A study on load balancing within microservices architecture

Computer science and engineering, 15 credits

Stockholm 2019-01-31
Alexander Sundberg
Abstract

This study addresses load balancing algorithms for networked systems with microservices architecture. In microservices applications, functionality and logic have been split into small pieces referred to as services. Such divisions allow for higher levels of scalability and distributivity than obtainable for more classical architectures where functionality and logic is packaged into large non-separable applications. As a first step, we investigate existing load balancing algorithms in the literature. A conclusion reached from this literature survey is that there is a lack of proposed load balancing algorithms for microservices, and it is not obvious how to adapt such algorithms to the architecture under consideration. In particular, many existing algorithms incorporate queues, which should be avoided for micro services, where the small services should be provided in fast manner. Hence, we provide modified and new candidates for load balancing, where one such is a probabilistic approach including a distribution that is a function of service providers’ load. The algorithms are implemented in microservices simulation environment developed in Erlang by Ericsson AB. We consider a range of scenarios for evaluation, where amongst other things, we vary the number of service consumers and providers. To evaluate the load balancing algorithms, we perform statistical analysis, where first and second order moments are computed for relevant metrics under the different scenarios considered. A conclusion drawn from the results is that the algorithm referred to as ”Round Robin” performed best according to the results from various simulation scenarios. This study serves as a stepping stone for further investigations. We outline several possible extensions such as more in-depth statistical analysis accounting for the time-varying aspects of the systems (currently omitted), as well other classes of algorithms.
## Contents

Abstract

List of Figures v

1 Introduction 1
   1.1 Purpose ................................................................. 2
   1.2 Problem statements .................................................. 2
   1.3 Specifications, requirements, and restrictions ..................... 3

2 Background 5
   2.1 Introduction .......................................................... 5
   2.2 Load balancing in distributed systems ............................... 5
   2.3 Distributed and centralized load balancing .......................... 5
   2.4 Service-oriented architecture ...................................... 6
   2.5 Microservices ........................................................ 6
   2.6 Simulation ............................................................ 7
      2.6.1 Network-simulator ............................................. 7
      2.6.2 Sim-Diasca ...................................................... 7
   2.7 Related work ......................................................... 7

3 Methodology 11
   3.1 Method specification ................................................ 11
   3.2 Method description ................................................. 11
      3.2.1 Simulator ....................................................... 12
      3.2.2 Theoretical base for load balancing algorithms ............... 16
      3.2.3 Proposed load balancing algorithms .......................... 17
      3.2.4 Scenarios ....................................................... 20
   3.3 Result Analysis ..................................................... 21

4 Implementation 23

5 Results 25
   5.1 Utilization of resources .......................................... 26
   5.2 Throughput .......................................................... 27
   5.3 Response Time ...................................................... 28
   5.4 Analyzing results according to metrics ............................ 30

6 Discussion 33
## Contents

7 Conclusion and future work

7.1 Conclusion ......................................................... 37

7.2 Future works ....................................................... 38

Bibliography .......................................................... 39
List of Figures

3.1 Service Provider .................................................. 13
3.2 Workload of 10 ticks ........................................... 14
3.3 Service Consumer ................................................ 15

5.1 Utilization of resources ......................................... 26
5.2 Utilization of resources for each scenario .................. 26
5.3 Throughput ......................................................... 27
5.4 Throughput for each scenario .................................. 27
5.5 Response time ...................................................... 28
5.6 Response time for each scenario ............................... 28
5.7 Response time without balanced algorithm ................... 29
5.8 Response time without balanced algorithm for each scenario ... 29
5.9 Bar chart visualizing each algorithms scenario specific superior results according to defined metrics ..................... 30
5.10 Bar chart visualizing each algorithms superior results according to defined metrics ........................................ 31
Chapter 1

Introduction

A new software architecture referred to as Microservices is currently adopted by large companies such as Google and Netflix [1]. This comprises a transition away from applications with a monolithic structure. Microservices Architecture means that functionality and logic are split into smaller pieces, referred to as services. This stands in sharp contrast to the classical approach where most functionality is built in a single application. By dividing the functionality into smaller sections, reusability and maintenance become significantly more accessible. As the logic is divided into different services, and these services are deployed separately on different nodes, services need a way to communicate with each other. The architecture only permits function calls to be remote rather than local. As a consequence, performance within the system can be compromised by high latency of network traffic. Thus, the communication infrastructure between the microservices needs to be lightweight [2].

Generally, a system with microservice architecture is built by many different services; each manifested by [3]:

1. Several instances providing the service
2. Load balancer(s) spreading the workload over available instances.

Load balancing is a common approach to improve availability and scalability within an application with microservice architecture. A system with microservices architecture is inherently distributed. High availability and scalability is required in a distributed system [2]. Since the distributed application is mostly structured as non-uniform [4], different parameters like workload and connection time can vary per service. As these parameters vary, the structure of a suitable load balancing algorithm may differ. Hence, it might be challenging to choose a good load balancing algorithm that meets the requirements of each service.
1.1 Purpose

The purpose of this project is to evaluate load balancing algorithms for various services within applications of microservice architecture. The project is requested by Ericsson AB. Since different types of services provide different functionality and logic, the relevance of a load balancing algorithm varies depending on the variability of load sizes and lifetimes observed for different services. To the best of our knowledge, there is a lack of load balancing algorithms specifically introduced for the microservices architecture. Beyond the scope of microservices, there are plenty of defined load balancing algorithms. A pertinent first goal of this inquiry is to investigate to what extent those can be adopted or adapted to the microservices architecture within the scope of requirements from the company issuing this project. Existing algorithms use different approaches to optimize parameters. The main goal of this project is to provide case studies for load balancing algorithms performance when used for various general use cases.

This first study paves the way for further inquiries with more specific use cases for an even more thorough evaluation of the different load balancing algorithms, where algorithms can be tuned and modified to meet specific requirements.

1.2 Problem statements

Different load balancing algorithms should be determined for load balancing a system with microservices architecture. Multiple research reports present algorithms for load balancing within different applications of distributed systems. As mentioned above, there is a lack of classifications of load balancing algorithms for microservices. The problems of this study are stated as:

- How to adapt existing load balancing algorithms, or provide new ones for microservices within the scope of requirements.
- According to different metrics, investigate if any of the included load balancing algorithms have comparatively superior performance within a certain scope.
- Furthermore, examine if any of the load balancing algorithms included in this project performs better than average in multiple scopes, e.g. good performance for multiple applications.
1.3 Specifications, requirements, and restrictions

The scope of the general problem defined in section 1.2 have been restricted in order to meet specifications provided by Ericsson while matching the time frame of the project. Limitations have been identified as follows:

- The research on the included algorithms for this project will only be examined for general use cases, meaning applications with general characteristics for different microservices intended to be implemented by the company issuing this project. More specific use cases will be excluded from this research. A specific use case can be a single application of microservices whose can be seen as an outlier in comparison with the other applications.

- The performance of an algorithm will only be examined individually. This means that mixed scenarios where different algorithms run concurrently amongst the nodes will not be considered.

Specifications for the outcome of the project have been discussed with the company issuing this project. The company is going to use distributed load balancing functionality for a communication mechanism meant for microservices. To have multiple algorithms available for the load balancing functionality is important to increase availability and scalability for different microservice applications. Conducted case studies for different algorithms is requested. Case studies for different algorithms performance within general use cases can be referred to when choosing an algorithm for the load balancing functionality within different services.

Regarding which algorithms to investigate, it is specified by the company that an algorithm to be included in this research should be able to independently make a load balancing decision. This means that when a load balancing algorithm is called upon to provide a forwarding decision, it is of high importance to make an independent decision without waiting for other factors within the system. Based on these specifications, algorithms involving load balancing of (or within) queues will be excluded.
Chapter 2

Background

2.1 Introduction

It is important to understand load balancing in general. Knowledge about the concept of distributed load balancing is also important since this is the type of load balancing functionality to be simulated and investigated.

2.2 Load balancing in distributed systems

An application in a distributed system can most often be split into tasks which are executed on different processing resources within the system. The performance of an application in a distributed system is dependent on the allocation of tasks over the available processing resources [5]. To achieve maximum efficiency for such a system, the load over the processing resources must be balanced referred to as load balancing. This is to prevent any resource from being overloaded or idle at any given time, which would compromise the objective of achieving maximum performance in the system [6]. The aims of load balancing is to optimize the usage of resources, maximize throughput of the system, increase response time and prevent overload of a processing resource [7] [4].

2.3 Distributed and centralized load balancing

Independent of whether the system is distributed or not, load balancing can either be centralized or distributed [8]. In the centralized approach, one node or a group of nodes are responsible for disposal of service information from end to end within the system. The centralized node or nodes are also responsible for distributing workload in a balanced
manner across the available providing nodes. The downside of a centralized approach is that the central decision making can be overloaded when used in larger systems; so-called single point of failure.

When using a distributed (also referred to as decentralized) approach, all nodes available in the system contributes to the load balancing functionality and load balancing decisions are taken locally at the nodes. The state information of a node should be disposed to nodes requiring state information. Every node is responsible for managing their own resources. This approach can provide better scalability [4].

2.4 Service-oriented architecture

Service-oriented architecture is an architecture for software design. Service-oriented architecture is an arrangement of units into groups that provide different functionalities, referred to as services [9]. Services, are requested, in turn, by service consumers. When a service is requested the functionality is provided to the consumer from a so-called service provider. The provider conducts the necessary work to provide intended functionality in the service.

Service consumers and service providers exchanges data via messages. Upon creation, a service is published, which enables service consumers to initiate contexts. For a service to be be so-called discoverable, the service should be published to the service registry. Via this service registry, consumers find services to direct service requests to.

2.5 Microservices

Microservices is a new branch of the service-oriented architecture; defined by Martin Fowler [10] ”...the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.”

Microservices are small operations intended to do a specific task that can be independently deployed, tested, and scaled. Complex applications are disassembled into fragments allowing for a fluent distribution of services that increases the performance of the system [11].
2.6 Simulation

2.6.1 Network-simulator

To simulate the behavior of a system with microservices architecture and to analyze the behavior of a system when load balanced with different algorithms, a simulator with high scalability and ability to simulate distributivity is required. We focus our attention to a particular simulator, referred to as network-simulator. Network-simulator is developed by Tomasz Bosak at Ericsson AB. The simulator is based on a simulator platform called Sim-Diasca, see below. The simulator is specifically designed to simulate the behavior of a network of microservices. Communication between nodes that is part of a network of microservices, and also the work that a node does locally.

2.6.2 Sim-Diasca

Sim-Diasca is an acronym for Simulation of Discrete Systems of All Scales[12][13]. Sim-Diasca is a simulation platform that enables discrete-event simulation of distributed systems of large scales. The simulator platform is implemented with the functional programming language Erlang with a lightweight layer called WOOPER (Wrapper for Object-Oriented Programming in Erlang)[14], which provides for implementation of object-oriented applications in Erlang. Erlang [15] has been shown to be a beneficial programming language when developing large-scale distributed applications [13]. To provide for simulations of distributed systems, the simulator engine includes predefined abstractions for actor-based models. The engine also provides a communication protocol to exchange data between the models during simulation.

2.7 Related work

After conducting a literature review, a lack of load balancing algorithms defined for microservices architecture was identified. Nevertheless, there is an extensive literature on load balancing in general (sometimes defined for adjacent architectures). In what follows, we try to capture the main directions and important algorithms found.

This survey in [8] conducts classifications for different important load balancing algorithms used within distributed systems, see the references therein. Strengths and weaknesses for the different distributed load balancing algorithms are presented, some of the algorithms will be presented below. Although, it should be emphasized that no simulation or real-life testing is presented by the authors. The classifications are relevant
for distributed systems in general, but no information regarding load balancing specific for microservices is presented. As this project will conduct a research for different load balancing algorithms performance within a system built with microservices; the basis of the studies is similar, but this study will differ in this regard.

Now we continue by addressing a collection of relevant load balancing algorithms below.

The algorithm called randomized [4] forwards a request to a randomly selected provider. Randomized does not consider any parameters when forwarding requests. This algorithm does not require any communication or retrieval of state information to make a load balancing decision. Although it is stated that this algorithm has the highest response time of all algorithms investigated in [4].

The Round Robin [16] algorithm forwards requests to the next available provider a queue. Round robin does not consider the length of job, which leads to higher response time and bad resource utilization if job time vary (compared to more homogeneous settings). That said, Round Robin algorithm performs well within a homogeneous environment.

This algorithm max-min [17], determines a minimum execution time for available jobs, and directs the job with the highest execution time to the provider with the lowest completion time. When directed, the job will be removed from the job queue and the next job with highest execution time will be directed to the provider with the lowest calculated completion time. A problem with the max-min algorithm is that low execution time jobs will have a higher response times.

The min-min algorithm [17] also determines a minimum execution time for jobs available, similar to max-min. The difference is that min-min will direct the job with the lowest completion time instead of highest to the provider with the lowest calculated completion time. Logically the downside of this algorithm is the opposite of max-min algorithm. High execution time jobs will have a higher response times.

Now, there are also so-called meta heuristics that have been introduced for load balancing. One such is the so-called Honeybee foraging algorithm. This algorithm is inspired by the way honey bees search for and harvest food [18]. Bees can be split into two categories, forager bees, and harvester bees. Forager bees search for food sources and return to the hive when a suitable food source has been discovered. This food resource is advertised by a forager bee, after which harvester bees follow the forager bee back to the food source to harvest. When harvesting bees return from the food source they inform the hive of the remaining quantity of food to harvest from the food source. This enables more bees to return to the food source if food is available or the food source to be abandoned if exploited.
This strategy can be implemented as a load balancing algorithm. Either a request is sent to a randomly selected provider, which represents foraging for a food source. Or a request is sent to the most profitable provider, which represents harvesting a food source. An advert board is available for consumers to retrieve information about the providers before deciding which provider to send the request to. After each request has been served, the providers profit is updated and compared to the global profit of the service which is listed on the advert board. The profit from a provider is decided on the elapsed time for a request to be served.

This algorithm works well in heterogeneous environments, and due to the scouting part of the algorithm, it does not provide for a proportional increase in throughput when resources are increased.

There is also a centralized, stated to be an improved version of the round robin algorithm presented in [19]. This algorithm directs requests in a classic round robin fashion. The improvement amounts to a central load balancing decision model (CLBDM). This model monitors sent requests and received requests by a user within a connection directed by the load balancer. When requests are monitored the expected completion time of that request is computed. If the time for that request exceeds a certain threshold, the request will be redirected with the vanilla round robin algorithm.

The drawbacks of this algorithm are a single point of failure may occur. However, the load balancing decisions are not dependent on the CLBDM. The model is a background monitor, meant to provide a better resource utilization without interfering with the initial load balancing decisions.
Chapter 3

Methodology

3.1 Method specification

As stated in Section 2.2, some of the goals of load balancing is to optimize utilization of resources, maximize throughput of the system and reduce response time. To evaluate these parameters for every algorithm in different scenarios, data is collected during the simulations and different mathematical operations are applied on the collected data. The obtained statistics is then analyzed to evaluate the relevance of the different algorithms within load balancing for microservices. The simulation tool Network Simulator will be used as specified by Ericsson AB. This simulator was developed on the platform Sim-Diasca, see Chapter 2. The project, and the methods therein, serve as a pre-study on load balancing algorithms for microservices and aims to meet the requirements put forward by Ericsson AB.

3.2 Method description

A literature study is first conducted to identify existing algorithms for load balancing. A thorough literature study is required to investigate different algorithms’ behavior when applied to distributed systems. Based on the information retrieved from the study several algorithms is chosen to be included in the project.

The next step is to implement the algorithms in the simulation environment. Different scenarios are defined where we vary the number of consumers, the CPU cycles and the CPU consumption/cycle. This is specified in Table 3.1 further down. We restrict the setting to all-to-all type, i.e., all consumers communicate with all providers. The scenarios capture varied and static workloads. The goal with these defined simulation
Chapter 3

scenario’s is to capture a range of scenarios including those where congestion nearly occurs as well as those that are less crowded. To be precise, this serves to examine the algorithms’ ability to handle various lifetimes of contexts and sizes in scenarios with higher and lower rate of providers in proportion to load balancers/consumers.

The simulator platform is called Sim-Diasca, see Section 2.6.2. The use of this simulator will speed up the development cycle in this project since implementation of real microservices on a simulator platform would be too time-consuming for the limited time-span of this project. By avoiding the implementation of a microservice architecture within a simulator platform it is possible to bypass many of the difficulties associated with development of real applications. Since we are interested only in the load balancing performance, this chosen approach serves our purpose. The programming language to implement the load balancing algorithms is Erlang since the simulator is developed with this language. We choose not to provide further details about the Erlang language and its specific characteristics at this point, see Chapter 4 for more details about the implementation process. Even though it should be mentioned that a great part of the development process centered around learning this language and deploying and tweaking the environment for the simulations.

All simulations produce a substantial amount of data to be used for statistical analysis, see Chapter 4 for details. Python scrips are developed for running statistical analysis and producing plots. A benefit of using Python as a tool is the availability of libraries for statistical analysis. Another relevant choice for deriving statistics and plotting would be MatLab. Due to the economic aspect of the project Python is used since it is open-source. For the use of MatLab a license to use the software is required.

The computer used for implementation, simulation and analyzing data produced from the simulations is provided by the company issuing this project, i.e., Ericsson AB. All software that is intended to be used is open-source.

3.2.1 Simulator

The simulator used is implemented on top of the network-simulator that is described in Section 2.6.1. A tick refers to 10 ms and is the discretization unit in the simulations that will be used. The simulation environment consists of three different types of so-called actors. The actors are:

- Service Provider

A service provider is an actor within the simulation environment, which receives service requests and provides that particular service. Each received service request
includes CPU usage $U$ and CPU cycles $n$ required to provide the requested service. Every provider has a maximum value of usage per cycle. This is stored as a hash table that stores current usage per cycle, referred to as a resource manager. The cycles are set as ticks. Thus, every tick within the simulation is a CPU cycle for the resource manager.

**Service Provider**

Figure 3.1 shows a service provider’s interactions with other actors within the simulation environment. Each arrow represents a message type, which is either sent or received by the provider, indicated by the direction of the arrow. Each box represents an action executed by the provider, these actions are explained below.

The request message indicates that a request for a service to be provided has been received by the provider. Based on the included CPU usage and cycles required for the request; the tick for the requested service to be provided is calculated with respect to the current CPU usage per cycle stored in the resource manager. The tick calculated to be the finish tick for the service is scheduled for returning a response, which will indicate that the requested service has been provided. This scheduled response is sent as a terminate message.

The terminate message is sent to an endpoint, referred to as Termination Service. A terminate message is sent every time a service has been provided by the provider (request completed). This message includes total usage for the provided service and time stamps for when the service request was sent and when the service was
provided. This message is scheduled and sent once for every service request provided. To enable that consumers make load balancing decisions based on available providers’ workload, each provider within the simulation sends it’s workload $\sigma$ with a certain interval $I$, referred to as the update interval to all consumers within the simulation. This is represented by ”push workload to consumers” in Figure 3.1. The workload is calculated according to (3.1) below where $\sigma(t)$ is workload from current tick $t$ until next update interval $t + I$, and $U(i)$ is CPU usage at cycle $i$.

$$\sigma(t) = \frac{1}{I+1} \sum_{i=t}^{t+I} U(i)$$  \hspace{1cm} (3.1)

This is to present providers’ current CPU usage since some load balancing algorithms require the workload of available providers to make a load balancing decision. The integration of the CPU usage over a time period equal to the time interval for pushing updates to consumers is seen as a valid surrogate of a provider’s current workload.

In Figure 3.2 an example of a provider’s CPU usage units per cycle and workload for 10 cycles is presented as a bar chart. The CPU usage units are shown over the y-axis and the cycles over the x-axis. The red dotted horizontal line represents

![Figure 3.2: Workload of 10 ticks](image-url)
the workload $\sigma(t)$ for presented CPU usages $U(i)$. The workload is calculated according to the function (3.1) with the interval $I = 10$, and $i$ for $U(i)$ is $t \leq i < (t + 10)$.

For all scenarios simulated in this project every provider has 100 CPU usage units available per cycle and every provider has an update interval $I$ equal to 12 ticks (120 ms), motivated in section 3.2.4.

- **Service Consumer**

The service consumer is the actor that makes use of the different load balancing algorithms to evaluate with its implemented load balancing functionality. Every service consumer generates a service request within an interval, referred to as generation pace. For all consumers in every scenario, the generation pace is set to 10 ticks, which is discussed in section 3.2.4.

![Service Consumer Diagram](image)

**Figure 3.3: Service Consumer**

In Figure 3.3 a service consumer’s interactions within the simulation and internal work-flow is presented. As mentioned above, the provider’s push their state information to all consumers existing within the simulation environment. When a consumer receives an update message from a provider, that provider’s state is updated in the consumer’s locally stored list of all providers state information.

Whenever a request should be generated, the consumer’s load balancer retrieves the list of stored state information. Based on the providers last known state information and the algorithm used, the load balancer makes a load balancing decision.
in terms of which provider to direct the request to. The different algorithms used
in different simulation scenarios are described below in Section 3.2.3.

- **Termination Service**

Termination Service is an actor within the simulation that receives messages from Service Providers when a service has been provided. For every simulation, there is only one Termination Service actor. Thus, every provider existing within a simulation sends its termination messages to the same Termination Service actor. This is to simulate a response when a service has been provided. Thus, every provider’s performance data throughout the entire simulation will be stored in one place, which facilitates when analyzing the data. The Termination Service receives every terminate message and collects the data included. The data included in a terminate message is CPU usage, cycles of CPU usage and time stamps when the service request was sent and when the service was provided. This is to determine the response time of that message. The Termination Service can, therefore, keep track of the performance of the system and response time for every provided service.

### 3.2.2 Theoretical base for load balancing algorithms

Every service consumer within the simulation has its own load balancing functionality, which is referred to as a load balancer. For every load balancer, a list of last known available providers process identifiers are stored to refer to when providing a load balancing decision. All actors within a simulation has a unique process identifier (PID), which is established upon construction. Let

\[
L = \{p_1, p_2, p_3, p_4, \ldots, p_n\}
\]

(3.2)

be the list of \(n\) provider PIDs, where \(p_k\) is the \(k\):th PID. When a service request shall be directed to a service provider, the load balancing algorithm at the consumer makes a decision of which provider it should select. This decision amounts to selecting a PID. Thus, the algorithms output \(p_i\) in Section 3.2.3 comprises the PID of service provider, which is located at index \(i\) in list \(L\) (3.2). Over time the selection of the PID for the algorithm varies depending on the resource availability of the providers and the specific design of the algorithm. The description of the algorithms can be simplified to describe the rules that should be applied for as each service is treated by the load balancing algorithm. This means that we restrict our choices of algorithms to such that are memory less; where decisions taken at a certain time instance for a certain service only depend on the current state of the system and do not depend past time’s decisions.
and states. Subsequently, let \( I_n \) denote the interval of integers between 1 and \( n \), where \( n \) is the number of elements in list \( L \) (3.2).

### 3.2.3 Proposed load balancing algorithms

1. **Random**

   The random algorithm is practically the same algorithm as the one referred to as "randomized" in section 2.7.

   An integer \( x \) is drawn from the uniform discrete distribution with \( I_n \) (see Section 3.2.2) as support. The selection of provider PID amounts, simply, the following

   \[
   p_i = p_x \tag{3.3}
   \]

   where \( p_i \) is the algorithm’s output, which is providing the selected provider PID (see Section 3.2.2) and \( p_x \) is the element at index \( x \) of list \( L \) in (3.2).

2. **Round Robin**

   This algorithm is inspired by the algorithm called round robin in section 2.7. A minor change is that the initial load balancing decision is completely random. This is to avoid a scenario where every distributed load balancer directs their initial request to the same provider.

   The load balancer stores a variable \( y \) of type integer, locally. The load balancer increments integer \( y \) after every directed service request. For the initial run of this algorithm a random integer \( r \) is drawn from the uniform distribution over \( I_n \) (see Section 3.2.2) and directs the first request to \( p_i \), where

   \[
   p_i = p_r \tag{3.4}
   \]

   In Equation (3.4), \( p_r \) is the element at index \( r \) of list \( L \) defined in (3.2) and \( p_i \) is the output of this algorithm’s load balancing decision (see Section 3.2.2). While the output is provided, let

   \[
   y = r \tag{3.5}
   \]

   where \( y \) in (3.5) is the stored variable and \( r \) the random integer drawn in the initial run.

   For load balancing decisions being made after the initial decision:

   The load balancer uses the sequentially updated integer variable \( y \), which is:

   \[
   y = r + k \tag{3.6}
   \]
where \( y \) in (3.6) is the updated variable after \( k \) number of requests directed including the random integer \( r \) drawn in the initial run. Let

\[
    x = (y \mod n) + 1
\]  

(3.7)

Where \( n \) in (3.7) is the number of elements in stored list \( L \) in (3.2), the modulus operation enables this algorithm to iterate through the elements of \( L \) in (3.2) in a cyclic and endless fashion. When \( x \) has been computed, locally stored \( y \) is incremented by one unit and the provider PID is selected as

\[
    p_i = p_x
\]  

(3.8)

where \( p_i \) in (3.8) is the algorithms output which is described in Section 3.2.2 and \( p_x \) is the element of list \( L \) at index \( x \) where \( x \) is the variable computed in (3.7).

The two following presented algorithms are both inspired by max-min and min-min algorithms, which, are described in section 2.7. Since one of the requirements of the microservices architecture under consideration is the exclusion of load balancing algorithms that involves any form of queues; the two following algorithms do not store future service request in any form of queue. Thus, both algorithms directs requests instantly.

Although, both algorithms does account for the provider’s computation time of providing the requested service, in terms of the providers’ current workload. Hence, assuming that the provider with the lowest workload has the lowest computation time.

3. Probabilistic

This algorithm is based on the assumption that a provider with a lower workload has a higher probability of providing a service with a lower computation time than a provider with higher workload, and therefore directing it to the provider with the last known (since last received update of providers state) lowest workload. In essence, when selecting a provider, the load balancer is computing a probability distribution as a function of the work loads of the providers. Then a PID is drawn from this distribution. We now make this specific.

The load balancer first provides a uniform random number over \( I_n \) (see Section 3.2.2). Let

\[
    \alpha(x) = 1 - w(p_x)
\]  

(3.9)

Where \( w(p_x) \) is the workload of \( p_x \), where \( p_x \) is the PID of the element at index \( x \) in list \( L \) (3.2) and \( \alpha(x) \) is a positive weight proportional to a probability
corresponding to \( p_x \). Let
\[
P(x) = \frac{\alpha(x)}{\sum_{k=1}^{n} \alpha(n)}
\] (3.10)
where \( P(x) \) is the probability that \( p_x \) should receive a service request. Now, a random number is drawn from the uniform (continuous) distribution with \([0,1]\) as support. Provide a uniform random number \( r \) within the range of \( 0 - 1 \).

\[
P(x) > r
\] (3.11)
where \( P(x) \) is defined in (3.10), then

\[
p_i = p_x
\] (3.12)
where \( p_i \) refers to the algorithm’s output where the load balancing decision is provided and \( p_x \) is the provider PID located in list \( L \). \( L \) is defined in Equation (3.2). The algorithm starts over when:

\[
P(x) \leq r
\] (3.13)
where \( P(x) \) is defined in (3.10) and \( r \) is a random number within the range of \( 0 - 1 \).

4. Balanced

This algorithm plainly selects the provider with the lowest workload accordingly:

\[
p_x = \min(L)
\] (3.14)
where function \( \min(L) \) returns the PID \( p_x \) of the provider with the lowest workload in list \( L \) at index \( x \). \( L \) is defined in (3.2). Remove element \( p_x \) from list \( L \) and make a new list with \( p_x \) as the first element:

\[
L_{new} = \{p_x, p_1, p_2, ..., p_n\}
\] (3.15)

Store \( L_{new} \) as

\[
L = L_{new}
\] (3.16)
where \( L \) is defined in (3.2). Let

\[
p_i = p_x
\] (3.17)
where \( p_i \) refers to this algorithm’s output for a load balancing decision and \( p_x \) is the element at index \( x \) in list \( L \) which is defined in Equation (3.2).

Since the \( \min(L) \) function in (3.14) starts to iterate through list \( L \) from the last element of the list to search for the provider with the lowest workload. The \( p_i \), in
which the service request is directed to, will be stored as the first element of list $L$ after directed service request. This is to prevent consecutive service requests from being directed to the same provider $p_x$, if more than one provider has the same workload.

### 3.2.4 Scenarios

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Consumers</th>
<th>Providers</th>
<th>CPU consumption/cycle</th>
<th>CPU cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>50</td>
<td>25</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>75</td>
<td>50</td>
<td>25</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>50</td>
<td>25</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>50</td>
<td>10+r(1-20)</td>
<td>6+r(1-8)</td>
</tr>
<tr>
<td>5</td>
<td>75</td>
<td>50</td>
<td>10+r(1-20)</td>
<td>6+r(1-8)</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
<td>50</td>
<td>10+r(1-20)</td>
<td>6+r(1-8)</td>
</tr>
</tbody>
</table>

Table 3.1: Table presenting different simulation scenarios. Function $r(x-y)$ represents a uniform distributed random integer within the range of $x$-$y$.

The scenarios presented above in table 3.1 is a compact presentation of the different simulations. Consumers and Providers are specifying the number of consumers and providers active in a simulation scenario. CPU consumption/cycle and CPU cycles are presenting the workload a consumer will consume from a provider with each service request sent. CPU consumption/cycle is defining the usage a job will consume of a providers CPU and CPU cycles for how many cycles this job will consume the specified amount of usage of a providers CPU.

These scenarios are simulated separately to evaluate the performance of the different algorithms presented in subsection 3.2.3. Thus, these scenarios are simulated once per algorithm. For each simulation scenario purposed for one of the proposed algorithms, that specific algorithm is being used by every consumer in the simulation when load balancing. Every simulation scenario has a simulation time of 6000 ticks, which represents 60 seconds, e.g. 1 tick = 100 ms. Since every consumer’s generation pace is set to 10 ticks (1 second), each consumer approximately generates 60 service requests during the simulation time and that is seen to provide enough data to evaluate the performance of every algorithm. The decision of the update interval for the providers is based on the generation pace of the consumers. It is set to 12 so every consumer will nearly receive one update per generated service request.

With the time aspect of this project taken into consideration when deciding on different simulation scenarios, the scenarios presented above in table 3.1 are the scenarios simulated and analyzed in this project. Multiple other parameters could be taken into
consideration and tweaked throughout different scenarios, beyond the presented scenarios in table 3.1. Due to limitations because of the time frame, more widely covering scenarios are not evaluated in this project. Suggestions for scenarios beyond the presented ones are discussed later on in future works section.

Different amounts of consumers instead of varying the numbers of providers are simulated. This is to evaluate the different algorithms performance for an overcrowded network as well as a network where the same number of consumers as providers are consuming workload. The consumers provide the load balancing functionality within a system. Thus, the number of consumers varies in the different presented scenarios to see how well the proposed algorithms in subsection 3.2.3 perform with different numbers of load balancers. According to previous work in section 2.7, some algorithms can perform worse in a heterogeneous environment rather than in a homogeneous environment. These scenarios presented above are meant to cover simulations of homogeneous environments as well as simulations of environments of a more heterogeneous kind in terms of varying workload. With simulation scenarios that cover both static and varying workloads and these being produced and spread over 50 providers by different numbers of consumers, the proposed algorithms in subsection 3.2.3 performance can be analyzed within multiple environments e.g. use cases.

3.3 Result Analysis

To be able to analyze load balancing algorithms performance and conduct a case study, specific metrics are defined. These metrics will be to measure different parameters relevant to a load balancing algorithms performance. Metrics is defined as following [20] [8]:

- Utilization of resources

  Calculating the variance for the mean for each tick with a reported workload for the whole system. This is done for all scenarios per algorithm. Thus, 6 variances will be presented per algorithm to see how the load varies.

- Throughput

  Throughput is one of the parameters that can show the performance of a system. It will be calculated as mean CPU usage per tick, per provider. Hence, total CPU usage for the whole system divided by the amount of ticks for the simulation per provider. This result will also be in form of 6 results per algorithm.
• Response time

Response time is one of the important parameters to optimize when load balancing. The response time should be as low as possible. A mean response time per scenario produces 6 mean response times per algorithm to evaluate.

Every result for these metrics will be visualized with boxplotting, to get a good view of the different load balancing algorithms performances within these different scenarios.

The load balancing algorithms should have good performance with respect to the defined metrics and measures. Generally, a decent load balancing algorithm should be able to handle a certain number of service requests not exceeding a specified latency limit.

Statistical data is compared for the different algorithms and the different scenarios. Use of first and second order moments, we perform a thorough statistical analysis, with the aim of identifying differing performance amongst the algorithms for the scenarios defined in Table 3.1. in Section 3.2.4. The aim is that this statistical analysis provides means, as well as insight, for future investigations on how to optimize different load balancing algorithms; in particular how to tweak parameters for the algorithms to optimize performance. An example is the probabilistic method where the weight functions could be chosen to be parameterized polynomials or some monotone function instead of a linear function.

We should mention at this point that a possible risk with such a statistical comparison is that it provides small differences for different algorithms in different scenarios. But this could actually be seen as an interesting result and not jeopardize a good outcome for this project.
Chapter 4

Implementation

To implement a simulation environment that provides for the evaluation of different load balancing algorithms, three actors have been defined and implemented. These actors are referred to as ”service provider”, ”service consumer”, and ”termination service”. The behavior of the different actors is described in Section 3.2.1 The actors have been implemented based on the interface of a generic service for microservices predefined within the simulator network-simulator, the network-simulator is described in section 2.6.1. This generic service interface provides for communication between the actors within the simulation as well as simulating work being done locally by an actor. The actors have been developed as classes in an object-oriented environment to easily implement the predefined interface and be constructed with a few parameters defining the behavior of an actor.

The entire load balancing functionality has been implemented in the class called service consumer. The functionality consists of different methods which return a unique provider ID intended to direct the next service request to. These methods conduct logic according to the different load balancing algorithms and are explained in Section 3.2.3. Which algorithm to use when directing service requests are established upon the construction of a consumer. Some of these algorithms require available providers state information to provide a load balancing decision. Thus, a method implemented will be triggered when a state information update from a provider is received by a consumer. This method updates that specific provider’s workload in a list containing all available providers workload. This list is locally stored at that consumer.

As previously described, the service requests are received by the provider class. The functionality to simulate the providing of services were already defined within the interface for a generic service. Although, the functionality for pushing updates containing the provider’s state information have been implemented. The initial update is sent to
every available consumer within the system when the provider is initiated. When the update messages have been sent, the provider object schedules the next push of state information. The scheduled time depends on the update interval which is established upon the construction of a provider object.

The actor called termination service which is also described in Section 3.2.1 were already implemented to be used in another simulation environment within the network-simulator. This class has only been modified to conduct the purposes of this projects simulations, e.g. providing the data necessary to statistically analyze the results for a load balancing algorithm.

To facilitate the retrieval and processing of this data; a Python script has been programmed. This Python script retrieves all data produced by simulation. All data produced by a simulation is stored in a folder. The path of this folder is specified as a parameter when initiating the script. Upon initiation, the script runs mathematical operations on data located in specified folder to produce the defined metrics in Section 3.3.
Chapter 5

Results

In this chapter, the results of all simulations are presented and compared according to the defined metrics in Section 3.3. The simulation results are presented in form of boxplots and bar charts. The boxplots are used to identify outliers for the different algorithms as well as visualizing the overall results of the algorithms according to a specific metric for every simulation scenario. The bar charts are meant to provide visualization for every algorithm’s performance to identify scenario specific results. Every simulated scenario is presented and explained in Section 3.2.4

The boxplots are presenting the value of presented metric on the y-axis and the algorithm on the x-axis. The bar charts are also providing the value of the visualized metric result on the y-axis, but since these plots are scenario specific, the number of consumers used in the simulation is represented on the x-axis. For every metric, the bar charts are also plotted in two different graphs. The first graph plots the simulation scenarios where the size of the workload for a service request is static, e.g. every service request will consume the same amount of workload throughout the simulation. These scenarios are numbered 1-3 in Table 3.1. The second graph plots result for the scenarios where the workload varies for each service request within a certain range. The scenarios with dynamic workload are numbered 4-6 in Table 3.1.
5.1 Utilization of resources

This section presents simulation results according to metric utilization of resources, which is defined in Section 3.3.

![Box plot showing utilization of resources](image1)

**Figure 5.1:** Utilization of resources

In Figure 5.1 the simulation results according to the metric utilization of resources algorithms are represented in form of a box plot. This plot is showing the mean, variance and outliers in workload on the y-axis and the different algorithms on the x-axis. This shows the spread of all simulation results for every algorithm.

![Bar chart showing utilization for each scenario](image2)

**Figure 5.2:** Utilization of resources for each scenario

The utilization results are also plotted in form of a bar chart in Figure 5.2. On the x-axis, the amount of consumers using the load balancing algorithm in a specific scenario is plotted with the variance of workload throughout that simulation on the y-axis. In this figure, there are two plots, one showing the simulation results for scenarios with static workloads and the other are visualizing results for simulation scenarios with dynamic workloads.
5.2 Throughput

The simulation results for all scenarios are presented below in terms of the metric throughput, which is defined in Section 3.3.

![Figure 5.3: Throughput](image)

The boxplot in Figure 5.3 above shows every simulation scenarios throughput result for each algorithm. The throughput is shown over the y-axis and the different algorithms over the x-axis.

![Figure 5.4: Throughput for each scenario](image)

In Figure 5.4 there are two bar charts. The first is presenting the simulation results according to the throughput metric for simulation scenarios with static workloads and the second one plotting the simulation results for scenarios with dynamic workloads. On the x-axis, the number of consumers used in a scenario is shown and the y-axis shows throughput results.
5.3 Response Time

The average response time for every simulation scenario is presented below. The metric response time is explained in Section 3.3.

![Figure 5.5: Response time](image)

Figure 5.5 is a boxplot presenting all scenarios average response time. This is plotted with the response time in seconds on the y-axis and the different algorithms on the x-axis.

![Figure 5.6: Response time for each scenario](image)

The bar charts above in Figure 5.6 visualizes every scenarios simulation result according to the metric response time. The results are presented with seconds on the y-axis and the number of consumers used on the x-axis. The first graph shows the results for simulation scenarios with static workloads and the second graph presents the results for scenarios with dynamic workloads.
Another set of plots are provided below to be able to more thoroughly analyze the results for the following algorithms: Probabilistic, Round Robin and Random. Hence, the results of the algorithm called Balanced is excluded from the following plots to help distinguish differences between the remaining plotted results.

Figure 5.7: Response time without balanced algorithm

Figure 5.7 is a boxplot plotting the same values as Figure 5.5, with the only difference that this plot does not visualize simulation results for the algorithm Balanced.

Figure 5.8: Response time without balanced algorithm for each scenario

The bar charts above in Figure 5.8 is plotting the same values as Figure 5.6 but excluding the simulation results for the algorithm called balanced.
5.4 Analyzing results according to metrics

When analyzing the results for every simulation scenario according to the metrics, superior results per scenario for each metric is concluded. Every superior result in any of the metrics for a simulation scenario has been summed. This is to evaluate the overall performances for the different algorithms when simulated within the different scenarios.

Figure 5.9: Bar chart visualizing each algorithms scenario specific superior results according to defined metrics

Figure 5.9 above shows how many superior results according to the metrics an algorithm had for every simulation scenario. The figure contains 3 bar charts which are plotting the different algorithms performances for the different scenarios simulated. In which of these bar charts the result of a scenario is shown depends on the number of consumers used for that scenario. On the x-axis, the amount of superior results an algorithm had is shown and the y-axis shows the algorithms name. The color of a bar determines if an algorithms superior result was for a static or a dynamic workload.

Seen in Figure 5.9, algorithm random had the best results for varied workloads being spread over the same number of providers as consumers. Although, with the same amount of consumers as providers load balancing static workloads, round robin had the best performance. When the number of consumers is higher than the number of providers, round robin performed best in the scenarios with dynamic workloads. Round robin also performed best when 100 consumers load balanced static workloads across the providers. However, the algorithms round robin, probabilistic and random had one superior result each according to the metrics. Round robin had the lowest response time.
while random had the highest throughput and probabilistic had the lowest variance in workload.

Figure 5.10: Bar chart visualizing each algorithms superior results according to defined metrics

Figure 5.10 presents every superior result for each algorithm in one graph. This is to specifically visualize the algorithms overall performances for every scenario simulated. Across the x-axis, the algorithms names are presented and the number of superior results is presented across the y-axis for all scenarios simulated.

Round robin had the best simulation results compared to the other algorithms simulated with 10 superior results as seen in Figure 5.10. Even if the results are split into dynamic and static workloads round robin performed best. Random performed decently with 5 superior results which are 3 with dynamic workloads and 2 with static workloads. The probabilistic algorithm performed worse than round robin and random overall but had the same number of superior results for static workloads as the random algorithm. Balanced did not have a single superior result for any simulation scenario and therefore performed worst of every algorithm simulated.
Chapter 6

Discussion

Results are presented thanks to the implementation of the different proposed algorithms and simulations of these for various scenarios. These results indicate with no doubt that round robin had the best overall performance according to defined metrics. The balanced algorithm did not have a single superior result, which can indicate that this kind of aggressive load balancing approach may not be suited for this kind of distributed load balancing. Since every load balancer is individually trying to optimize the system by directing service requests to the provider with the lowest load, and there are multiple load balancers in this distributed load balancing approach. Thus, every load balancer seems to send their service requests to the same provider and therefore overloading that provider.

To provide an even better evaluation of the proposed algorithms more scenarios could be simulated and analyzed. For example, simulating scenarios with different numbers of providers and varying the generation pace of service requests. Comparing the different results for the proposed algorithms according to the metric throughput, it is shown that these results did not have a big difference for any of the scenarios. This indicates that the simulated scenarios did not quite saturate the system. To see how the algorithms performed with larger service requests or faster generation pace of service requests, e.g. nearly saturating the system would most likely provide a bigger difference in throughput results for the different algorithms. Thus, provide a better result for the project in terms of evaluating the performance for systems using the different algorithms. The similar results for the metric throughput can also depend on the time simulated for these different scenarios. If a longer time period were simulated, the throughput results for the different algorithms could have a bigger variation for the different algorithms and
therefore more distinguishable differences. Also, additional statistical analysis operations could have been done when analyzing the data produced by the different scenarios simulated to present an even deeper analysis of the simulation results.

According to an article reviewing different load balancing algorithms in Section 2.7[4], it is stated that the random algorithm had the highest response time of all algorithms reviewed. This can be referred to when analyzing and discussing the results of this project since this project also included a random load balancing algorithm. Thus, a decent load balancing algorithm should generally perform better than the random algorithm when load balancing. Given the results presented; the algorithms called balanced and probabilistic performed worse than the algorithm called random and therefore does not seem suited as algorithms for load balancing microservices by this projects point of view. In Section 2.7 [16] it is also presented that another survey of load balancing algorithms in distributed systems states that the round robin algorithm does not perform as well with more heterogeneous settings. This indicates that varying the workload was not quite enough to simulate a heterogeneous environment. To create and simulate a more heterogeneous environments more parameters can be varied. It would be interesting to simulate environments with consumers that have different generation paces of service requests as well as providers with different CPU capacities. Since not all of these parameters have been evaluated but only set to a fixed value for every scenario simulated in this project, the conclusion of the results for the proposed algorithms performances should not be seen as final conclusions. For example, there is a possibility that the proposed algorithm referred to as probabilistic can perform better than the random algorithm and even round robin in an even more heterogeneous environment. Since probabilistic considers the last known workload but yet does not load balance in the same aggressive fashion as the algorithm referred to as balanced. Thus, the conclusions of these results should rather be seen as a stepping stone towards further investigations into the subject. Such investigations can be done in multiple directions and a few of them that we identify will be outlined in the future works section.

For load balancing to be executed instantly by every consumer, algorithms based on providers state requires the providers to inform the consumers about their state with a certain interval. This interval has been set to a fixed value for all scenarios simulated in this project but varying this value for different scenarios could also affect the results for the algorithms performances. How a load balancing algorithm which is balancing load based on available providers state perform can be affected by the differences between the update interval for the providers and the generation pace of service requests within the system, e.g. how many service requests for each consumer will be load balanced and directed per update.
In the introduction, it is stated that large companies are adopting systems with microservices architecture. As larger companies services are used by a large part of our society, their actions can affect different areas in our society, such as environmental and economical. In Section 2.2 it is stated that balancing the load over processing resources can maximize efficiency by avoiding overload of resources as well as minimizing the idle time. Thus, by balancing loads within microservices architectures with efficient and relevant load balancing algorithms; the hardware energy consumption for these systems will be lowered. A lowered energy consumption for the large system’s is beneficial within both environmental and financial aspects of our society.

Maximizing the efficiency in terms of reducing idle time while avoiding overload for a system also results in increased system performance. A system implemented with a well-performing software architecture will provide for fast and reliable service consumption and therefore increase the number of implementations and users of this architecture. Since this type of architecture is avoiding centralized implementations and functionality, personal information stored is spread in a decentralized fashion and thereby increasing the security of stored sensitive information.

This project has delivered an initial evaluation of different algorithms and a proposed algorithm that potentially can be used when load balancing within a microservices architecture. To provide for a more thorough evaluation of these algorithms, fixed parameters such as generation pace and update interval should be varied in additional simulation scenarios. Although, because of the execution of this project, a simulation environment to simulate different load balancing algorithms and tools to visualize and analyze simulation results have been developed.

The simulator platform required the implementation and tweaking of the simulation environment to be done with the programming language Erlang. Since the student who conducted this project had no experience with the programming language Erlang, a big part of the project was to get familiar with the language and the simulator platform. This lead to less time than estimated to simulate and analyze different algorithms. Although, an initial evaluation of load balancing algorithms was achieved as planned while providing a simulation environment to Ericsson AB to enable future research.
Chapter 7

Conclusion and future work

7.1 Conclusion

The thesis has proposed algorithms for load balancing systems implemented with microservices architecture, according to requirements by Ericsson AB. This project has been issued to gain insight on different distributed load balancing algorithms within a microservices architecture. The project was structured to provide an evaluation of the performance of different load balancing algorithms for different scenarios.

Thanks to the approach evaluating the different load balancing algorithms, an expansion of functionality for an existing simulator was implemented. This simulator provides for simulating microservices architecture. The expansion of the simulator provides for functionality to simulate different load balancing algorithms.

The thesis concluded that the algorithm called round robin performed best in most scenarios except for one where round robin, probabilistic and random performed well according to one metric each. Also, that an aggressive load balancing approach like the algorithm balanced is not suitable for these kinds of systems since the algorithm did not perform well in a single scenario according to any of the defined metrics. The scenarios aimed to provide for simulations of heterogeneous environments did not receive expected results but were still relevant to evaluate. Although, since multiple parameters could not be varied because of the limitations, it is stated that it is possible that the probabilistic algorithm could receive a better result for more heterogeneous systems. Thus, the projects conclusions should be seen as a starting point for future investigations since the time period limited the scenarios evaluated.


7.2 Future works

More scenarios could be simulated and analyzed. These scenarios could vary the generation pace of service requests, and CPU capacity for providers. Also simulating with different numbers of providers within a system. This will provide for performance results for the different algorithms in an even more heterogeneous environment. Since microservices are meant to provide for various functionality and logic this is highly relevant.

There is plenty of room for introducing new algorithms. The algorithms obtained from the literature were defined for other architectures than the one under consideration, and it would be interesting to define equivalent algorithms for the micro services architecture. The limited time span of the study did not permit for such research, but would serve as an obvious candidate for future investigations. Furthermore, we hypothesize that the simplicity of the microservices architecture makes it possible to cast the load balancing problems into forms that resemble those of classical (linear) optimization problems such as the assignment problem or the max flow problem. However, is not certain that the solutions to those would be possible to implement in a distributed fashion while respecting the constraints of the micro services architecture.

Another thing to consider, would be to incorporate algorithms which are not memoryless and more resemble dynamical systems models. Decisions made by the algorithms in this study are based on state information obtained at snapshots in time, and could be extended to depend on time. Those algorithms could be seen as models optimized over a model class as is done in the field of systems identification. Yet, another direction would be to consider machine learning methods, in particular supervised learning methods where we train algorithms to meet certain desired objective that are perhaps even scenario specific.

In the evaluation of the algorithms one could consider scenarios where the communication between the consumers and the providers is not all-to-all. This means, in the most general case that we could have a dynamic network of consumers and providers and each consumer can only send request to a (small) subset of providers. The latter setting is more realistic in real world applications we proximity between providers and consumers can be based on euclidean distance and connectivity.

Now, with regards to the statistical analysis, we identify several extensions to the one conducted in this study. It would be interesting to look at cross correlations and perhaps even causality between the states of the providers over time. This would allow to identify the reasons behind phenomena such as overload at a provider. We have to a large extent omitted to consider the aspect of time, and such an extended analysis would help to give a better understanding of how things evolve over time.
Bibliography


Alexander Sundberg