This configuration was made by David Jaenson. The policies which determine how and when service instances should be deployed or torn down were built by both authors.

1.3 Structure of Thesis

- **Introduction** This section present the thesis context and structure of the thesis. A brief introduction to cloud computing, cluster managers and distributed systems is given.
- **Background** This section present the relevant background to the different subjects of the thesis. The topics of cloud computing and distributed systems are discussed. The most essential tools and systems used in thesis are introduced and discussed (Mesos and Nginx).
- **Problem definition** This section presents the improvements made possible by using dynamic scaling as opposed to static allocation. A general description of the problem that this thesis aims to investigate is given. Afterwards, a

specific description of the specific problem that this thesis attempts to solve is given.

- **Architecture** This section presents how the framework was built, its architecture and the design trade-offs. A description of what specifically is demanded of a working system is given. It goes into detail which external frameworks and systems were used and why these specific tools and systems were chosen.
- **Results** This section presents the results in a tabular format. These results show the performance of the policies used under different input data sets. A few plots are presented here as well to highlight specific details regarding certain policies.
- **Discussion** This section presents the discussion of the results. The results of each policy under each of the input data sets is discussed in detail. Possible error sources are discussed. Possible improvements that were not implemented due to time constraints are mentioned.

2

Cloud computing

Cloud computing allows sharing and usage of remote resources, such as data, computing power and services on demand via a network connection. It provides users with a network of configurable resources that are scalable with minimal configuration. Cloud computing also allows users to access such resources from a distance [16]. This allows an organization to ignore the gritty details of taking care of computer infrastructure, with less maintenance and more ease of resource management. A visualization of the cloud can be seen in Figure 2.1. The user only needs a network connection to access the resources of the cloud, it is generally device agnostic.

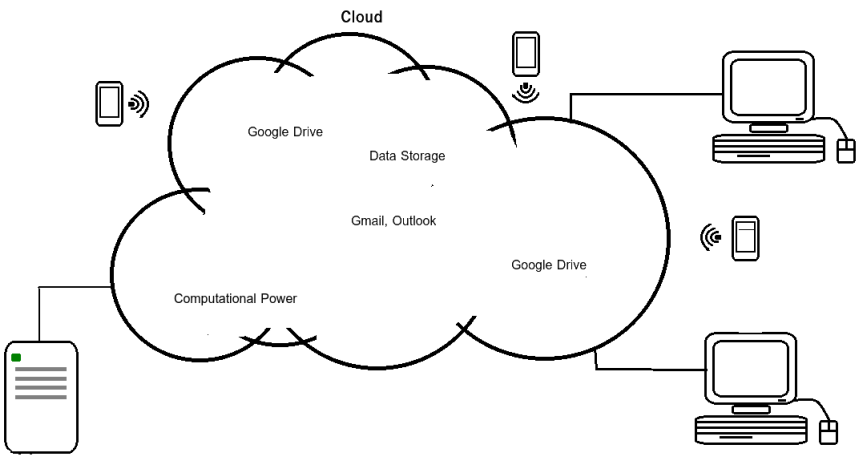

Figure 2.1 A visualization of the cloud. The cloud allows access to computational resources over a network connection from a wide variety devices.

2.1 Essential Characteristics

The national institute of standards and technology (NIST) from U.S. Department of Commerce [17] has defined a model of cloud computing. It defines cl oud computing [18] via its essential characteristics, service models and deployment models.

- On-demand self-service: The end users are able to access and request the resources needed by themselves. The system should be transparent such that users can understand the cost, quality and drawbacks. The user can use the system without the need to confer with the service provider.
- Broad network access: Cloud services are accessible using a heterogeneous collection of devices such as computers, tablets and mobile phones. The resources can be accessed from multiple sources and locations.
- Resource pooling: Resources such as storage, processing and memory are grouped together and are offered to multiple customers. This increases the resource utilization and increases the available resources to the end customers. The customer's need will be catered to without apparent limitations on resources and can be changed at will.
- Rapid elasticity: Resources appear, to the end user, as unlimited or immediately available. The customer does not see the details of allocations or deallocations of resources and is able to scale the needed resources on demand.
- Measured service: Every component in the cloud system is measured in some way that allows transparency. The customer who uses the cloud service can see exactly the cost of any action performed, such as scaling more resources, adding extra services and so on. This provides both the service provider and the end customer with a usage history, and ensures that a change in the system by either party will be recorded.

2.2 Service Models

Cloud computing offers different services such as software-, platform- and infrastructure related services. The computing stack can be divided to three different layers: software-, platform- and infrastructure as a service [19, pp. 12-17, 55-70]. Each layer can be built on top of each other and resemble a pyramid as illustrated in Figure 2.2. The different abstraction layers allow more focus on building the core software while abstracting certain layers to third party providers.

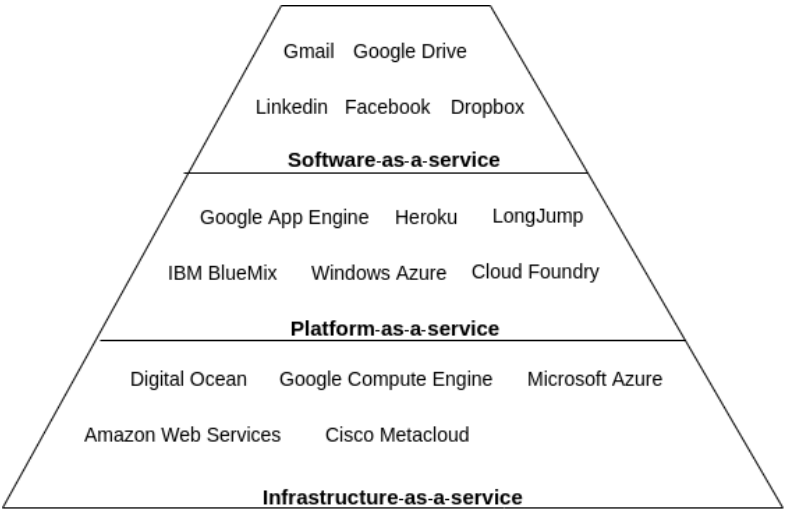

Figure 2.2 The cloud computing stack can be categorized as three layers building on top of each other. The infrastructure-as-a-service layer allows access to, for example, virtual machines. The platform-as-a-service layer gives access to a platform that hosts a pre-configured stack of tools, thereby shortening development cycles, for example, by automatically scaling resources. The software-as-a-service model offers access to cloud applications, for example, APIs or software accessed via a web browser.

Infrastructure-as-a-service The infrastructure-as-a-service (IaaS) model provides customers direct control over the hardware, or the virtualized hardware. The user can monitor and manage all resources that are offered such as networking, data storage and computational power. Instead of having to manage a physical data center, users can use IaaS instead. Examples include Amazon Web Services [2] (AWS) and Digital Ocean [1].

Platform-as-a-service The platform-as-a-service (PaaS) model offers customers a platform and an environment as a service. The model offers, for example, operating systems and pre-installed packages. This makes available to devel-

opers a platform to build software without needing to manage the hardware that lies below nor the management that come with it. The platform can be very specific such as a web server platform only for a specific frameworks or programming languages. This may result in a shorter development cycle and cost reductions. Applications built to run on a PaaS infrastructure also have the advantage of the cloud characteristic such as broad network access, scalability and much more. Examples of PaaS include Google App Engine [20] and Heroku [21].

Software-as-a-service The software-as-a-service (SaaS) model offers customers cloud applications as a service. The applications are often accessible through a web browser. Examples include Google Docs and Google Maps. This delivery model removes the need of having to download or install software on individual physical machines. It therefore allows for ease of use for the customer but it also allows the service provider to manage the application's configuration and updates to the software since everything resides on a host that belongs to the vendor.

2.2.1 Deployment Models

The cloud model can be deployed in different ways depending on who the client is [19, pp. 18-22]. For a company that requires great security and more control over the resources, a private cloud deployment model is generally preferable. A public deployment model allows for more resource flexibility but multiple clients will share the same resources. If multiple companies have the same configuration, a community cloud can offer more specific computational power. Sometimes, a hybrid model is better since it can use the advantage of being a mix of different models.

Public cloud The public cloud model is a deployment cloud model where all resources are publicly available. This is the most commonly used model since it offers flexibility and is cost effective. The customer has no direct control over the infrastructure's specific location. The resources inside the cloud are often shared between between multiple users with a multi-tenant system.

Private cloud The private cloud model is a cloud model in which only a single organization can access the cloud. The model ensures better control and privacy over the resources that are offered. The platform is often deployed in a secure environment behind a firewall and only offers access to authorized personnel. This gives a greater security and control over the data and resources to the user.

Community cloud The community cloud model is a deployment cloud model in which many organizations can share the resources of the cloud. This is often suitable for a set of organizations that have the same type of resource needs.

Hybrid cloud The hybrid cloud model is a model combining different cloud deployment. It allows better security than a public cloud and higher resource availability than a private cloud. Sensitive operations can then be operated in the private cloud service and the less sensitive operations can be performed in the public cloud.

2.3 Quality of Service

Cloud computing offers great flexibility with virtually unlimited resources. Quality of Service (QoS) is a very important term in the context of the cloud. It represents the quality of the provided service. For example, the availability, scalability and reliability of the provider's resources. However, providing a higher quality service typically leads to higher operational cost for the cloud provider.

2.3.1 Service level agreements

A service level agreement (SLA) [19, pp. 127-135] is an agreement between a servicing party and a client detailing some sort of requirement that the servicing party must maintain for the service supplied to the client. Examples of SLAs for web services include uptime requirements and response time requirements. These requirements are often specified using the nines notation to describe the required reliability of the SLA. For example, a system wherein an SLA uptime is described to be five nines (5N) should guarantee that the system is online 99.999% of the time. An SLA typically includes a clause specifying damages in the event that the servicing party violates the requirements of the SLA. These damages may be quite substantial which incentivizes the service provider to provide the specified Quality of Service defined in the SLA.

2.4 Privacy and Security

The services that are provided by the cloud are many. Services such as Overleaf, Dropbox and Google Docs are used every day and their privacy is an important part of the service quality. Potentially sensitive information is stored in cloud. Information such as authentication details, sensitive files and credit card details may be stored. A database leak from any cloud service could affect millions of individuals.

2.5 Cloud Computing Services

The landscape of cloud computing platforms is varied. The cloud provider can focus on only infrastructure, such as Digital Ocean, but more often, there are multiple services that are provided in one platform, often both infrastructure and platform services. By combining both infrastructure and platform services together, many

providers can efficiently provide the customer all needed resources on a single platform, thereby enabling a simple method for control and cost overview. The platforms often expose an API where the customer can programmatically alter their usage of the platform's services. This enables simplified upscaling and downscaling.

2.5.1 Software-as-a-service

Google Drive Google Drive is a software-as-a-service platform that provides users with storage and integrates with a multiple of different services such as Google Doc and others [22]. The services are free of charge and come with 15GB storage for free as of 2017. If users require more storage, it can be purchased for a monthly subscription [23].

Microsoft Office 365 Microsoft Office 365 [24] is another very popular SaaS platform that provides its users with a complete suite of online productivity tools. The service also offers space to store and backup files. A comparison of the pricing of both Microsoft Office 365 and Google Drive can be seen in Table 2.1.

Table 2.1 Prising comparison of SaaS storage platforms, as of 2017-05-27.

	Free	Monthly
Google Drive	Yes 15 GB online space	100 GB - 1.99$ 1 TB - 9.99$ 10 TB - 99.99$
Microsoft Office 365	No	1 TB - 12.5$

2.5.2 Platform-as-a-service

Google App Engine Google App Engine [20] can be categorized as a platform-as-a-service. Google App Engine can be used together or separate from Google Compute Engine . It can be used to build scalable applications with built in services and many APIs that can enhance the speed of development of web services by offering logging, databases like CloudSql, memcache and user authentication. Google App Engine can automatically detect the workload of the application running and scale the service in response.

Heroku Heroku [21] is another platform-as-a-service platform much like Google App Engine. Heroku's platform provides developers with an extensive set of APIs that allow developers to focus more on development of their application and less on infrastructure. Heroku has over 140 add-ons available in their ecosystem that enable faster development and a simplified overview of an application.

2.5.3 Infrastructure-as-a-service

Digital Ocean Digital Ocean is an IaaS provider. It is a cloud service that enables developers and system administrators to use virtual machines by an on-demand pricing model [1]. DigitalOcean calls these virtual machines "droplets". The cost of a droplet on DigitalOcean depends on the the number of cores, the amount of RAM and disk space the customer demands. The prices are for hourly usage. As an example, the hourly price of a virtual machine having 2 CPUs, 2 GB RAM and a 40 GB SSD drive is $0.030 as of November 2016 [25]. It is not possible to scale resources individually, that is, droplets can be resized by switching to a higher performance droplet configuration overall (increasing CPU, RAM and disk space) but it is not possible to only increase one of these resources of a droplet.

Amazon Web Services Amazon Web Services [2] (AWS) is a collection of cloud computing services provided by Amazon. Amazon Elastic Compute Cloud (EC2) provides virtual machines which are rentable by customers. EC2 allows customers to rent virtual machines based on three different time-based renting options. On-demand instances are priced hourly, with the price rounded up for every started hour. Reserved instances are rented for longer periods of time and are priced with a discount as compared to on-demand instances. Amazon also allows customers to rent instances based on spot-prices in which a customer is able to "bid" on renting instances which may be cheaper than on-demand instances. The spot prices are calculated hourly, rounded up.

Google Compute Engine Google Compute Engine [26] is an IaaS platform where Google offers high performance machines to customers. The machines can be customized according to customer needs. The pricing is minutes based, with the prices rounded to the nearest 10 minutes. Google also offers discounts on long running machines of up to 40% of the full price. The pricing is also based on the machine's type and the amount of persistent disk storage.

3

Distributed Systems

A distributed system is a system where components are connected and communicate with each other via message passing. The components' available resources are shared and coordinated in the system. To the user these inter-connected components are hidden and the user communicates with the distributed system through an interface. Examples include multi-core computers and the Word Wide Web. A distributed system can be composed of heterogeneous components, such as stationary computers and laptops. An example of different software components are different operating systems [27].

The components inside a distributed system are heterogeneous and autonomous. A component can have its underlying operating system handle its hardware and resources while the distributed system's layer lie between the higher layer such as the user interface, and software and the hardware layer to coordinate the system. The distributed system layer usually refers to the middle layer inside the system.

A distributed system can be defined as a system where all coordination and communication of the system is performed via message passing over some network [28, p.2].

3.1 Distributed System Architecture

Distributed systems may physically consist of multiple different computers, where they connect and transfer information through a network. However, from the user's point of view, the system is regarded as a singular system with the combined resources and computational power of its nodes. There are many different models of distributed systems.

Client-Server The client-server model uses a request/response communication paradigm where clients send requests and awaits responses from server.

Two tier model The simplest system with a server which responds to clients' requests.

Multi tier model Separates the logic more so that it is possible to have multiple different layers handling different types of logic.

Manager-Agent The manager coordinates and assigns work to each agent that is in communication with the manager. Agents usually do not communicate with each other.

Peer-to-Peer There is no central authority or coordination. All peers are equal participants in the network.

3.1.1 Client-Server model

The communication consists of requests and responses where clients send requests to a specific target and the server will process the request, and return an appropriate response, see Figure 3.1 [29]. The world wide web (WWW) can be viewed as a

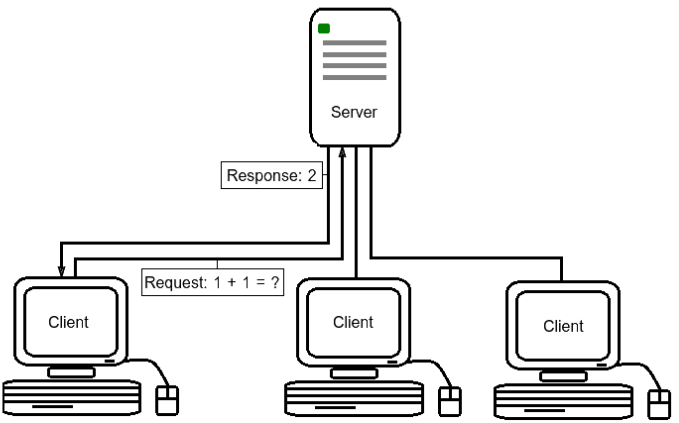

Figure 3.1 Client Server Model. A one-to-many model where multiple clients sends requests to a server which the server in turn sends a response to.

very large distributed system where resources are hosted by different servers. These resources are presented to the user through a web browser. The three components that WWW are based on are:

- HyperText Markup Language (HTML) is a markup language use to describe a website.

- Uniform Resource Locators (URLs) which specify a resource's location in the network.
- A Client-Server Architectural model where a server provides services to clients who request them.

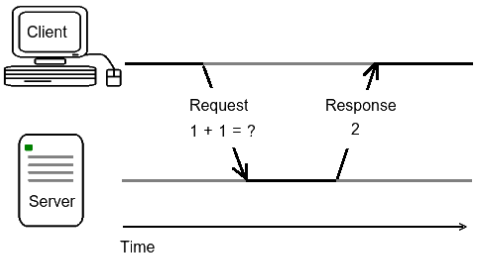

Figure 3.2 A request and a response between a client and a server. The client in this example sends a query to the server, which the server later responds to.

Figure 3.2 shows the basic flow of the client-server model. The server has different services that are offered to the clients. One such service may be to perform addition. In this case the request response flow could be as follows: the client sends a request to the server to compute (1 + 1) and receives a response (2) from the server. Websites on the Internet work like this. Users, by using a web browser (Google Chrome, Firefox), access a specific server located by a uniform resource locator, URL, such as http://www.google.com?search=query.

Two tier server-client model Two tier models are the simplest architecture for the server-client model. It consists of a single server that responds to multiple clients. The server is responsible for communication and coordination between clients. This model can also further be categorized into the thin-client or fat-client models, illustrated in 3.3.

Thin-client model Most websites are built with this client model, where the client are responsible for the presentation layer. The server is responsible for all other layers, which include, business, data and processing layers. Since the client does not handle too much logic, this will place a larger burden on the server. Most websites work this way. For example Google search has a very lightweight client interface while the logic layer server-side is much more complex.

Fat-client model Massive multiple player games are often built in this way with more resources needed on the client side. The client handles not only the data presentation, but also much of the processing. The server is often responsible for coordination of the communication and keeping track of the global state.

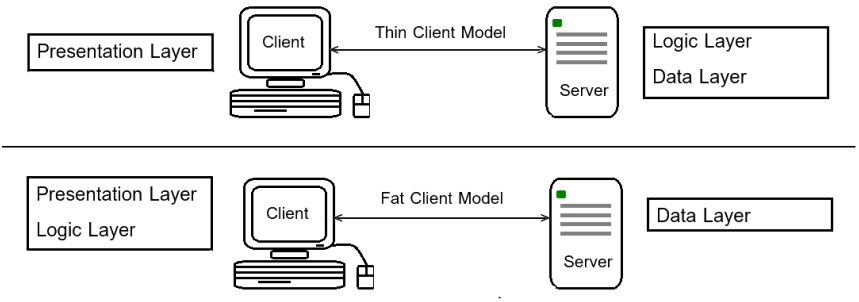

Figure 3.3 An illustration of the thin-client and fat-client models. In the thin-client model most of the logic is handled server-side. The fat-client model is more resource demanding for the client as the application logic is primarily performed client-side.

Multi tier server-client model The architecture of the server can be split to multiple tiers to support the workload and enable separation of concerns. Each layer can be running on different hosts. This adds a degree of flexibility for the distributed system. Developers can add more layers to increase the performance. For example, one layer may handle interacting with the client while another layer handles the database and storage. For example, a three tier architecture can separate the server into different layers handling logic and data, as seen in Figure 3.4. In this example, when a request is sent, the logic layer authenticates the client and performs the necessary calculations in order to retrieve the data from the data layer that is to be sent back to the client.

3.2 Manager-Agent model

The manager-agent relationship is a common way to organize a distributed system. In this setup one or more managers distribute workload to a number of agents. These agents next execute their assigned work. Many distributed systems with this architectural model are built for a specific purpose that comes with distinct types of jobs. For example, a long running job that requires high priority and short running jobs that have a low priority. Utilizing information regarding what types of work that the manager will distribute to its agents allows the manager to efficiently distribute

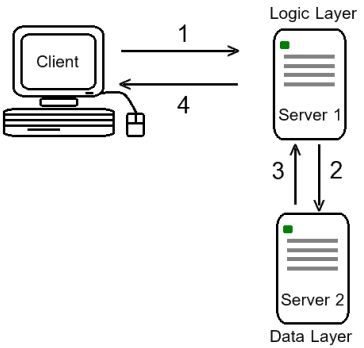

Figure 3.4 The multi-tier model adds a degree of flexibility and separation of concerns by placing different kinds of logic into different layers. For example, the server-side logic can be subdivided into different layers handling the routing/controller logic while a separate layer handles all the database logic. In the example above a client sends a request to a server (1). The logic layer requests data from the data layer (2) based on the client's request. The data layer responds with the appropriate data (3). The server responds to the client with data (4).

the workload across agents [29]. Figure 3.5 shows a manager-agent architectural example. The manager has three agents. The manager sends a request to an agent to perform a calculation of 1 + 1. The agent next performs the work and once the computation is complete responds back to the manager with the result. For a user outside of the system, the interaction layer will be with the manager and the agents that perform the work are hidden.

3.2.1 Peer to Peer

The peer to peer model is a decentralized model where there is no distinction between nodes. Each node is both a consumer and producer in the system. This model allows a great flexibility of being able to add new resources or services as needed and this model allows for large scale systems. The communication often happens through middleware, like a message broker, where services can update and provide new information to each other as needed [29]. A common protocol employing this method is the Bittorrent protocol [30]. Each peer inside the system will both provide and consume resources unlike the previously mentioned models. The specific network layout of a peer-to-peer network can vary as is illustrated in Figure 3.6.

- Ad-hoc networks do not constrain the connection into any particular form. Such systems are scalable, and fully decentralized. But in order to find any

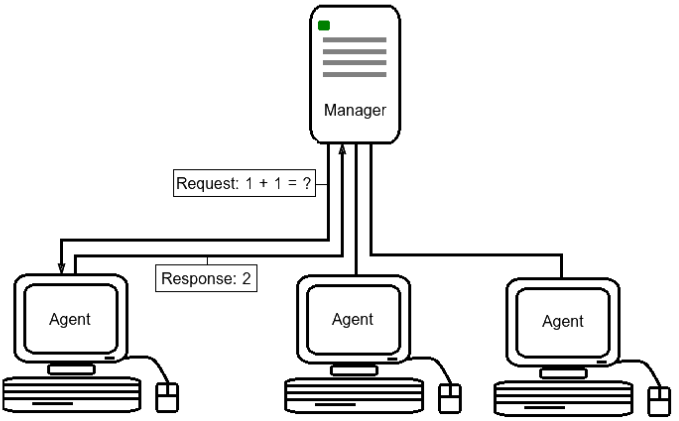

Figure 3.5 In the manager-agent model a manager has many agents to which it distributes a workload. Note that the relationship is the opposite compared to the client-server architecture: one manager node requests work to be performed by several agent nodes. When an agent node has completed its work it responds to the manager with the results of the work.

particular resource in the system, a node needs to poll every node which can be very inefficient in systems with a large amount of nodes.

- Structured networks may be arranged in a structure that allows the system to find resources in an efficient way. For example, a tree structure where each node which joins the network connects to its parent node. This may allow nodes to more quickly identifying the location of resources.

3.3 Scalability

In a distributed system, scalability is one of the most important aspects to consider. If the system is under an increasing workload, such as more users are using it, then the system should be able to accommodate such increases by adding additional resources to the system. The system should be able to perform well under increases of usage or increases of data. There are two primary ways of scaling a system, see Figure 3.7, horizontal and vertical scaling.

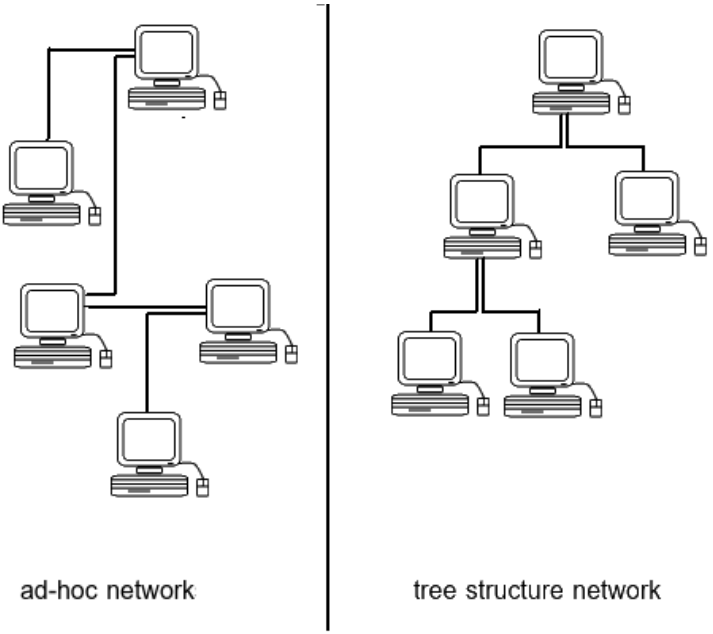

Figure 3.6 The structure of peer-to-peer networks vary. In peer-to-peer networks all nodes are equal participants of the network. They both consume and produce resources as opposed to the consumer-producer relationships of both the client-server and manager-agent models previously mentioned. Resource discovery in ad-hoc peer-to-peer networks may involve polling every single node in the network. Using a structured network, such as a tree structured network, may allow faster resource discovery.

Scaling Vertically Vertical scaling is when resources are added to a single node in the system, for example, more RAM or CPU cores to a single computer. The drawback to this method of scaling is that the costs of adding to a single computer grows exponentially. Upgrades also result in downtime when the node is going through the upgrade, and this results in the data and service temporarily becoming unavailable.

Scaling Horizontally Horizontal scaling is when new nodes are added to the system. In the context of the manager-agent model new agents are added, in a client-server architecture new servers are added and in the peer-to-peer model new nodes are added. This is often the preferred method of scaling since it allows for more flexibility to the system and the hardware costs remain linear.

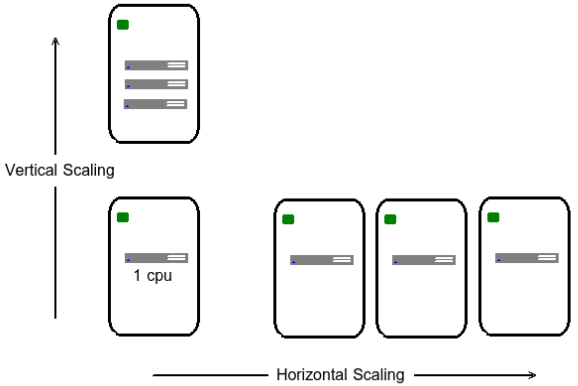

Figure 3.7 By scaling horizontally new nodes are added to the system. By scaling vertically more resources are added to an already existing node.

In order for a distributed system to be scalable, the system should be able to achieve:

- Linear efficiency: If one single resource is enough to support 10 users, upon adding another resource to the system, the system should be able to support 20 users.
- Data efficiency: In order to find data, even in a large data set, the performance loss should be at most logarithmic.
- Software resource scalability: There should be no hard limit for the number of resources that are deployable. An example in violation of this principle is the original IP-protocol since it is currently running out of IPv4 addresses and soon will hit a hard limit and thus will be unable to scale further.

3.4 Failure Management

Fault tolerance in distributed systems is different from fault tolerance in a singular machine where a hardware failure leads to the collapse of the entire system. A distributed system needs to be able to handle unexpected failures of its components without compromising the stability of the system and incurring downtime. Inside a distributed system where multiple components and software services are in cooperation, the probability of failure in the system increases as more nodes are added to the system. The distributed system needs to be able to handle partial failures gracefully. If any component failure is experienced, the system should be able to continue

to operate as if though the component was meant to be removed and thus this should only should affect the linear efficiency.

Redundancy can be used to handle the failure. By always have at least one backup node to execute an action, such as duplicate databases stored in different nodes, the system will provide data access even if one physical host fails. In other words, this prevents the problem of a single point failure.

3.5 Resource sharing

One of the most valuable aspects of distributed systems is that resources can be shared across the network to the user independently of their location. For example, in an office, it is common to share the same printer instead of having one printer for each employee.

Distributed systems consist of numerous separate components. The resources that belong to these components are, ideally, distributed inside the network to optimize the usage. For example, if a large job requires more disk than a computer has to offer, another computer's disk may be used to store the remainder of the required data.

Resources are shared via a network where components do not need to be physically connected with each other. Access transparency and location transparency are directly related to resource sharing mechanisms in a distributed system. In terms of transparencies, when a printer is shared to the system, the user should be able to access and use the printer without needing to know the physical location of the printer itself.

3.6 Resource Discovery

Node discovery inside the system allows for a dynamic distributed framework. This would provide a way to dynamically increase and decrease the needed resources when necessary.

For distributed systems that follow the manager-agent model, the resource discovery is simple. The manager provides the necessary connection information to the agent. When a new agent connects to the manager with the provided information, its resources will be added to the system.

The peer-to-peer resource discovery requires more effort as there is no central system that can be notified when new nodes are added to the system. When a new node is added to the system this addition needs to be propagated to every node in the entire system for it to be fully visible to every other node.

3.7 Resource allocation

A distributed network can have access to an abundance of resources due to resource sharing. Different jobs inside the system will require different amounts of resources and therefore need to be scheduled correctly to maintain an efficient use of resources. The resource allocation needs to take many dimensions into consideration such as the job priority, when a job starts and the duration of that job. Most often, a distributed system comes with a resource allocation algorithm where these attributes are involved.

3.7.1 Resource scheduling

A scheduling policy is an algorithm that distributes workload to a group of workers, for example, agents in the context of an manager-agent model. The most ubiquitous scheduling policy in the context of load balancers is round robin which simply distributes the workload to workers in circular order. Scheduling policies are also used for distributing workloads in distributed systems. In the context of a manager-agent setup you typically have a manager that distributes workload to a number of agents based on a scheduling policy. The scheduling policy attempts to model how to schedule resources to agents such that an optimized resource utilization is achieved while still allocating enough resources such that the system is not underprovisioned. A good scheduling policy therefore must take into consideration what type of data it is to schedule and how much resources are expected to be required at any one point in time.

Reactive scheduling Reactive scheduling refers to a scheduling policy that reacts to immediate events. For example, a reactive policy in the context of scheduling is a policy that scales up/down the system in the case of an increase/decrease in the number of requests for resources.

Predictive scheduling Predictive scheduling refers to a scheduling policy that attempts to predict more long-term trends in the workload. For example, a predictive policy in the context of HTTP might look at the number of requests that arrived during a particular time during the day and assume that the same number of requests will arrive at that same time the next day and therefore preemptively schedule more resources during that time of day.

3.8 Resource isolation

A computer needs to provide its resources to different processes, processes also have different priorities depending on the process. This means that some processes are allowed to utilize more of a certain type of resource. In Linux, control groups can be used to isolate resources. In the same sense, inside a distributed system, one

service may require more resources than another. Each computer has access to a certain amount of resources of which they will provide to the distributed system.

Virtualization and containerization [31] are both ways that are used to isolate operations on a physical machine. This allows multiple different virtual systems to be isolated from each other while still operating inside a single physical machine.

3.9 Distributed Frameworks

A distributed framework is the framework that is built to accomplish some task in a distributed system. Different distributed framework have different goals, what they have in common is that they use distributed resources to accomplish these goals [29].

3.9.1 Mesos

Mesos is a system for scheduling other distributed frameworks. It can be described as a lightweight distributed kernel or a cluster manager. In Mesos a number of managers elect a running master when a quorum among the managers has been achieved. The elected manager enables the scheduling of the connected agents' resources via other distributed frameworks which are run on top of Mesos. Mesos agents announce resources to the Mesos manager which then forwards these resource offers to the distributed frameworks. Thereby Mesos acts as an abstraction layer allowing the distributed frameworks to schedule resources to agents without requiring that the distributed frameworks have any detailed knowledge or communication with the agents. Simply put, Mesos acts as the kernel while frameworks, which are similar to processes on a traditional (localized) system, are offered resources such as CPU, disk and memory [13].

3.9.2 Borg

Borg is a cluster management system. Borg was built due to a need for unifying scheduling two different types of jobs: long-running services and batch jobs. [14]. These two job types previously were scheduled by a framework and maintained by two different systems. With Borg, the machines that were previously used by the two systems could be shared and therefore the resource utilization could be increased. By assigning a priority and a quota to tasks, the system could make decisions about either assigning tasks to machines or reject them. The scheduler of Borg used a scheduling algorithm where it used *feasibility checking* and *scoring*. *Feasibility checking* finds machines that can offer the resources that is necessary and *scoring* is used to find the most appropriate machine [11].

3.9.3 Omega

Omega is a proposal for a cluster management system. The framework uses a different approach to scheduling resources by using a shared state architecture. The

system divides the jobs into two categories. The first category is comprised of many short term running jobs. The second category has much fewer jobs, but these jobs have a longer duration, and this category also has a lot more important jobs that have higher priority. The problem introduced by these two jobs when running together is while it take only a few seconds to schedule long time running jobs, these will block the other short term jobs until a resource allocation decision has been made. This is solved by Omega by using parallelism and a shared state architecture. Omega maintains different schedulers for different jobs where each scheduler maintains an exact copy of the system. Each scheduler makes its own decisions about resource allocations and job scheduling. When the decision has been made, an attempt to change to the desired state will be made [12].

4

Mesos

Many distributed applications are typically run in concert in a data center. As the workload for different applications vary there is a need to be able to allow resource sharing among these distributed applications. Mesos [13] is a cluster manager that allows for this sort of resource sharing. It accomplishes this by acting as a distributed systems kernel to the distributed applications that run on top of it. Thus, Mesos acts as an abstraction layer for the distributed network. By offering resources such as CPU and memory to distributed applications Mesos enables distributed applications to always have access to just the necessary amount of resources. The distributed applications which run on Mesos have to be customized as "Mesos frameworks". This is due to the fact that Mesos presents applications with resource offers from which applications can request resources. By using this model of being a scheduler of schedulers Mesos enables rapid creation of efficient distributed applications. The paper which details the principles that Mesos is built upon was first published in 2010. It has since seen adoption by many large organizations such as AirBnB and PayPal [32].

4.1 Manager-Agent model

Mesos uses a manager-agent model as the base architecture. Typically a Mesos cluster will have a couple of manager nodes. Among these manager nodes an active manager is chosen when a quorum has been reached. By having multiple managers (one active and several passive) Mesos enables fault tolerance in case of a manager failure. If a manager fails unexpectedly a new active manager is chosen among the passive ones. The active manager node is the central controller for all actions taken in Mesos. This manager controls multiple worker nodes called agents. These agents, upon start up, send details about themselves to the Mesos manager regarding their available resources. These resources can consist of CPU, memory, disk space and much more. The manager thus can keep track of the available resources in the cluster. Distributed applications in Mesos are customized as so called Mesos frameworks. Multiple such Mesos frameworks can be run concurrently. The man-

ager relays the currently available agent resources to these frameworks in the form of resource offers. The frameworks can then accept or decline these resource offers. If a framework accept such a resource offer or part of it that framework is able to run a job on the agent on which the resources exists. The frameworks can also communicate directly with their running tasks via message passing. Mesos tasks are stateful and allows the frameworks to keep track of their current state. A typical example of Mesos' scheduling flow is shown in Figure 4.1.

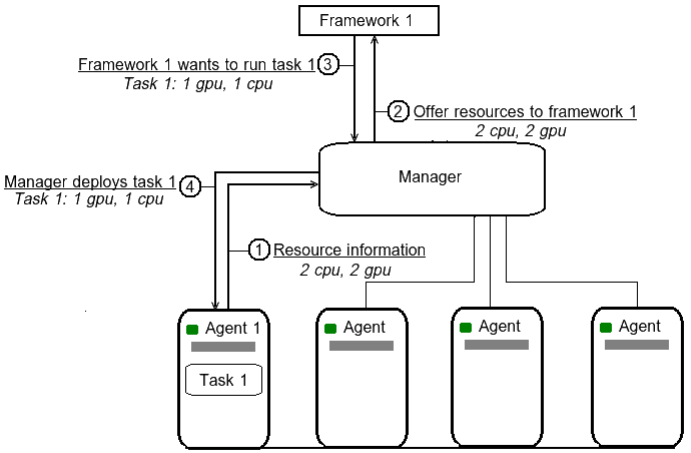

Figure 4.1 Resource utilization example

Scheduling example

1. *Agent 1 has 2 CPUs and 2 GPUs available. This resource information is sent to the manager node.*
2. *The manager uses a fair scheduling policy which determines that all the resources should be offered to framework 1.*
3. *This framework then decides it only need 1 CPU and 1 GPU to run task 1*
4. *The task is deployed in the agent 1. The remaining resource can now be offered to framework 2 since framework 1 does not need any more resources at this time.*

4.2 Managers

A cluster run by Mesos requires at least one manager to be able to assign tasks and relay resource offers. By using multiple managers, fault tolerance is increased allowing the system to be more robust since in the event of one manager crashing another manager can take

charge thus handle failures more gracefully. The manager typically handles all the logic while controlling the system and will make decisions regarding the scheduling of the system's resources. Since multiple Mesos frameworks can be set up in a single cluster system, this also allows different types of allocations to be done depending on the implementation of the manager. This allows Mesos to be very dynamic and configurable.

4.3 Agents

The agents are the workers in the context of Mesos. These workers register their resources at the manager and are assigned work, relayed by the manager, by the frameworks. The work model of the agents consists of "executors" and "tasks". An executor can have many tasks. An analogy to the concept of executors and tasks is a process (executor) and the process' work queue (tasks). The least complex frameworks assign one task to every executor.

4.4 Frameworks

The distributed applications which run on top of Mesos are called "Mesos frameworks". There are many such distributed applications which have been ported to Mesos. Mesos enables multiple such framework to run concurrently in the Mesos cluster. Frameworks enabling running other cluster managers on top of Mesos exists for Hadoop, Spark and many more. A list of the most popular frameworks is available on Mesos' website [33].

4.5 Resource offers

Resource offers are sent to the frameworks by the manager. These resource offers can in practice describe any type of resources. By default, however, they typically describe the CPU, memory and disk that an agent has available at the time. A list of such resources is offered to each framework in sequence. When a framework deploys new work based on the resource offer the resources used are subtracted from the resource offer list. Each list index describe the currently available resources of an agent in the Mesos cluster.

4.6 Message passing

Message passing in Mesos enables communication between the frameworks and their assigned agents. The protocol used to encode the messages in Mesos is Google's protobuf format [34]. The protobuf format makes it possible to encode messages that are, while less general than JSON or XML, smaller in size and faster to decode. This is done by pre-defining templates for each valid message. An example such a protobuf message and its usage in Java is displayed below

```
message Account {
    required int32 id = 1;
    optional string email = 2;
}
```

In the example above a message template defines the "Account" message type. The format defines templates that define attributes. Each attribute contains modifiers, a type, a name and are numbered. Messages may also contain child messages templates.

```
Account sampleAccount = Account.newBuilder()
    .setId(1)
    .setEmail("random@example.com")
    .build();
sampleAccount.writeTo(outputStream);
```

Google's protobufs provides bindings for most common languages. The above example displays how to build a message using a template written using protobuf's Java bindings.

The default behavior of messaging in Mesos is unreliable, i.e, when sending a message there is no guarantee that the message has been delivered to its intended recipient.

4.7 Fault tolerance

Fault tolerance in the case of a failing manager node is handled automatically in Mesos as long as there are a sufficient number of backup managers. Mesos always has one manager node which is the acting manager. This manager is selected when a quorum has been reached by the managers in the Mesos cluster. If a manager becomes unresponsive or crashes a new manager is elected and resumes the responsibilities of being the acting manager. In the case of failure of an agent, an agent's work detail has to be rescheduled to another agent which can complete the task previously delegated to the failed agent. This agent rescheduling needs to be performed by the Mesos framework.

4.8 Containerization

Executors and tasks run are isolated in Mesos using containers. When an agent is instructed to launch a new executor this executor is placed in an container by a containerizer. Mesos at the time of writing supports three types of containerizers: Composing, Docker and Mesos [35]. Mesos defaults to the Mesos containerizer which forks the original agent process blocking the child's execution until the child process has been isolated. In the case of Linux the resource usage isolation for the Mesos containerizer process is done using cgroups.

5

Nginx

Nginx [36] is a webserver, load balancer and reverse proxy. It was originally designed to be able to handle serving requests for web sites with a large workload. In contrast to other more resource intense web servers, Nginx uses an asynchronous event driven model whereby it has a number worker processes that accepts new connections, see Figure 5.1. Each one of these processes is able to handle multiple concurrent connections. Typically Nginx would use the same number of workers as the number of logical cores on the machine on which it runs. By not launching and tearing down threads or processes constantly Nginx saves a substantial amount of CPU-cycles and does not need a separate thread context for each connection or request. Another result of using this model instead of the traditional one-thread-per-connection model, is that Nginx has a substantially lower memory footprint.

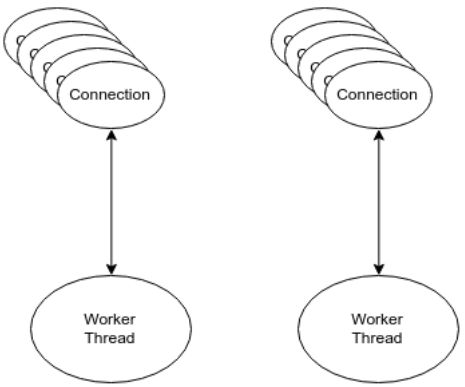

Figure 5.1 Nginx's worker model: each worker process handles multiple connections. By using this model as opposed to the one-thread-per-connection model Nginx is able to achieve a much lower memory footprint.

The more traditional worker model used by, for example Apache by default, uses one process per connection. This model, also known as the MPM worker model results in a separate process context needed for each connection. The overhead can thus be quite substantial. The MPM model is illustrated in Figure 5.2.

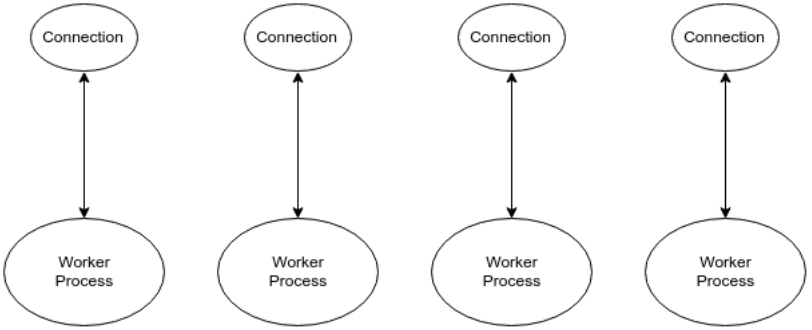

Figure 5.2 Apache's default MPM worker model uses a separate process context for each new connection. This results in an increased overhead per connection compared to Nginx's worker model.

5.1 Nginx Load Balancing

Load balancing is a first class member of Nginx. Due to Nginx's light weight worker model, its proven stability over time and its configurability, Nginx has become of the most ubiquitous load balancers. There are a wide range of protocols for which Nginx supports load balancing. The most common protocol for which Nginx's load balancing capabilities are typically employed is HTTP. Nginx is also capable of employing health checks on the servers to which it is routing requests. By employing these health checking capabilities, Nginx is able to gracefully handle individual server failures by not routing subsequent requests to the server which failed the health check.

5.2 Nginx configuration

Nginx is highly configurable system. Nginx uses configuration files to enable it to be a versatile system. A configuration file is made up of contexts and directives. Contexts are blocks delimited by { and } and define in which contexts a certain set of directives should be applied. Contexts can be nested. Directives define what should happen in a certain context.

```
# Autogenerated at: Mon Dec 05 10:29:48 EST 2016
server {
    listen 2020;
    location /api_a/ {
        proxy_pass http://API_A/;
    }
    location /api_b/ {
        proxy_pass http://API_B/;
    }
}
```

```
upstream API_A {
        least_conn;
    server server1:31018;
    server server2:31042;
}

upstream API_B {
        round_robin;
    server server3:31018;
    server server4:31042;
}
```

The above example configuration is an example of an Nginx configuration that would enable load balancing of two APIs. First the server context defines in what context the load balancing should take place. The directive listen sets the port for incoming requests to 2020. Next follows two locations that define what should happen when a request has a URL prefix of either /api_a/ or /api_b/. The proxy_pass directive in these location contexts forces the server to proxy these requests to API_A or API_B respectively. These two API server locations are defined by the upstream contexts. API_A defines two such back end servers that requests are to be passed on to by a work scheduling algorithm of least_conn. The least_conn scheduling algorithm selects which back end server to reverse proxy the request to by selecting the server with the least number of active connections. The API_B context also defines two back end servers but uses round_robin to vary the server selected to pass the request to. Round robin scheduling selects the next server by way of varying the servers circularly. For example requests would in the above case be passed in the following order: server3, server4, server3, server4 etc.

6

Problem definition

Cloud computing offers a great way to have access to virtually unlimited resources. This results in great flexibility when it comes to resource usage, allowing the user to decide how much resources are to be used at any given time.

Resources deployed statically must be deployed such that enough resources are always available to handle the peak workload or accept less availability during peak workloads. Due to the importance of not incurring down time and always being online this results in wasted resources that are idle most of the time which in turn leads to higher costs, see Figure 6.1.

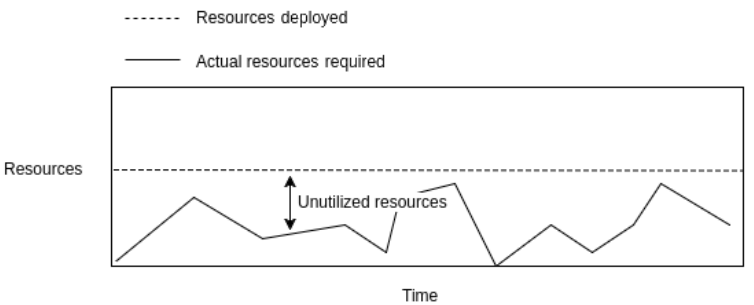

Figure 6.1 An example of a static deployment of resources. The amount of resources deployed are such that the system is capable of handling the peak workload. However, this leads to a lot of resources being unused most of the time which in turn leads to increased costs. Note that the amount of deployed resources is constant.

A better way to use the resources would be to deploy only the resources that are needed at a given time. To achieve this, a system where the resources can be deployed and torn down dynamically is very useful, see Figure 6.2. By using different scheduling algorithms, a system can greatly improve the resource usage.

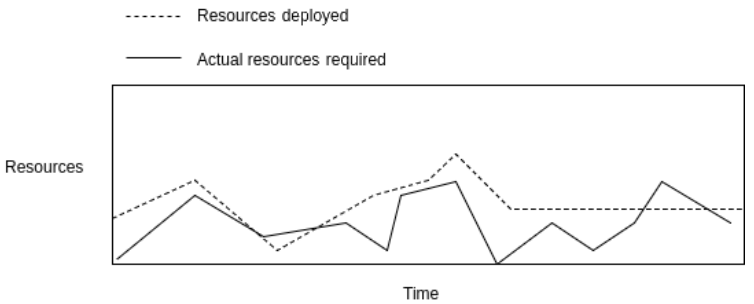

Figure 6.2 By dynamically deploying resources a system should be able to lower resource usage while still being capable of handling the peak load. This in turn should lower costs as the average resource usage should be decreased. Note how the amount of resources deployed vary with time.

A scheduling policy that only deploys exactly the amount of resources needed at any time is an optimal scheduling policy, see Figure 6.3. This is the ideal scheduling policy as it keeps resource usage at a minimum while incurring no loss in the quality of service.

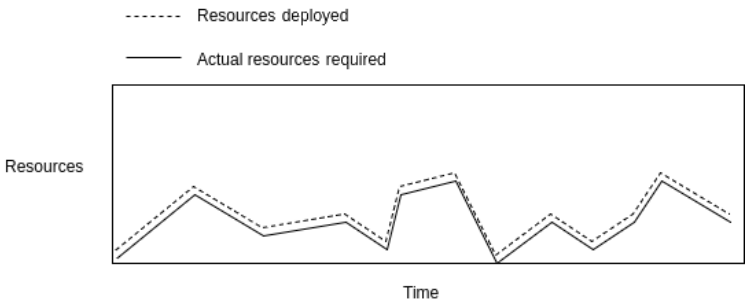

Figure 6.3 An optimal dynamic deployment of resources should deploy the exact amount of resources required for handling the load at that point in time. Note how the resources deployed and the resources required are identical at every point in time.

The services that are offered by a company often come with an assurance of a certain quality of service. If the specification of the quality of service is violated, the company usually will have to pay a predefined cost. This in cost in conjunction with the specified quality of service define the service level agreement (SLA) between the provider and the consumer. The resources that are required to avoid violating the SLA by the company represent another type of cost. These two costs together define a cost that can be optimized.

6.1 Concrete System and Services

The system that is examined in this thesis offers many different services through an application programming interface (API) hub where the client can call on an API to use the service that are offered. The API in the context of Data Ductus is served over HTTP via a predefined interface consisting of different URLs, see Figure 6.4.

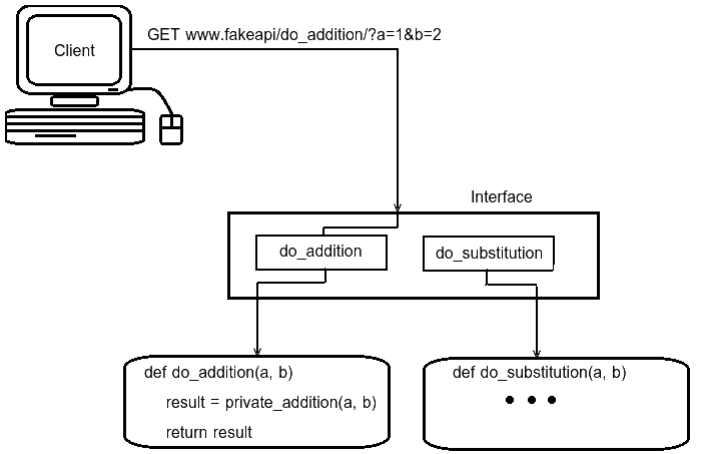

Figure 6.4 An API in the context of Data Ductus is served over HTTP. The API defines an interface of URLs which a client can use to request data and receive responses from.

Data Ductus offers a numbers of services that reside inside a data center. The system is composed of six virtual machines (VM), see Figure 6.5 for a more detailed overview. Each VM has a total of two gigabytes of RAM memory and two CPU cores. Each service is deployed to every VM with a specific resource allocation. This ensures all the services' ability to respond to the incoming workload. The resource allocation scheme is static and use up all existing resources inside the system no matter the workload. This leads to a very low resource utilization on average. The CPU utilization log from Data Ductus over the course of June 2016 shows that the utilization only averaged 2.7 percent of the total CPU resources that were available.

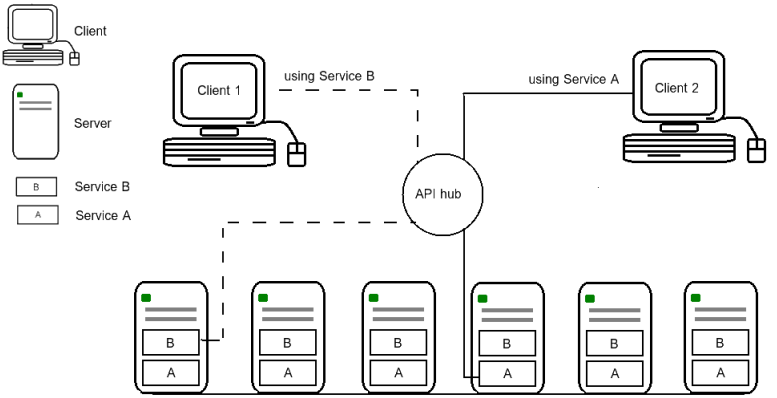

Figure 6.5 Data Ductus' system uses an API hub that routes the API requests and responses to their destinations. There are two APIs hosted by the system. One instance of both API A and B is running in all six virtual machines.

The primary resource bottleneck of the Data Ductus services is CPU resources. Each service has certain minimum resource requirements to be able to function correctly, i.e. the services require a certain amount of preallocated RAM and CPU to run and begin serving requests. The two services that reside in the data center are two APIs served over HTTP. Except for the initial static memory and CPU requirements, the resource requirements scale linearly with the number of active requests. The two APIs differ in their resource requirements. The most important properties of these services are the number of seconds it takes on average to start a new instance of a service, the number of seconds to tear down a service and the resource usage during the life-time of an incoming request to the service. The delay properties will directly affect how fast the system can respond to any change in the request rate.

6.2 Data Ductus' APIs

The two APIs that run in Data Ductus' data center are hereafter defined as A and B respectively. API A is an API for which the request rate is quite high on average. Each request typically consumes small amounts of resources and the response time is quite low when the system is not under a heavy work load. It takes a long time to launch a new such service instance due to it relying on certain external services during its upstart. However, after launching it relies very little on external services. In contrast, API B relies on few such services during its upstart while relying on more services during the processing of requests. The incoming request rate to API

B is typically quite low. Each request, however, takes more time to process than for API A. For a detailed comparison of API A and B, see Table 6.1.

Table 6.1 Description of DataDuctus APIs

Property	API A	API B
Avg. static memory	500 MB	300 MB
Start up time	60 s	20 s
Tear-down time	10 s	10 s
Avg. response time	800 ms	1500 ms
Min. response time	100 ms	500 ms
Max. response time	2000 ms	5000 ms
Dependence on third parties	Low	High
SLA violation	99%	99%
SLA max response time	5000 ms	8000 ms

The average static memory describes the RAM memory usage that is required to launch the service, i.e. the memory that is used irrespective of any incoming requests. The start up time of services is the time it takes from the service is initially launched until it is ready to process new requests. The tear-down time is the opposite of the start up time, i.e. the time it takes to shut down an active service. The average response time is the time it takes on average to process a request. However, the response times can vary a lot since different API calls may do very different computations. The minimum response time is the lowest recorded response time to that API under the Data Ductus static deployment. The maximum response time is the highest response time under the same. The dependence on third party described whether the API typically makes few or many external calls during its processing of a request. SLA violations describe the percentage of requests which must be within the SLA max response time. The SLA max response time in turn specify the response time that should not be violated.

7

Architecture

7.1 Experiment Design

The goal is to build a distributed framework that is able to dynamically utilize the available resources using different policies. With a scoring system that is based on how well the system performs under the same input, we will be able to look into the behavior and different attributes of the policies. To ensure fairness and correct behavior of the system, a service emulator was built to emulate the Data Ductus APIs and the baseline system's resource allocation. Data Ductus provided an access log of the incoming requests to their current system during 2016-05-31 - 2016-06-19. This log over response times was far too time consuming to replay exactly in real time. In order to use this data, the log data was compressed to exactly 60 times smaller. This meant a emulation via data replay over a period of 24 hours could be performed in 24 minutes. To measure the performance of the system, Data Ductus' original system (static deployment) was used as a baseline score. The score used was the resource usage inside the distributed system when replaying requests as specified by the access logs received from Data Ductus. The difference in score of the policy versus the baseline score reflects how good a policy is. A constraint on all strategies is that they must satisfy the SLA that was specified by Data Ductus.

7.1.1 System components

The system that was in use at Data Ductus consisted of six virtual machines servicing two HTTP APIs. The APIs' load was distributed to this system by way of an HTTP load balancer. The system that this project aimed to develop should mimic the original system to the extent possible, while also using dynamic scheduling of the APIs.

Since this thesis aimed to investigate possible resource usage optimization by using dynamic scaling, the scope of the project could be simplified by using a cluster manager already in existence as the core of the system. A cluster manager would help in allowing the scheduling strategies to ignore many of the lower level details of distributed systems, such as message passing and resource discovery.

A load balancer is useful for distributing the incoming HTTP requests since the services were to be deployed to multiple virtual and/or physical machines running the two APIs. Also, since the resource usage correlates with the number of incoming requests, there was a need to measure the number of incoming requests. The information regarding requests is either accessible from the load balancer (since it acts as the central gateway through which every request passes) or from the agents (since every requests is finally serviced by an agent after being routed by the load balancer). Motivated by it being simpler to retrieve this data from a central location, the load balancer was used to be the main point from which to retrieve historic and current request data. Hence, a requirement for the load balancer was that it had to have the current request rate and the response times of requests easily accessible.

7.1.2 Rationale in choosing particular system components

In choosing a cluster manager to use as the core on top of which the distributed framework was to be built, a number of key properties of the cluster manager were important. First, as mentioned in the previous section, the cluster manager should handle the low level details of the distributed system such as message passing and resource discovery. Secondly, the cluster manager should have an easily accessible API which the distributed framework would interface with. Thirdly, the cluster manager should preferably be widely used and have a large developer community since that would simplify the search for information about the cluster manager. Mesos [13] fulfilled all these attributes and was used as the cluster manager upon which the distributed framework was built.

The HTTP load balancer that was to be used also had to fulfill certain requirements. The main requirement was that it should allow retrieval of the current request rate and preferably more specific information regarding requests and responses. Two load balancers that are in widespread use are Nginx [36] and HAProxy [37]. Both of these load balancers fulfilled the requirements above. Nginx was used due to the authors' pre-existing familiarity with that system.

7.1.3 System components trade-offs

Mesos is a lightweight cluster manager and enables the creation of highly customizable distributed frameworks that run on top of it. It therefore fits perfectly with purpose of the thesis.

As Nginx was used as the system's load balancer, the access logs from Nginx [38] were used to retrieve the current request rate and more detailed information regarding individual requests. Nginx allows for disabling buffering of the access logs and immediate flushing. However, since Nginx logs requests when a request has been served with a response this may introduce a lag effect in the system. The current request rate measured by the distributed framework is thus unable to detect incomplete active requests that have not been served with a response.

7.1.4 Motivation for the design of the distributed framework

Mesos enforces that the distributed frameworks that run on top of it follows the manager-agent pattern. Therefore, the distributed framework naturally follows the same pattern. The manager keeps track of all information regarding the state of agents in this pattern and this lead to a "fat" manager scheduler and a quite "slim" agent task executor. The manager acts as the central point of the system and to reduce the complexity of the system, updates to the routing tables in the load balancer are handled solely by the manager.

7.1.5 Evaluation

The framework is working correctly if it can react to the incoming requests to a certain service. This reaction will be decided by the framework's control policy, different strategies will produce different results. This means that if the framework result is to be evaluated, a consistency of the system's behavior needs to be achieved. The consistency therefore will depend on the behavior of two external components which is the incoming requests, and the service that responds to these requests.

Data Ductus provided data concerning two services during June 2016, specifically the properties of each service and their corresponding access log. The most important information that was used in emulating the request load via data replay were the timestamp denoting the time at which a request was made to the system as well as the URL that the request targeted.

Two mock services were created to emulate the Data Ductus services. The important aspects of a service include starting time, the resource load and the tear down time. These mock services provided a consistency of output given the same input. The client emulator was created to feed the Data Ductus the workload input, extracted from Data Ductus' access logs, to the services, by sending requests to the framework.

The requests produced by the client emulator is dependent on the access log which can be produced with different specific patterns to give insight about how the policies and the framework react. An example would be how a policy reacts to a spike in the request rate.

The goal of the framework is to lower the resource usage while still maintaining a certain SLA. Thus, the measurement of success is the average resource usage over time. Since Data Ductus had an original control policy (static deployment), this was used as the baseline policy. The Data Ductus access logs are then used as the baseline input to each policy and the produced result is compared to the result of the Data Ductus policy.

In order to produce enough interesting test results, an automatic testing framework was created. Every test starts a vector of predefined parameters, these parameters define the state of the system which includes the states of the services and the states of the control policy. The test then goes through each data input set and produces the corresponding result. The evaluation data contained in the result of a

test run include the average resource usage over time and the number of times the response time violated the max response as specified in the SLA. The testing framework also produced plots visualizing the number of requests over time, the response time of each request over time and the resource usage over time.

The access log data from Data Ductus varied widely in the average request rate from day to day. Because of the time constraint, three dates were used as the validation input data, the lightest workload, a day deemed to have an "average" workload and the day with the most request intense workload.

7.2 System Design

The Framework The framework was built on top of Mesos APIs. The Mesos framework separates the logic layer and the resource layer. Mesos acts as a resource layer where each agent in the system provides the resources that they can offer to the framework. The framework is built upon this layer to be the logic provider.

Nginx was used to distribute the incoming workload to the Mesos agents, i.e. Nginx was employed as a load balancer in the system.

Java was chosen to be used to implement the client emulator.

Services and service instances A service in the context of the framework is a self-contained HTTP-server which acts as an HTTP API. The service can be configured to emulate different characteristics. For example, a service can be configured to always use a certain amount of RAM for a period of time on every incoming request. This was used to configure the services to emulate the Data Ductus APIs. A service instance is a deployed instance of a service which runs on an agent. There may therefore exists many such service instances of the same service type running concurrently.

Test Framework Python was chosen as the language used to analyze and evaluate the measurement data due to its extensive and ease-of-use libraries for working with data. To simplify the data analysis and the generation of plots, Matplotlib [39] was used.

7.3 Assumptions

These are the assumptions that were made to constrain the scope of the thesis.

7.3.1 Services and service instances

- All service instances of the same service type are considered to be identical to each other.
- Different services may use different amounts of resources.

- All requests made to the same service type are considered to have similar resource usage.
- All service instances use HTTP to transmit information.
- All service instances are stateless locally, i.e. they do not use persistent storage locally. However, services are not necessarily stateless, they may store persistent data on remote machines.
- All services must be able to be deployed independently as micro services.
- Minimum resource requirements for a service instance are less than the total resources of any agent.
- Service instance performance is only limited by the supported resource types (CPU, memory, ports).

7.3.2 Resources

- Supported resources are CPU, memory and ports. No disk.
- The framework has access to unlimited resources.
- All resources of the same type have the same performance, i.e. the CPU of one agent is assumed to have the same performance as the CPU of any other agent.

7.4 System Architecture

The system uses a typical manager-agent architecture, see Figure 7.1. One manager controls many agents. The manager deploys as many service instances onto the agents as necessary. The load balancer sits in front of the agents and routes incoming requests to the agents' service instances. The manager is in charge of configuring the load balancer's routing table when deploying or canceling a service instance.

7.4.1 Manager

The manager consists of two main components: a controller and a sensor, see Figure 7.2. The controller maintains a central state of the entire system, and makes a decision regarding how to schedule the services to the agents based on a policy. The state of the system is comprised of two parts.

- The external incoming workload, for example, how many requests per second, what kind of request, and to what service. The sensor communicates directly to the load balancer (Nginx) and receives information regarding the response times of the services.

- The internal system, how many instances are online, what resources are available. This information is available through Mesos and the agents.

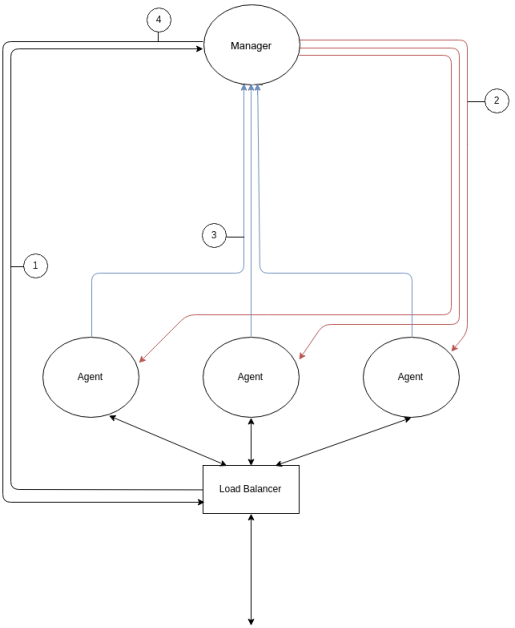

Figure 7.1 System Architecture Overview

1. *The manager retrieves profiling information from the load balancer. This profiling information primarily consists of the response times to serve the HTTP requests to the services that is run by the framework.*
2. *The manager instructs the agents to deploy (or tear down) service instances.*
3. *After the agents have deployed (or torn down) the service instances they send back the deployment information to the manager.*
4. *The manager uses the received deployment information to update the load balancer's routing table.*

7.4.2 Agent

An agent, as in Figure 7.3, consists of two components: an executor and task handlers. The executor takes orders from the manages through the Mesos framework to handle starting and terminating service instances. Service instances are wrapped by a task handler. Every time a service instance is started, a corresponding task handler will be created to handle the details of the service instance, which include staging, starting and terminating the service instance. The task handler will also communicate changes of the service instance state to the manager.

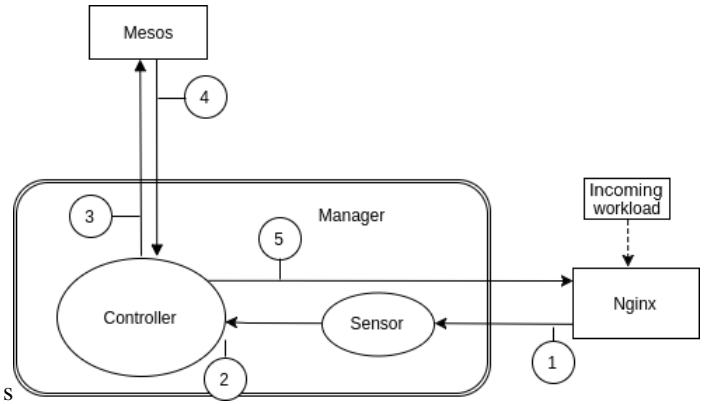

Figure 7.2 Manager Overview

1. *The sensor retrieves the log of requests from the load balancer.*
2. *The sensor updates the controller with new information.*
3. *The controller deploys or removes service instances based on the control policy.*
4. *Mesos updates the controller about the currently deployed service instances.*
5. *The controller updates the load balancer's routing table and configuration.*

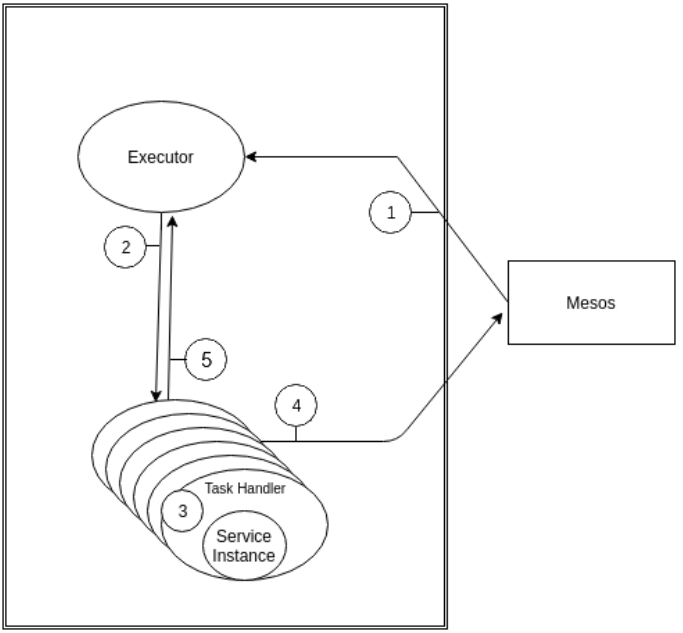

Figure 7.3 Agent Overview

Scheduling new service instances

1. *Mesos sends information to the executor to deploy new service instances.*
2. *The executor deploys task handlers which run the service instances.*
3. *The task handler starts the service instance.*
4. *The task handler updates deployment information to Mesos.*
5. *The task handler updates deployment information to the executor.*

7.4.3 Service Descriptors

The service descriptors are configuration files that contain information regarding the service and their corresponding SLAs. The service descriptors contain the following information.

- The service instance's minimum resource requirements.
- The service's unique identifier.
- The service's relative URL path.
- The service's executable file.
- The command to execute.

7.5 The Test Framework

7.5.1 Client Emulator

Different request patterns will lead to different responses from the system. In order to measure the framework responses with different policies against the same pattern of incoming requests, a client emulator is required. The client emulator is built such that it is able to take input data in the form of a timestamp and a URL denoting that a request should be sent at that time to the specified URL. It next sends one request at the time of each of the timestamps to the specified URL, see Figure 7.4.

7.5.2 Service instance emulator

The service instance emulator emulates a Data Ductus API. The emulator can have different CPU and memory requirements which are listed below.

- Passive memory: The memory resources that service instances use in the absence of requests.
- Active memory: The memory resources that service instances use per active request.
- Active CPU: The amount of CPU that a request uses for every active request.
- Active time: The amount of time it should take for one request to complete given that no other requests are competing for resources.

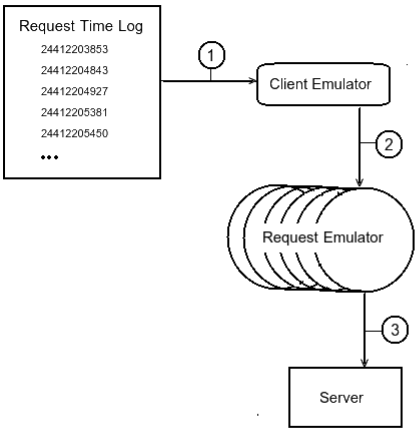

Figure 7.4 The client emulator.

1. *The client emulator reads the input request log.*
2. *The client emulator creates a new request emulator for each line in the input request log and instructs them to send a request to an URL at the defined time.*
3. *The request emulators send requests to the server at their specified times. The requests complete by either receiving a response or timing out after 60 seconds.*

7.6 Policies

To maximize resource utilization and avoid violating the the Data Ductus SLA, several different scheduling policies were created. Some of these policies can be added "on top" of any other scheduling policy, these policies are described as generic policies. These generic policies are used to modify the existing scheduling policies by adding a buffer of instance deployments and/or waiting before tearing down service instances.

7.6.1 Baseline Policy

The policies will be used to establish a baseline by which we can measure the performance characteristics for other policies. The policies will be used without any modifications or strategies.

Static policy This is the policy that Data Ductus is currently using. Each service will be deployed at all times in every virtual machine. This policy guarantees SLAs but is also the most resource expensive policy of the framework.

Perfect Policy This is the policy that we wish to achieve. In theory, a perfect policy will guarantee the SLAs while also using the minimum amount of resources required. This could be achieved if the policy had perfect knowledge of everything in the system and also knew the future and therefore could deploy the exact amount of resources needed and nothing more at any given time.

7.6.2 Generic Policy Improvements

The generic policy improvements are the possible improvements that can be applied to any of the implemented policies. Each of these improvements were intended to solve a specific pattern that can arise in the input data. The reason why these are not deemed standalone policies is that they only change certain attributes of the policy on top of which they run. For example, they may be added on top of another policy thereby modifying it such that a wait-before-tear-down time is introduced.

Policy with Space Buffer When the input data has many peaks in the requests rate, the time it requires to react to such an increasing request rate will be slow. In order to gain enough time to prepare such increasing demands of resources, the space buffer improvement can be added. This generic policy takes the output scaling deployment from the base policy it is running on top of and multiplies the number of instances to be deployed by a factor k_{space}, i.e., extra service instances are added for every new deployment. The number of deployed instances n_{space_t} given a scheduling policy p that deploy n_t instances with a space buffer k_{space} is equal to:

$$n_{space_t} = n_t + k_{space}.n_t, \text{where } k_{space} \geq 0$$

A policy using a space buffer will thus inevitably use more resources than the underlying policy.

Policy with Time Buffer In this policy a wait-before-tear-down time is used. Instead of simply looking at the current output value from the underlying scheduling policy, this generic policy looks at the k_{time} previous output values from the scheduling policy and uses the maximum number of deployed instances. This should prevent instances from scaling down to quickly due to a temporary drop in the request rate. The number of deployed instances n_{time_t} given a scheduling policy p with a time buffer variable k_{time} at time t is in this case defined as follows:

$$n_{time_t} = max(n_{t-k_{time}}, n_{t-k_{time}+1}, n_{t-k_{time}+2}, ..., n_{t-1}),\ k_{time} \geq 1$$

A time buffer will, just as the space buffer, also inevitably use more resources than the bare policy. However, a policy with a time buffer added on top is expected to prevent short term oscillations in the number of deployed instances. The scenario which the time buffer is expected to be a solution for is when a service instance is no longer required and a tear down of that instance is requested, however, the decline in the request rate was only temporary and a new service instance needs to be deployed even before the last one has completed its tear down process.

7.6.3 Implemented Policies

A bare implemented policy p is one with no generic modifications added on top. That is, the policy is used as is. A policy p at the deployment time t will deploy n_t number of instances as a response to the incoming response time r_t. At any given time, the minimum of deploy instances n_t for each service offer inside the system is one.

$$n_t = p(r_t),\ n_t > 0$$

Naive Reactive The naive reactive policy checks if any of the latest incoming requests has violated the max response time constraint as defined in the SLA. In the event of the constraint being violated, the policy will deploy one more service instance in response. The deployment of new services will continue until the constraint is satisfied. On the other hand, if the constraint is satisfied, the policy will tear down a service instead.

The policy p at the time t for a workload that creates response time r_t, where the maximum allowed response time is r_m (the maximum response time as defined by the SLA) is formally defined below.

$$n_t = \begin{cases} n_{t-1} + 1, & \text{if } r_t > r_m \\ n_{t-1} - 1, & \text{otherwise} \end{cases}$$

Proportional error policy Proportional error (PE) policy looks at the difference between the maximum response time and the current response time where the difference is direct proportional to the number of service instances that are to be deployed.

The policy p at the time t for a workload that creates response time r_t, where the maximum allowed response time is r_m (the maximum response time as defined by the SLA) is formally defined below.

$$n_t = n_{t-1} + (r_m - r_t) * k_{pe},\ k_{pe} > 0$$

where k_{pe} can be chosen arbitrarily.

Pre-scheduled Previous policies react to the response time only when the SLA's maximum response time has already been violated. This results in that a certain number of responses always will exceed the SLA's maximum response time. Furthermore, since there is always a delay between the detection of response time violations and the subsequent deployment more responses than the initial one are likely to be in violation. In order to prevent such behavior, an earlier reaction to the workload is preferred. Instead of reacting to the SLA's maximum response time, a lower limit of response time would likely produce better behavior. In this policy the action of deploying new service instances is performed when the average response time exceeds some percentage of the SLA's maximum response time.

The policy p at the time t for a workload that produces an average response time of r_a is formally defined below. The maximum response time as defined by the SLA is denoted r_m.

$$n_t = \begin{cases} n_{t-1} + 1, & \text{if } r_a > r_m * k_{pre} \\ n_{t-1} - 1, & \text{otherwise} \end{cases}$$

where the average response time r_a is calculated as the mean of response time between deployment time t and $t-1$ and k_{pre} is a value between 0 and 1.

8

Lab Implementation

8.1 Cluster setup

The lab implementation was done by trying to emulate the real-world conditions at Data Ductus to the extent possible. A Mesos manager was deployed on a virtual machine. We chose to only run one manager to reduce the costs of doing the testing and since we were not planning on testing the fault tolerance of the Mesos managers. Six agent nodes each running on a virtual machine were also deployed. These six virtual machines should emulate the set up currently existing at Data Ductus as well as possible. All virtual machines had 2 GB RAM and 2 CPU cores, i.e., they had the same properties as those at Data Ductus. The most significant performance difference that likely exists between the cluster setup in this thesis and the original cluster setup at Data Ductus is the performance of the individual CPU-cores. The virtual machines used in the lab setup were droplets from DigitalOcean [1] on the AMS2 data center region (Amsterdam). The operating system used by the virtual machines was Ubuntu 14.04.4 x64 [40]. The version of Mesos running on these virtual machines was 1.0.0. Since the resources of the Mesos manager's virtual machine exceeded the requirements for running the manager, we decided to run the load balancer (Nginx [36]) in the same virtual machine. By running Nginx in the same virtual machine as the Mesos manager, the fetching of the Nginx access log data from by the Mesos framework was significantly simplified: the Mesos framework only needed to read a local file and parse it.

8.2 Cluster configuration

The Mesos manager and agents were configured such that they were connected to each other, i.e., such that the Mesos manager could deploy tasks running service instances to the agents. Nginx's output to the access log was configured such that it only logged requests' destination URL and the response time of the requests in milliseconds, this was the only log data needed by the policies to determine the number of instances that were to be deployed.

8.3 Emulation of Data Ductus APIs and Evaluation of Policies via Data Replay

To simplify multiple runs of the policy tests, all actions, except for the initial cluster configuration described above, were automated via shell scripts. The actions that these scripts took on each test run is described below. All these steps were repeated for each of the different workloads (light, medium, heavy) for each of the policies and APIs.

8.3.1 Starting the Mesos Framework

The initial action taken by the scripts was to restart any currently running Mesos manager or agents. This was done as a way to make sure that there were no concurrently running Mesos frameworks that would degrade the performance of the tests. The access log of Nginx was also cleared so as to make it easier to analyze the response times later. Next, the scripts started the Mesos framework. The Mesos framework was feed a service descriptor, which the deployed services used to emulate a Data Ductus API (either emulating API A or B, described in Table 6.1). Initially, six service instances emulating an API were deployed. The reason for deploying six service instances at startup was that not deploying instances at startup resulted in a high number of SLA violations initially. The reason for these initial SLA violation was that the policy did not have access to any access log data to determine the number of service instances that should be deployed initially. Six service instances (half of the maximum number of service instances) was a reasonable starting point so that there were not a large amount of SLA violations nor to much resource usage before the policy had any data to base its deployment decisions on. Each service instance deployed was allocated 1.5x the average static memory of that API, see Table 6.1, which was sufficient for handling a large number of requests concurrently. Furthermore, each deployed service instance was allocated 1 CPU core.

8.3.2 Replaying Data Ductus' Access Log

After starting Mesos and the Mesos framework, the scripts started the data replay. The data that was replayed was the data for either API A or B and the corresponding light, medium or heavy workload. The replay was run locally from computers in Lund using the client emulator, see Figure 7.4, sending requests at the times specified in the Data Ductus access logs to the load balancer's URL. When a response to a sent request was received, its response time as well as the time the request was sent was recorded for later use during the analysis. The Mesos framework automatically scaled the number of deployed instances based on its policy as it received and responded to requests. The Mesos framework would poll the currently running policy every 1 second to determine the number of instances that should be deployed or torn down. Using the output from the policy our Mesos framework would de-

ploy/tear down the specified number of service instances as tasks to the agents, see Figure 7.2. Every time the Mesos framework scheduled service instances to be deployed or torn down it recorded the number currently running service instances as well as the current time in a log.

8.3.3 Parsing, Analysis and Plot Generation

When the data replay scripts finished, the scripts downloaded the log of containing the number services deployed over time. This data was automatically combined with the Data Ductus access log as well as the log of response times of every request sent to produce the figures and tables that were used in the Chapter 9, Results.

9

Results

The results were obtained by using different request input data sets representing different workloads. The sets used were a heavy workload, a medium workload and a light workload. These workloads are such that the "heavy workload" emulates via data replay the day with the highest number of incoming requests, the "medium workload" emulates an "average" day and the "light workload" emulates the day with the lowest number of requests. The emulations via data replay were performed for both API A and API B, see Table 9.1 for the specifications of the two APIs. The incoming request rate for API B was very low and as such resulted in a single deployed service node being capable of handling all incoming requests. Several of the policies had variables, e.g. k_{pe} and k_{time}, that could be adjusted. Several values for these variables were tested; we selected a few interesting such values for further analysis below.

Table 9.1 Description of Data Ductus APIs, same as Table 6.1. The average/min/-max response times are for the baseline policy, i.e., the static deployment currently in use at Data Ductus.

Property	API A	API B
Avg. static memory	500 MB	300 MB
Start up time	60 s	20 s
Tear-down time	10 s	10 s
Avg. response time	800 ms	1500 ms
Min. response time	100 ms	500 ms
Max. response time	2000 ms	5000 ms
Dependence on third parties	Low	High
SLA violation	99%	99%
SLA max response time	5000 ms	8000 ms

9.1 Workload Result

The tables below shows each policy as a row. The first column is the name of the policy at that row. The second column shows the average resource usage during the emulation via data replay, i.e. the average number of used CPUs during the testing. The third column shows the percentage of requests which had a response within the SLA's maximum response time. Column two and three are divided into two sub columns. The first of these two sub columns shows the results when the deployment delay (i.e. the time it takes to start a new instance) is 1 second with compressed emulation (API A). The second sub column shows the results for when this delay is increased by 5x for API A (API Ax5). Since the both the medium and light workload yielded the very similar results these are displayed in the same table, see Table 9.2.

Table 9.2 Resource usage and number of max response time violations when handling a medium workload

Medium Workload	Configuration	CPU Usage		Responses in time	
Service		API A	API Ax5	API A	API Ax5
Baseline	-	12	12	100%	100%
Naive Reactive	-	1	1	100%	100%
Naive Reactive with Space Buffer	$k_{space} = 1$	2	2	100%	100%
Naive Reactive with Time Buffer	$k_{time} = 5$	1	1	100%	100%
Pre-scheduled	$k_{pre} = 0.3$	1.1	1.4	100%	100%
Pre-scheduled with Space Buffer	$k_{pre} = 0.3$ $k_{space} = 1$	3.2	4.4	100%	100%
Pre-scheduled with Time Buffer	$k_{pre} = 0.3$ $k_{time} = 5$	6.5	3.7	100%	100%
PE	$k_{pe} = 0.001$	1	1	100%	100%
PE with Space Buffer	$k_{pe} = 0.001$ $k_{space} = 1$	2	2	100%	100%
PE with Time Buffer	$k_{pe} = 0.001$ $k_{time} = 5$	1	1	100%	100%

9.2 Heavy workload results

Certain interesting test runs are visualized below. The figures for each such interesting test run is subdivided into 3 subplots. All figures below use heavy workload input data set and are for the emulation of API A (without any artificial delay). The heavy workload input data set provides the most interesting results, see Table 9.3. This is due to the request rate in both the medium and the light data sets being very

low and therefore even with the minimum number of service instances, which is one, the framework and all policies were able to handle the workload, as seen in Table 9.2.

Table 9.3 Resource usage and number of max response time violations when handling a heavy workload

Heavy Workload	Configuration	CPU Usage		Responses in time	
Service		API A	API Ax5	API A	API Ax5
Baseline	-	12	12	100%	100%
Naive Reactive	-	1.6	1.87	50.8%	48.2%
Naive Reactive with Space Buffer	$k_{space} = 1$	3.78	5.1	51%	53.9%
Naive Reactive with Time Buffer	$k_{time} = 5$	2.50	2.96	69.6%	72.1%
Pre-scheduled	$k_{pre} = 0.3$	8.5	7.8	97,7%	99.94%
Pre-scheduled with Space Buffer	$k_{pre} = 0.3$ $k_{space} = 1$	10.8	11.0	100%	100%
Pre-scheduled with Time Buffer	$k_{pre} = 0.3$ $k_{time} = 5$	11.8	11.8	100%	100%
PE	$k_{pe} = 0.001$	4.77	5.78	61.6%	58.2%
PE with Space Buffer	$k_{pe} = 0.001$ $k_{space} = 1$	6.30	7.08	68.3%	65.0%
PE with Time Buffer	$k_{pe} = 0.001$ $k_{time} = 5$	4.58	5.09	76.0%	75.0%

The first plot (example shown in Figure 9.1) in the figures shows the incoming request rate averaged over 20 emulated data replay seconds to visualize trends over time. This plot is the same for every emulation of the same API and the same workload.

The second plot (an example is shown in Figure 9.2) visualizes the response times of each request over time. The response times on the Y-axis are denoted in milliseconds. The SLA's max response time is displayed via the yellow line. In some cases there are no displayed responses (see for example the gaps in the plot below between times 14:00-24:00). These gaps are a result of request failures. These request failures occur due to a request timing out if it has not received a response within 60 seconds.

The third plot (an example is shown in Figure 9.3) shows the number of CPU cores used over time. This is the measurement of resource usage over time and shows the scaling pattern of the policy used.

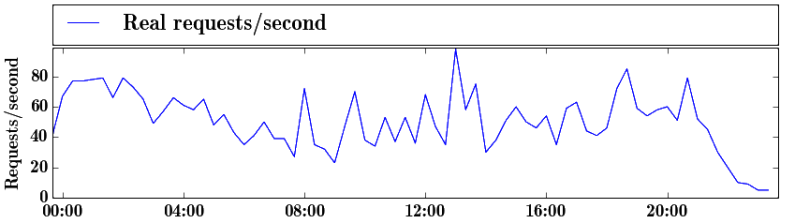

Figure 9.1 First plot, request rate per second over time.

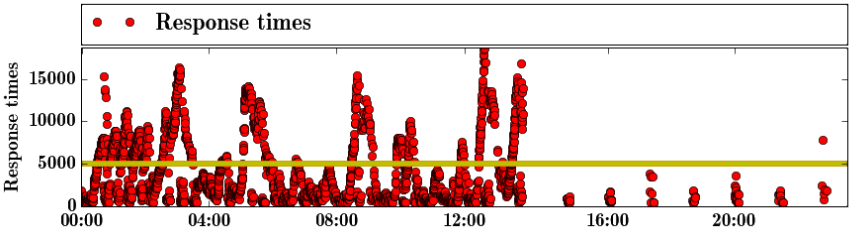

Figure 9.2 Second plot, requests' response times over time.

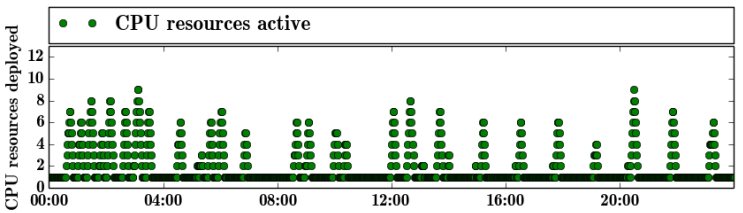

Figure 9.3 Third plot, deployed CPU resources over time

9.2.1 Naive reactive policy

The naive reactive policy is a slow reacting policy. If a very heavy workload is introduced, then the response of the policy increases the number of service instances one at a time. This means that a large lag effect is present and the workload queue stalls until enough resources have been deployed. This behavior is visible in Figure 9.4

where multiple spikes in the response times are visible. This is due to the system not expecting such an increase between the workload and therefore creates a large gap between resources in use and the resources actually required. The policy therefore takes a long time to deploy the resources required and scale in response, i.e., it is late in its response. In comparison to the baseline policy, the naive reactive policy

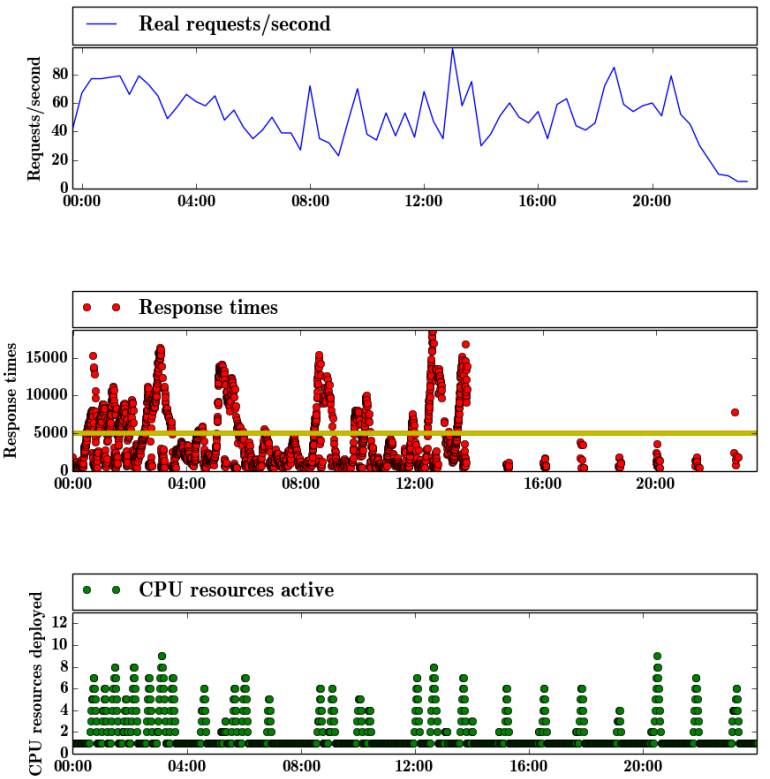

Figure 9.4 Naive reactive, heavy workload. Note the gaps in the response times plot during 14.00 - 24.00.

still shows an improvement in resource usage, see Table 9.3. However, the naive reactive policy violates the SLA's max response time half the time due to it being slow in increasing the number of deployed instances. The naive reactive policy with a space buffer barely increases the number of responses that are in time while increasing the resource usage. The reason why the naive policy with a space buffer

performs poorly is due to it having the attributes of the basic naive reactive policy while deploying twice as many service instances, see Figure 9.5. Since it deploys twice as many service instances during each polling interval when a max response time violation has been detected, one might assume that the space buffer naive reactive policy would perform better. However, due to the deployment delay these extra service instances were rarely available until the workload had already decreased. Therefore the CPU load was typically very high for one of the instances while being very low for all of the other deployed of service instances for this policy. During some peak workloads the request rate was such that requests began timing out due to the system not being able to scale fast enough, see the gaps in the response times plot between 14:00 - 24:00 in Figure 9.4.

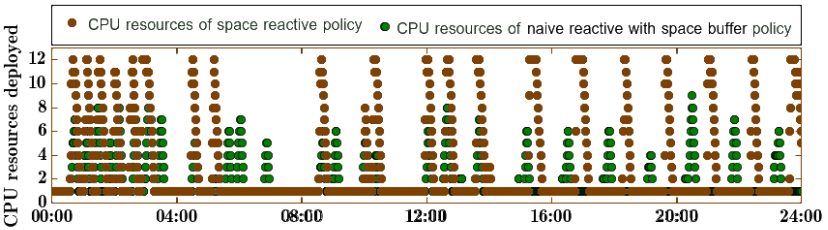

Figure 9.5 Deployed CPU resources over time of the naive reactive and naive reactive with space buffer policies

9.2.2 Proportional error policy (PE)

The policy that bases its scheduling on the proportional error shows improvements over the purely reactive policy in its ability to maintain the SLA. This is likely due to two factors: faster scaling and higher average resource usage. The policy increases the number of deployed instances faster which enables it to better handle bursts in the request rate. This faster scaling also makes the policy use more resources overall than the naive reactive policy. The best performing version of the PE policy is the one with a time buffer, i.e, it does not immediately tear down services instances once the workload decreases. This results in it better being able to handle the variability in the workload. The version of the PE policy that had the best results was the one that included a delay in the tearing down of service instances, see Figure 9.6. By using this delay factor the constant scaling up and scaling down due to small spikes in the request rate is reduced. Therefore the policy is better able to handle workload increases over longer periods of time that may have high variability in the short term.

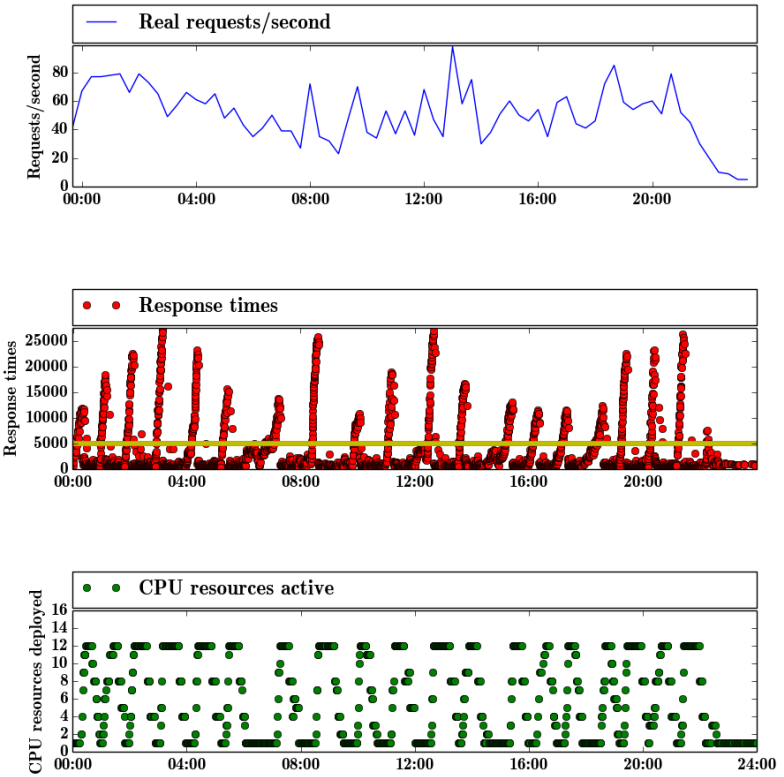

Figure 9.6 Time PE, heavy workload. Note the delay before the number of CPUs are torn down after a spike in the response time.

9.2.3 Pre-scheduled reactive policy

Both the previous policies (naive reactive and PE) show an improvement in the average resource usage but both violate the SLA. This is due to the naive reactive and proportional error policies deploy new service instances only when the maximum response time has already been violated. The pre-scheduled policy with the parameters chosen prioritizes maintaining SLA over resource usage. Since maintaining SLA in this case has a higher priority than the resource usage, this means that the resource usage increases a lot.

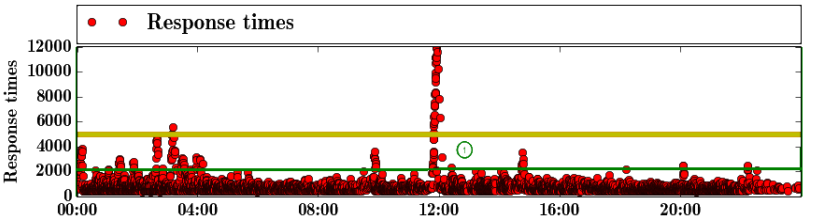

Figure 9.7 Pre-scheduled, heavy workload. The yellow line represents the SLA's maximum response time. Most of the responses are below the green line which indicates possibilities in lowering the resource usage.

However, the policy is able maintain the the SLA when used with a time buffer or space buffer. It uses nearly the same amount of resources as the baseline. As is visible in Figure 9.7, most of the response times are below 2000 milliseconds which is quite a bit below the the SLA's maximum response time. This indicates that even though this policy is the policy with the best results in terms of maintaining the SLA's maximum response time, there is room for improvement with regards to resource utilization.

As is visible in Figure 9.8 the average resource usage is consistently very high. Furthermore, most of the time when a large number service instances are torn down the response times increases significantly. This is most clearly visible at times 09:30, 12:00 and 15:00 in Figure 9.8. One anomaly in the results in Table 9.3 is the difference in the results for API A (97.7% responses in time) and API Ax5 (99.94% responses in time) for the bare bones pre-scheduled policy under a heavy workload. This is due to API Ax5 not experiencing the spike which is visible in Figure 9.8. When there is a response time violation, more are likely to follow. As is shown in Figure 9.8, most of the time, the number of CPU resources deployed are close to 12. The reason for the Ax5 not experiencing the same spike in the response time is because it still had 12 CPU resources active at the time of the spike at 12:00 at which time the request rate spike is most significant.

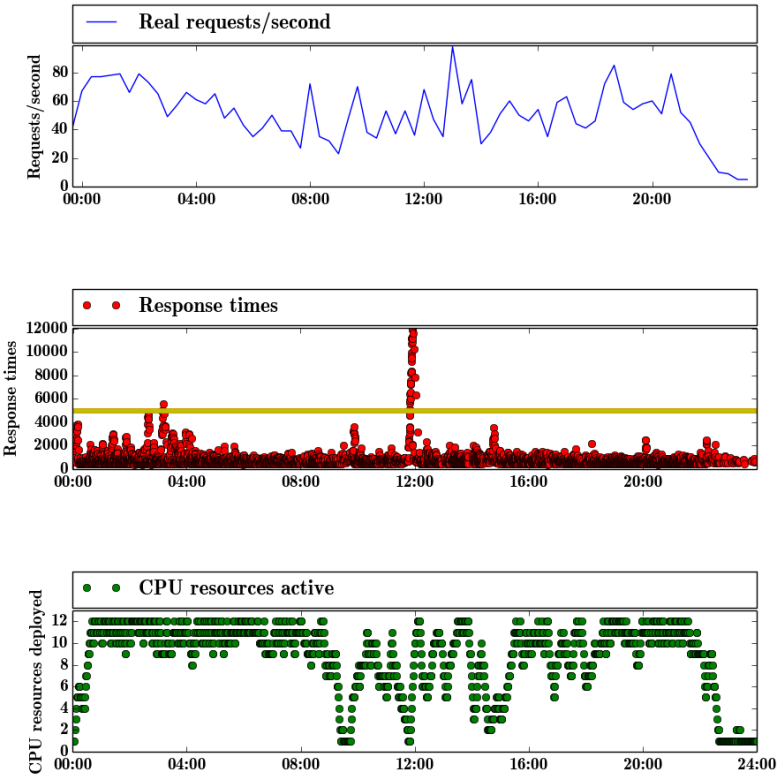

Figure 9.8 Pre-scheduled, heavy workload. Note that the vast majority of response times are below 2000 ms. There is a large amount of SLA max response time violations that occurs at 12:00. This is at the time during which the most significant spike in the request rate occurs. Due to the decline in the request rate preceding this spike, the number of deployed instances is only one when the spike starts which leads to a large number of SLA max response time violations. In contrast, the tests run for the Ax5 API in which a artificial deployment delay has been introduced does not experience the same SLA violations due to instances coming online 5 seconds later, i.e., at the time API A without a deployment delay starts tearing down service instances just before the spike in the request rate occurs.

10

Discussion

The system's performance is measured by the resource utilization. There are three actions that the system use in response to a changing request rate: scaling up, scaling down or keeping the resource usage constant. Note that these three actions and their corresponding effect on the system is different from each other.

The system scaling up is the action that prevents the maximum response time in the SLA from being violated. If the response scales less than is required by the current workload, a queue of work items will be created and the max response time will be continuously violated until enough resources have become active. Policies that deploy too much resources in response to an increasing request rate will not utilize resources optimally and waste resources. The solution is to find a good balance between the number of deployed instances, where the response can be weighted in relation to the user's need: whether the SLA's or the resource usage is more important.

The scaling down action is used to prevent resources being used without them performing any work. If the workload is decreasing, unnecessary resources should be deallocated, though depending on the incoming request pattern, scaling down immediately once the workload is decreasing may lead to problems. For example, if the workload is in a temporary decrease but then increases back to the same workload as before. If the deployment time is very long, the system would not be able to respond with enough resources fast enough and this would result in violated max response times.

10.1 About Deployment Time

Service instance deployment time is a very important factor, if not the most important in the system's behavior. The faster the service instances can be deployed, the faster the system can react to an increasing workload. In a perfect system, where there is no delay, the response to the workload should always be exactly the needed resources. When examining the framework, there are multiple delays in the system: such as the load balancer information, the message passing in the system, etc. But

the deployment time has the longest delay and this makes it a key factor in how well the system reacts.

Testing the framework with different deployment times is important to see how well the system behaves under different conditions. We can clearly see that all except one policy perform worse with an increased delay. In Table 9.3 we can see that some policies, such as the naive reactive with space buffer policy performs significantly worse where the average resource usage goes from 3.78 under A to 5.1 CPUs under Ax5 while only slightly increasing the number of responses in time.

10.2 About Workload Differences

The performance of individual policies depends primarily on the type of input data that is fed to the client emulator. If the request rate in the input data is consistently very low, all policies handle it well (minimal resource usage, minimal SLA max response time violations). This is because all policies at minimum schedule at least one service instance regardless of the request rate. If the request rate can consistently be handled by one service instance then there will be no need to increase the number of service instances. Furthermore, if there are many incoming requests that demand max resources available then there is minimal difference between policies since each policy will deploy the maximum amount of resources. However, if the request rate in the input data has a high variance the importance of the policy's performance increases significantly.

10.3 About The System Lag

The framework inherently has some lag due to the message passing and starting/killing executors in Mesos. These lag effects are actually quite small in comparison with other lag effects in the system. But for the results that are shown here, this might play a big role performance wise in the system. The fact that the input data set is compressed to be able to be sent 60 times faster means that all other time delays (such as the internal message passing in Mesos) should also ideally be reduced by 60 times as well. If the system's internal delay for starting a service instance is 1 second, this would correspond for a 60 second delay for the compressed data. The same is true for the polling time (the interval of the policy checking whether it should deploy new service instances). Since the polling time is set to 1 second, this means that during the emulations the system is using the real-world equivalent of polling times of 60 seconds.

10.4 Future Work

A sudden burst in the request rate, as is visible in Figure 10.1, typically results in numerous max response time violations. This occurs if a policy only deploys new instances when the max response time has been violated. There is a delay between detecting a response time violation and a new deployed service instance becoming active. Until this new instance is active all incoming requests are queued at the "old" instances. This typically results a large amount of response time violations during this interval while this queue is being processed by an inadequate number of service workers.

Another effect of this behavior can be seen in Figure 10.1. The figure shows some peculiar behavior of resource scaling up without breaking SLAs. In reality, this is due to request failures where API instances are unable to respond to clients within the load balancer's specified time out limit (60 seconds) and therefore the connection is terminated and simply dropped. Since the framework does regard these request failures as max response time violations no scaling of the number of service instances occurs.

The request density shows that the number of requests is not abnormally high, but is similar to the rest of input data. This behavior actually is the result of a sudden burst in the request rate which overloads the system and the system is unable to react in time.

There are a number of different approaches to solving this issue. One possibility is to, and this is the primary approach which was used in the pre-scheduled policy, begin deploying instances before the max response time is violated. By deploying new instances when the average response time of requests during an interval exceed half of the max response time, as defined by the SLA the impact of the issue is greatly reduced. Another approach would be to have the load balancer maintain a queue/buffer of its own. A service instance could then, through direct communication to the load balancer, ask the load balancer to pause the sending of new requests to that service instance until the service instance has reduced its own internal job queue. This would allow the load balancer to route these requests to the newly deployed service instances as soon as these are online. There are many constraints that have been placed on the current framework, and therefore these constraints should be removed in order for this framework to be usable in a real life application.

The current load balancer was chosen due to the authors' familiarity with the tool, but since there are many ways to improve the buffer and routing in the system, an even more customizable tool may be preferable.

Currently the evaluation framework is used only to study the performance of the framework and to evaluate different policies based on the input data set. But the evaluation framework can also be incorporated such that the policies implemented can use the information that it provides, for example, to build a learning algorithm. There are also very valuable data that can be obtained through correct profiling of the service instances, this can be done by sending controlled inputs to the system.

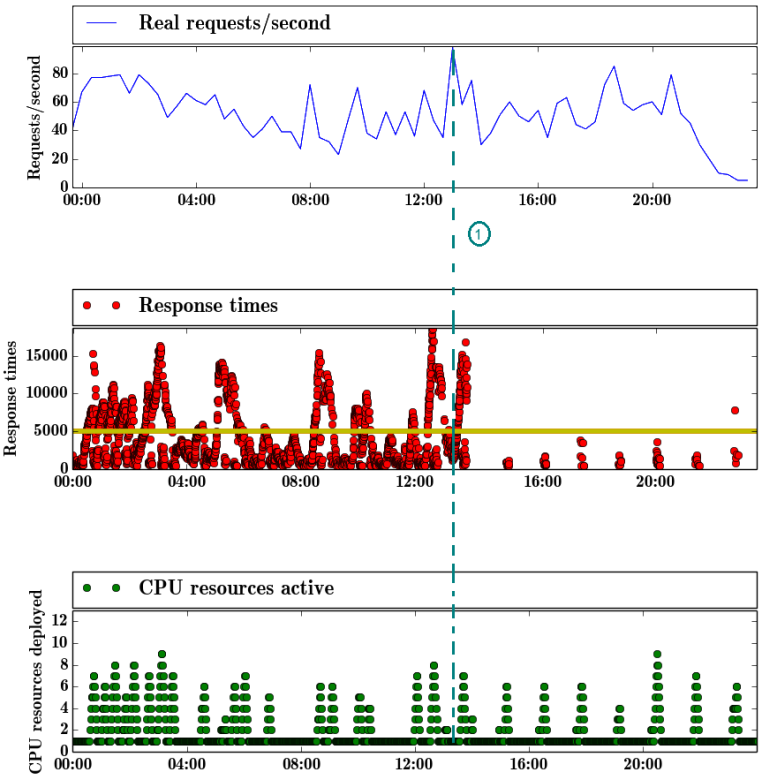

Figure 10.1 Naive Reactive. Note the spike in the request rate at 1. The subsequent deployment of new service instances is not quick enough to prevent a subsequent spike in the response times. This also leads to a workload queue which eventually leads to requests being dropped.

By using controlled profiling of the services more exact data regarding stress put on the system by different API calls. This data could be used to better predict how much many new service instances are needed to handle an increase in the workload.

10.5 Conclusion

A Mesos framework has been built such that it is simple to use any kind of policy. It allows for the policy to be studied by utilizing the evaluation framework. After

properly evaluating the policy based on an example input set that policy can then be customized to enable it to be handle the input load better.

More advanced and more generic policies can be built on top of the framework. The policies evaluated in this paper should be looked upon as a general start of how to build a policy to work with the system since the input to different service behaves very differently. It is important to note that this means that a policy for one input data set may give significantly different performance in comparison with other data sets. For example, different patterns in the short term workload may make a policy that has a longer wait before tear down time more appropriate. In other cases it may be beneficial to do scheduling based on the expected future workload when this workload exhibits a consistent time-based behavior. For example, if the workload always increases during Mondays we can preemptively schedule more resources during that time. The behavior of the services deployed to handle the incoming workload are important to take into consideration when designing a policy as well. The deployment time and tear down time of a service affect what type of scheduling policy is appropriate for such a system since the deployment and tear down times introduce lag effects in the system and to the chosen policy.

10.6 Related Works

Here we discuss some of the prior works that are related to the work of this thesis. We also discuss the similarities and differences of these prior works in relation to the work of this thesis.

Iqbal, Dailey, and Carrera (2009) [41] design and build a prototype for dynamic scaling of virtual servers based on service-level agreements defining the maximum average response time. They show that it is possible that to maintain such an SLA in experiments emulating a service while feeding the system with a synthetic workload. In contrast to a synthetic workload, this thesis uses a workload extracted from a large real world service. Furthermore, services of our thesis are emulated to mimic the behavior of the real world services at Data Ductus which are fed a real world workload. In the context of the SLA our thesis defines the SLA based on an upper maximum response time as opposed to a maximum average response time.

Kang, Koh, Kim and Hahm (2013) [42] show that dynamic scaling can be used to minimize SLA violations and costs. They propose a cost-oriented policy as well as a performance oriented policy. The workload used is based on long-running scientific data analysis jobs that take between 3 hours to several months to execute. In contrast to our thesis, the experiments by Kang, Koh, Kim and Hahm are simulated.

AutoScale [43] investigates dynamic capacity management policies for scaling. AutoScale works on a much larger scale than this thesis. While we, in this thesis, deploy new "tasks" running inside containers inside virtual machines, AutoScale deploys resources by starting new physical servers. Most of the policies examined in AutoScale also differs from ours. However, AutoScale proposes a conservative

shutdown policy which is similar to the the wait-before-tear-down generic policy described in this thesis.

Bia, Yuanb, Tiec and Tan (2013) [44] present an automatic management architecture for dynamic scaling of virtual resources based on SLAs. They examine a multi-tier architecture where each tier can be dynamically scaled. Among the types of SLAs they examine are SLA restrictions based on response times and cost. They perform their experiment using simulation and compares their proposed policy with two other policies.

Beltran and Guzmàn (2012) [45] show that by using dynamic scaling of virtual resources IaaS providers can improve the resource efficiency by 50% while still maintaining an acceptable QoS based on response times. They present a design and build an implementation of their framework.

Zhao, Peng, Yu, Zhou, Wang and Du (2013) [46] propose a dynamic scaling approach to virtual machines to lower costs while maintaining an SLA. They experiment with a varying workload while, by using their dynamic scaling scheme, achieve stable response times. They show that by utilizing the granularity in pricing that is becoming more and more common at many cloud computing providers, dynamic scaling solutions are able to adapt to a varying workload more accurately while still being cost effective.

Bibliography

[1] *Digital ocean website*. 14, 2016. URL: https://www.digitalocean.com/.

[2] *Amazon web services website*. Accessed: 2016-11-14. URL: https://aws.amazon.com/.

[3] *Old amazon pricing*. 2010. URL: https://web.archive.org/web/20100218101156/http://aws.amazon.com/ec2/pricing.

[4] *New amazon pricing*. 2016. URL: https://aws.amazon.com/ec2/pricing/on-demand/.

[5] *Netflix website*. Accessed: 2016-11-14. URL: https://www.netflix.com/se-en/.

[6] *Quora website*. Accessed: 2016-11-14. URL: https://www.quora.com/.

[7] *Netflix's use of aws*. 2016. URL: https://aws.amazon.com/solutions/case-studies/netflix/.

[8] *Quora's use of aws*. 2016. URL: https://www.quora.com/Which-Amazon-Web-Services-products-does-Quora-use.

[9] *Google website*. Accessed: 2016-11-14. URL: https://www.google.com/.

[10] *Facebook website*. Accessed: 2016-11-14. URL: https://www.facebook.com/.

[11] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale cluster management at Google with Borg". In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[12] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: flexible, scalable schedulers for large compute clusters". In: *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 351–364. URL: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

[13] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center*. Tech. rep. University of California, Berkeley, 2010. URL: http://mesos.berkeley.edu/mesos_tech_report.pdf.

[14] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, omega, and kubernetes". *ACM Queue* **14** (2016), pp. 70–93. URL: http://queue.acm.org/detail.cfm?id=2898444.

[15] *Dataductus website*. Accessed: 2016-11-14. URL: http://dataductus.se/.

[16] Mohammad Haghighat, Saman Zonouz, and Mohamed Abdel-Mottaleb. "Cloudid: trustworthy cloud-based and cross-enterprise biometric identification". *Expert Systems with Applications* **42**:21 (2015), pp. 7905–7916. ISSN: 0957-4174. DOI: http://dx.doi.org/10.1016/j.eswa.2015.06.025. URL: http://www.sciencedirect.com/science/article/pii/S0957417415004273.

[17] *Nist website*. 22, 2017. URL: https://www.nist.gov/.

[18] Robert B Bohn, John Messina, Fang Liu, Jin Tong, and Jian Mao. "Nist cloud computing reference architecture". In: *Services (SERVICES), 2011 IEEE World Congress on*. IEEE. 2011, pp. 594–596.

[19] *Architecting The Cloud*. John Wiley & Sons, Inc., 2014. ISBN: 9781118691779. DOI: 10.1002/9781118691779.ch2.

[20] *Google app engine documentation*. Accessed: 2017-05-14. URL: https://cloud.google.com/appengine/docs/.

[21] *Heroku*. Accessed: 2017-05-30. URL: https://www.heroku.com/.

[22] *Google drive website*. Accessed: 2017-02-07. URL: https://www.google.com/drive/.

[23] *Google drive pricing*. Accessed: 2017-02-07. URL: https://www.google.com/drive/pricing/.

[24] Melanie Gass. *A Day In The Worklife of Microsoft Office 365*. Lulu.com, 2013. ISBN: 1304627640, 9781304627643.

[25] *Digital ocean pricing*. 21, 2016. URL: https://www.digitalocean.com/pricing/#droplet.

[26] *Google compute engine website*. Accessed: 2016-11-14. URL: https://cloud.google.com/compute/.

[27] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems*. Prentice-Hall, 2007.

[28] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. 5th. Addison-Wesley Publishing Company, USA, 2011. ISBN: 0132143011, 9780132143011.

[29] Serge Haddad. *Distributed systems : design and algorithms.* Hoboken, NJ : John Wiley and Sons ; London : ISTE, 2011., 2011. ISBN: 9781118601365. URL: http://ludwig.lub.lu.se/login?url=http://search.ebscohost.com.ludwig.lub.lu.se/login.aspx?direct=true&db=cat01310a&AN=lovisa.004926656&site=eds-live&scope=site.

[30] *Bittorrent protocol*. Accessed: 2017-05-14. URL: http://www.bittorrent.org/beps/bep_0003.html.

[31] R. Dua, A. R. Raja, and D. Kakadia. "Virtualization vs containerization to support paas". In: *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. 2014. DOI: 10.1109/IC2E.2014.41.

[32] *Mesos in production*. 26, 2017. URL: http://mesos.apache.org/documentation/latest/powered-by-mesos/.

[33] *List of mesos frameworks*. Accessed: 2017-02-07. URL: http://mesos.apache.org/documentation/latest/frameworks/.

[34] *Protobuf*. Accessed: 2017-05-29.

[35] *Mesos containerizers*. Accessed: 2017-05-29. URL: http://mesos.apache.org/documentation/latest/containerizer/.

[36] *Nginx*. Accessed: 2017-05-30. URL: https://www.nginx.com/.

[37] *Haproxy*. Accessed: 2017-05-30. URL: http://www.haproxy.org/.

[38] *Nginx log module details*. Accessed: 2017-05-14. URL: http://nginx.org/en/docs/http/ngx_http_log_module.html.

[39] *Matplotlib*. 27, 2017. URL: https://matplotlib.org/.

[40] *Ubuntu*. Accessed: 2017-05-30. URL: http://www.ubuntu.org/.

[41] Waheed Iqbal, Matthew Dailey, and David Carrera. "Sla-driven adaptive resource management for web applications on a heterogeneous compute cloud". In: *IEEE International Conference on Cloud Computing*. Springer. 2009, pp. 243–253.

[42] Hyejeong Kang, Jung-in Koh, Yoonhee Kim, and Jaegyoon Hahm. "A sla driven vm auto-scaling method in hybrid cloud environment". In: *2013 15th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 2013, pp. 1–6.

[43] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. "Autoscale: dynamic, robust capacity management for multi-tier data centers". *ACM Trans. Comput. Syst.* **30**:4 (2012), 14:1–14:26. ISSN: 0734-2071. DOI: 10.1145/2382553.2382556. URL: http://doi.acm.org/10.1145/2382553.2382556.

[44] Jing Bi, Haitao Yuan, Ming Tie, and Wei Tan. "Sla-based optimisation of virtualised resource for multi-tier web applications in cloud data centres." *Enterprise Information Systems* **9**:7 (2015), pp. 743–767. ISSN: 17517575. URL: `http://ludwig.lub.lu.se/login?url=http://search.ebscohost.com.ludwig.lub.lu.se/login.aspx?direct=true&db=bth&AN=102340666&site=eds-live&scope=site`.

[45] Marta Beltrán and Antonio Guzmán. "An automatic machine scaling solution for cloud systems". In: *High Performance Computing (HiPC), 2012 19th International Conference on*. IEEE. 2012, pp. 1–10.

[46] He Zhao, Chenglei Peng, Yao Yu, Yu Zhou, Ziqiang Wang, and Sidan Du. "Cost-aware automatic virtual machine scaling in fine granularity for cloud applications". In: *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2013 International Conference on*. IEEE. 2013, pp. 109–116.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER´S THESIS

Date of issue
June 2017

Document Number
ISRN LUTFD2/TFRT--6037--SE

Author(s)
David Jaenson
Tuan Nguyen

Supervisor
Per-Gustaf Stenberg, Data Ductus
Johan Eker, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle
Automatic Scaling of Web Services using an Adaptive Distributed System

Abstract

In this thesis a Mesos framework is built to enable dynamic scaling of a collection of HTTP application programming interfaces (APIs) in response to a varying request workload. Several scaling algorithms ("policies") that run on top of this Mesos framework are designed and built. Furthermore, a performance evaluation framework is designed and built. The evaluation framework is used to evaluate the scaling algorithms running on top of the Mesos framework. The evaluation is performed by emulating two real world APIs using request data extracted from a real world system's access logs. Based on this evaluation the positive and negative attributes of the design of the Mesos framework and the policies are discussed.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-80

Recipient's notes

Security classification