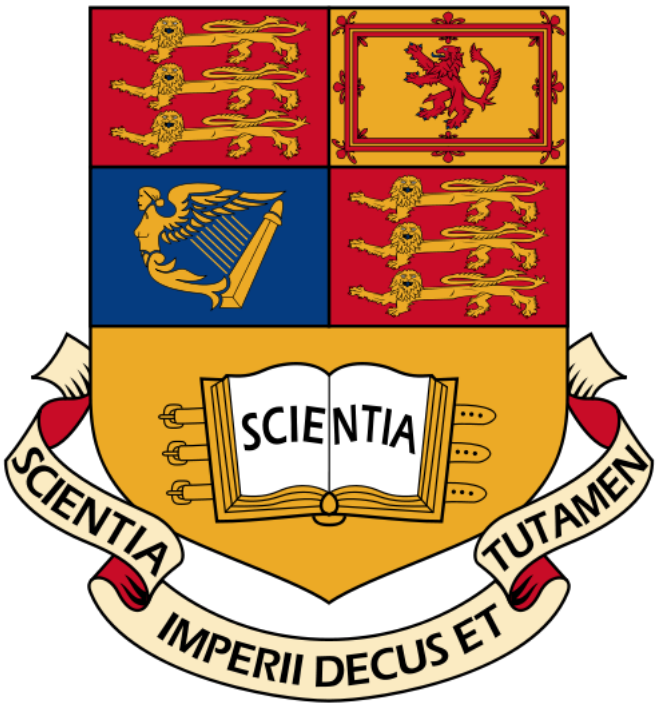


IMPERIAL COLLEGE LONDON - DEPARTMENT OF COMPUTING

MENG FINAL YEAR PROJECT

Performance Evaluation of Cloud Services

Author:
JAMAL ZARAK KHAN

Supervisor:
DR. MARIA VIGLIOTTI
Second Marker:
DR. WAYNE LUK

Submitted in part fulfillment of the requirements for the degree of MEng Computing (Software Engineering).

JUNE 20, 2013

Abstract

Cloud computing is everywhere. It has enabled the concept of computing to become a utility that can be used on demand as a service. Performance and cost aspects of cloud computing are very important in a world where most organizations and institutions are leveraging the cloud more and more.

To evaluate the performance and costs we must infer simple models from a seeming complex system. We must decompose the cloud into components and use these as the building blocks for analysis.

Queuing theory provides a very concise mathematical approach to analysing systems that involve a waiting and servicing period.

In this project we practically analysed the service times for database and web server components for different cloud computing options (Amazon EC2 instances) and fitted them to a model of a queue. This was followed by cost and performance analysis of the different cloud options and building more complex models for analysis. Our findings were that larger Amazon EC2 instances have a higher cost per request than smaller instances.

Acknowledgements

I would like to thank my supervisor Dr. Maria Vigliotti for her invaluable help and being patient with me throughout this project. Getting an insight from Dr. Giuliano Casale about my project topic was very helpful and I am grateful for his help.

Without the rigorous courses I have taken at the Department of Computing I wouldn't be be able to tackle lots of interesting problems out there in the world, and I would like to thank all the teaching staff who have directly and indirectly influenced my learning throughout the degree. My personal tutor Professor Paul Kelly should be mentioned who was a large motivating factor in my first year.

The friends I have made throughout college life have influenced and changed my views about the world. I would like to thank Danish Khan and Alessandro Galli for the never ending escapades. Special mention to Rafal Szymanski who helped out with this project and has been a constant source of motivation. I would also like to thank Suhaib Sarmad, Maciek Albin, Salim Brigui, Amir Al Fakieh, Bilal Abou El Ela, Abhijit Chandgadkar & Oyetola Oyeleye.

Finally there is no greater source of motivation and perseverance that I can find than from my family. The consistent persistence from my father (Zarak), mother (Farhat) and grandmother (Zubeida) has moulded me and for this I am always grateful. My siblings Qaiser, Nazakat, Zeenat and cousin Javed have always believed in me. I would like to also thank my aunt Rubina and my uncle Ayoob who gave me the opportunity to start using a computer from a very young age.

Contents

Chapter 1

Introduction

Many software developers leverage services that are provided by the so-called 'cloud' to develop applications and websites. Before the era of cloud computing, programmers spent lots of time deploying their software on servers which they had to customize to suit their needs as opposed to developing it – configuration and setup was a big hassle.

With the advent of cloud computing, time and expense to configure and deploy software has been reduced to a fraction of what it was before leading to companies spending much more time developing software rather than deploying it.

However, companies relying on cloud services face outages and denial-of-service attacks which affects their customers in turn. There is also inefficient usage of the cloud as these companies and developers do not know which service options to take, for example whether to use a larger or smaller server for their task. This choice has cost implications, which can often fall in the millions for large corporations.

Companies are often also confused which cloud computing components to use as there is limited performance tools and they are often stuck with the component choices they make for a long time.

Queuing theory is a mathematical approach to model and study queues together with their associated parameters and evaluations.

This project tries to focus on how we can effectively model and analyze the performance of cloud services using queuing theory.

1.1 Outline of this report

In Chapter 2 we first explore what cloud computing really is with a focus on Amazon Web Services (one of the largest cloud providers in the world). We briefly look at some probability distributions, maximum-likelihood estimation, stochastic processes, Markov chains and processes, background that will be useful throughout the project. We also look at queuing theory and how it links to our model of cloud computing.

Chapter 3 provides an overview of the approach taken in decomposing the project and also explains some key concepts that will be required going forward.

In Chapter 4 we go through the process of creating the service evaluation tool that forms the backbone of our project. The design choices, challenges and more implementation specifics are described.

In Chapter 5 we describe the process of analysing the data which we derived from our service tool and we try to fit this data into probability distributions so that we can effectively model the cloud computing models into an M/M/1 queue.

Chapter 6 focuses on taking the approach of using Java Modelling Tools to try and create more complex queuing networks.

Chapter 7 tries to take a more analytical approach to try make some sense of the data, distributions and other calculations that we derived throughout the project. It particularly focuses on the performance and cost of cloud computing. We also describe what limitations we faced throughout the project.

Chapter 8 outlines the achievements of this project, improvements that could be undertaken and future work.

Chapter 2

Background

2.1 Cloud computing

The concept of cloud computing first derives from data centers. A data center together with the hardware and software is referred to as a cloud and when this is made available as a pay-as-you-go service we call this a public cloud. Private clouds are internal data centers that are owned by private organizations. In this project we focus on public clouds.

Cloud computing is the provision of computing capabilities, (e.g processing and storage) as a use-on-demand service over the internet [1].

Broadly cloud computing can be divided into two main categories. One is virtualization of software architecture and the other is on-demand computing power as a utility. In this project we will be focusing on the latter.

This concept of computing power as a utility provides us with a new paradigm of computing. In the past our computing power and digital storage capacity and limited by the constraints of the hardware that we physical own or have access to. We now have access to computing and storage resources on demand via the internet. Consoles and interfaces have been provided on top of these cloud services that help to manage the resources we are using with relative ease.

On demand computing power can also be divided broadly into different categories (each of which is a service model):

- Infrastructure as a Service (IaaS): Providers offer physical (often in the form of virtual) machines to customers. Additional resources such as file storage space, network access and pre-installed software is often included. (e.g. Amazon Web Service EC2)

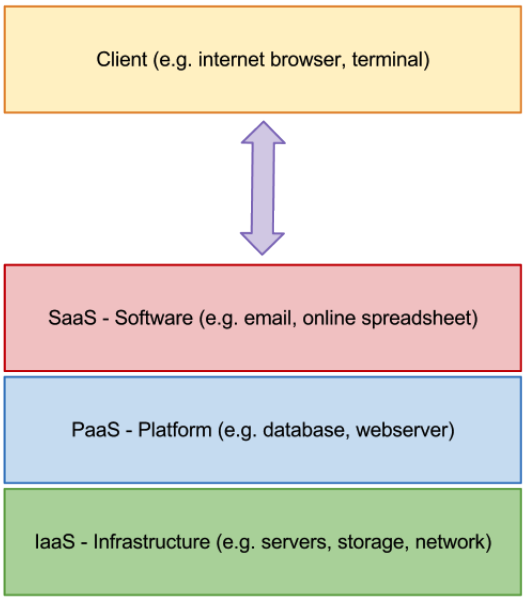


Figure 2.1: Layers of cloud services

- Platform as a Service (PaaS): Whole solution stacks are provided to clients(e.g. Heroku).
- Software as a service (SaaS) : Providers give users access to software via software clients (often lightweight). The actual software runs on the provider's servers (e.g. Google Docs, Dropbox).
- Network as a service (NaaS): Providers give clients access to network connectivity services (e.g. Amazon Elastic IP).
- Storage as a service (STaaS): Providers give customers access to (often file) storage facilities and infrastructure (RackSpace Cloud Files & Amazon S3). Useful to decouple the code and application file systems and space.

Companies with millions of users find cloud computing to be very useful as they do not need to invest in expensive data centers or customised infrastructure, the computing and storage services can be outsourced.

Scaling operations is very easy as customers can request the cloud service providers for more resources and the users often receive the extra capacity instantaneously to cope with the loads, this contrasts to the past where more physical servers had to be installed.

Cloud computing users can also have multiple instances of servers or other utilities running at the same time and they can shut down these instances immediately using the tools that are given by the cloud service providers. Cloud service providers also

provide different services such as storage facilities, virtual servers and load balancers which ofter work together efficiently.

Even though there are many clear benefits with cloud computing it does have its drawbacks as clients have to blindly trust cloud providers.

It's not uncommon for there to be outages, which can affect the services that these cloud reliant companies are providing to millions of users. There have been reported outages of Amazon Web Services and Google's cloud products [6].

The cloud has enabled the concept of computing to become a utility.

2.1.1 Amazon Web Services and Amazon Elastic Compute Cloud

Amazon.com Inc. provides cloud services which are referred to as Amazon Web Services (AWS) and their flagship product is Amazon Elastic Compute Cloud (Amazon EC2). AWS is one of the easiest to use and most popular cloud services available on the market.

A sample AWS architecture for a cloud application is shown in figure 2.2.

Amazon EC2 is a web service that provides resizable computing capabilities in the cloud.

The most important functionality that EC2 provides is:

- We can choose a particular operating system (e.g. Ubuntu) and create an instance of it.
- We can choose parameters for the instance, namely the number of CPU cores and memory. Costs vary per the parameters.
- The geographic location of the instance can also be chosen.

After the setup of the instance is complete we can then remotely login via SSH [5] into the instance and start using it for our own utility. The whole software stack running on the instance can be customized from the kernel upwards.

We are billed per hour that the instance is alive, also known as instance-hours. More information about AWS is in appendix A.

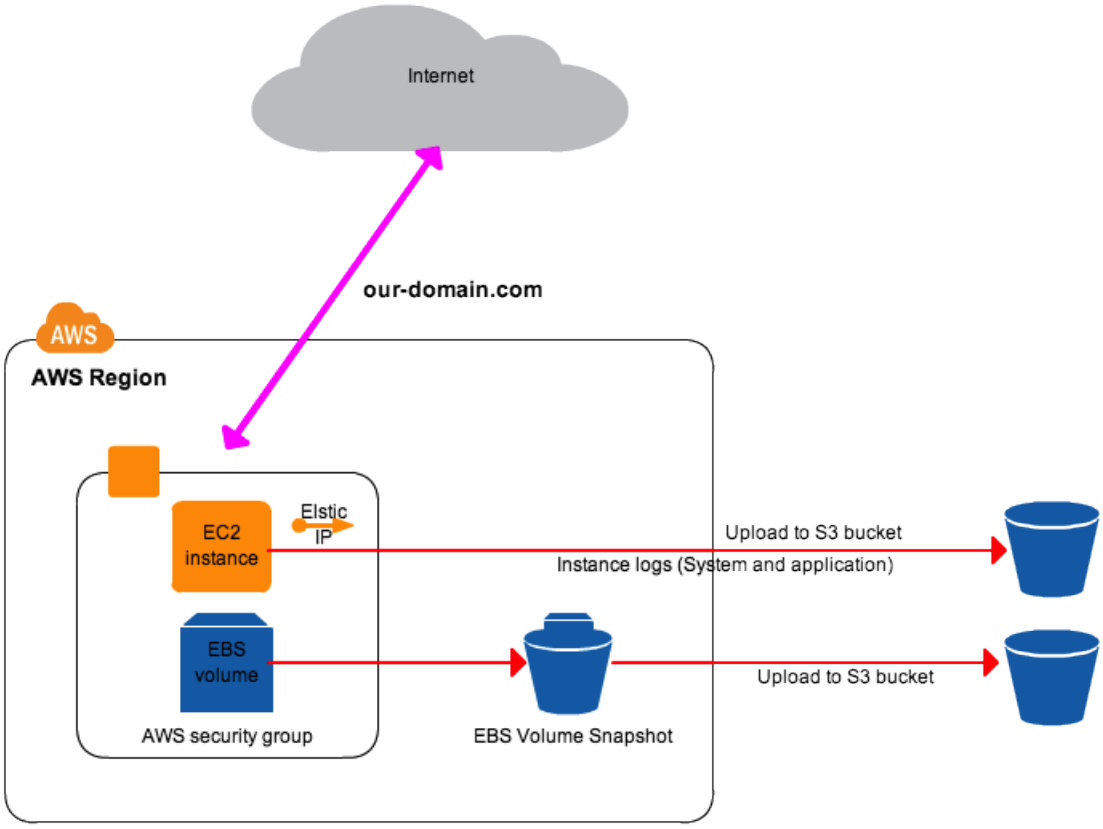


Figure 2.2: Sample AWS architecture

2.2 Mathematical Modeling

We will be modeling arrival and service times for particular processes using certain probability distributions.

2.2.1 Binomial Distribution

The binomial distribution [20] is the discrete probability distribution of n independent Bernoulli experiments consisting of k successes, where a Bernoulli experiment is an experiment where the event value can be from a set of two possible outcomes (usually success and failure).

The probability distribution is given by:

$$f(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k} \tag{2.1}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{2.2}$$

This equation represents the probability that k successes occur in a total of n trials, given the probability of success being p.

2.2.2 Poisson Distribution

The Poisson distribution is the discrete probability distribution which, given the average rate of an event (λ) conveys the number of occurrences of a certain event in a given time span. Events are independent of one another.

We have a rate parameter λ which represents the number of events occurring in a given time span, so for example the number of cars passing a point per hour. If we are approximating the binomial distribution to the Poisson distribution then we can use $\lambda = n \times p$.

If we were to model X = the number of cars passing per minute as a binomial distribution, using our rate parameter (λ =rate of number of cars passing per hour), and increasing the number of trials to 60 per hour, so every minute, we have a new rate parameter $\lambda/60$. We have our approximate probability of k successes as:

$$P(X = k) = \binom{60}{k} \left(\frac{\lambda}{60}\right)^k \left(1 - \frac{\lambda}{60}\right)^{n-k} \tag{2.3}$$

As we increase the number of trials, the above equation becomes a Poisson Distribution as shown below:

$$\lim_{x\to\infty}\left(1+\frac{a}{x}\right)^{x}=e^{a}$$
$$\lambda = n \times p$$
$$P(X=k)=\lim_{n\to\infty}\binom{n}{k}\left(\frac{\lambda}{n}\right)^{k}\left(1-\frac{\lambda}{n}\right)^{n-k}$$
$$P(X=k)=\lim_{n\to\infty}\frac{n!}{k!(n-k)!}.\frac{\lambda^{k}}{n^{k}}\left(1-\frac{\lambda}{n}\right)^{k}\left(1-\frac{\lambda}{n}\right)^{-k}$$

After some further simplification we get the probability distribution of the Poisson distribution as:

$$P(X=k)=\frac{\lambda^{k}e^{-\lambda}}{k!} \tag{2.4}$$

We can also calculate the distribution created by the sum of two Poisson random variables. Let X_1 and X_2 be independent Poisson random variables where X_i has a Poisson with rate parameter λ_i then $Z = X_1 + X_2$ has Poisson with rate parameter $\lambda_1 + \lambda_2$.

2.2.3 Exponential Distribution

We can derive the exponential distribution[21] from the Poisson distribution. Given a Poisson distribution with a rate parameter λ we can model the the distribution of the waiting times between successive events (with parameter $k = 0$) as:

$$D(x) = P(X \leq x)$$
$$D(x) = 1 - P(X > x)$$
$$D(x) = 1 - e^{-\lambda x}$$

The probability distribution function for an exponential distribution is:

$$P(x) = D'(x) = \lambda e^{-\lambda x} \tag{2.5}$$

2.2.4 Stochastic Processes and Poisson Processes

A **stochastic process [10] S** is a family of random variables $\{X_t \in \Omega \mid t \in T\}$, each defined on some **sample space** Ω (the same for each) for a parameter space T.

- T and Ω may be either discrete or continuous

- T is normally regarded as time (real time: continuous or every month / after job completion: discrete).
- Ω is the set of values each X_t may discrete or continuous values.

The Poisson process is a renewal process with renewal period (inter-arrival time) having cumulative distribution function F and probability density function (pdf) f

$$F(x) = P(X \leq x) = 1 - e^{-\lambda x}$$
$$f(x) = F'(x) = \lambda e^{-\lambda x}$$

where λ is the parameter of the Poisson process.

The memoryless property of a probability distribution is defined as follows for a random variable:

$$P(X \leq t + s \mid X > s) = P(X \leq s) \; \forall t, s \geq 0 \tag{2.6}$$

This means that the probability of an event happening in an interval is independent of the events that took place before it.

2.2.5 Markov Chains and Markov Processes

If we have a process that is in a state S_1 at time t, it's state S_2 at time $t + s$ has a probability distribution that is a function of s only, so state S_1 does not contribute to it (i.e. it is memoryless). This is called the Markov property (MP).

Stochastic processes with discrete parameter spaces and that satisfy MP are called Markov chains. Stochastic processes with continuous parameter spaces that satisfy MP are called Markov Processes.

Markov Chains

To describe Markov chains, we have a set of states $S = \{s_1, s_2, \ldots, s_r\}$ and a process that starts in one of these states moving progressively from one state to another. If we are currently in state s_i the move to state s_j is denoted by a probability q_{ij}. This probability does not depend on what the state was before s_i. Markov chains have a transition probability matrix which describe these probabilities:

$$Q = \begin{bmatrix} q_{00} & \cdots & q_{0i} \\ \vdots & \ddots & \vdots \\ q_{i0} & \cdots & q_{ii} \end{bmatrix}$$

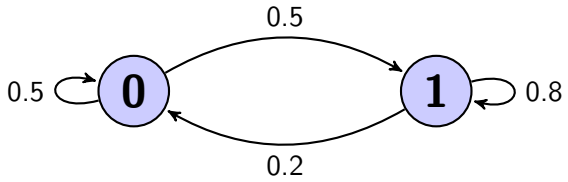
All the rows sum to one. This matrix describes the following probability:

$$q_{ij}(n) = P(X_{n+1} = j \mid X_n = i)\ \forall i, j, \forall n \geq 0$$

For example we can have:

$$Q = \begin{pmatrix} 0.5 & 0.5 \\ 0.2 & 0.8 \end{pmatrix}$$

represented diagrammatically as:



Continuous-time Markov chains (CTMC)

Continuous-time Markov chains can be described as a discrete time Markov chain in which transitions can happen at any time. The time spent in each state is non-negative and has an exponential distribution.

Markov Processes

For Markov processes we have a continuous time parameter space and a discrete state space:

$$X = \{X_t \mid t \geq 0\}\ X_t \in \Omega$$

where Ω is a countable set. We have a similar state transition property q_{ij}, the Markov process is uniquely determined by the products:

$$a_{ij} = \mu_i q_{ij} \quad \text{where } i \neq j$$

μ_i is the rate out of state i and q_{ij} is the probability of selecting j next.

The a_{ij} are the generators of the Markov process. Let $a_{ii} = -\mu_i$, we get the generator matrix for a Markov process:

$$A = \begin{bmatrix} a_{00} & \dots & a_{0i} \\ \vdots & \ddots & \vdots \\ a_{i0} & \dots & a_{ii} \end{bmatrix}$$

It follows that:

$$\sum_{j \in S} a_{ij} = 0.$$

An example of a Markov process is a Poisson process:

$$a_{ij} = \begin{cases} \lambda, & \text{if } j = i + 1. \\ 0, & \text{if } j \neq i, i + 1. \\ & \text{not defined if } j = 1. \end{cases}$$

as

$$\underbrace{P\left(\text{arrival in } (t, t+h)\right)}_{i=i+1} = \overbrace{\lambda h}^{a_{i,i+1}h} + o(h)$$

2.2.6 Queuing Theory

Kendall's Notation

Kendall's notation [13] is used to describe a queuing node using three factors, $A/S/c$ where:

- A denotes the arrival process
- S denotes the service time distribution
- c denotes the number of servers.

For example if we have an $M/M/1$ queue this means we have a Markov arrival process, a service time distribution as a Markov process and 1 server.

Single Server Queues

A Markov process with a state space $\{0, 1, \dots\}$ is called a birth-death process if the only non-zero transition probabilities are $a_{i,i+1}$ and $a_{i+1,i} (i \geq 0)$, representing births and deaths respectively.

The single server queue (SSQ) model consists of:

- a Poisson arrival process, rate λ
- a queue which arriving tasks join
- a server that processes tasks in the queue in a certain order (FIFO for example) and has exponentially distributed service time (parameter μ)
- the state is given by the queue length (including task being served)

Figure 2.3: M/M/1 queuing node

Figure 2.3 is is a diagram of an M/M/1 queue. The queue is a stochastic process, has a single server where the arrivals are determined by a Poisson process and the job service times can be modeled using an exponential distribution. It's the most basic type of queuing model.

The server serves one customer at a time on a first-come-first-served (FCFS) basis. When the customer is served then they leave the system and the next customer is served. Time taken for the service node to serve a customer is known as the service time. The queue length buffer is of infinite length for waiting customers.

Response times can be calculated mathematically using the **Random Observer Property**. The state of a system at equilibrium seen by an arrival of a Poisson process has the same distribution as that seen by an observer at a random instant.

The utilisation of a queue ρ is defined by:

$$\rho = \frac{\lambda}{\mu} \tag{2.7}$$

and for a queue to be stable, we have the condition:

$$\rho < 1 \tag{2.8}$$

The queue length is given by j. Response time = residual service time of task in service (if j $\geq$ 0) + j service times, (each task is an independent and identically distributed variable - i.i.d). The mean response time is given by:

$$W = \sum p_j(j+1)\mu^{-1} = \left(1 + \frac{\rho}{1-\rho}\right)\mu^{-1} = \frac{1}{\mu - \lambda} \tag{2.9}$$

and the mean queuing time is given by:

$$W_Q = W - \mu^{-1} = L\mu^{-1} = \frac{\rho}{\mu - \lambda} \tag{2.10}$$

Burke's Theorem

Burke's Theorem [4] states that if we have a queue in steady state with a Poisson arrival process with rate parameter λ then:

1. The departure process is a Poisson process with rate parameter λ.
2. At a time t the state of the queue is not dependent on departure process before time t.

Little's Law

Little's Law [17] states that the average number of customers in the queuing system L is equal to the rate with which customers arrive $\lambda \times$ the average amount of time a customer spends in the system (mean waiting time) W, so:

$$L = \lambda \times W \tag{2.11}$$

Open Queueing Networks (OQN)

An Open Queuing Network [2] is a network where customers can arrive into the network from an external source and there can also be external departures from the queuing network.

This is useful in our model of cloud computing as customers arrive and leave the system from an external source.

For a network to be stable there it must reach a steady state. If it's unstable than at least one buffer will have unbounded growth. For a stable network the arrival and departure flow must be greater than the arrival rate, so we get the stability equation as stated above for each node: $\forall i, \rho_i = \left(\frac{\lambda_i}{\mu_i}\right) < 1$.

For a OQN we have traffic equations [11] that are used to calculate arrival rates λ_i. For a particular node i we have the departure rate at i equals the arrival rate at the same queue so:

$$\lambda = \gamma_i + \sum_{j=1}^{N} \lambda_j q_{ji} : i = 1, 2, \ldots, N$$

To represent the solutions for our queuing network we are dealing with a system of linear equations, so we can define a vector representation as:

$$\begin{aligned}
\overrightarrow{\lambda} &= (\lambda_1, \lambda_2, \ldots, \lambda_N) \\
\overrightarrow{\gamma} &= (\gamma_1, \gamma_2, \ldots, \gamma_N) \\
\overrightarrow{\lambda} &= \overrightarrow{\gamma} + \overrightarrow{\lambda} Q \\
\overrightarrow{\lambda}(I - Q) &= \overrightarrow{\gamma} \\
\overrightarrow{\lambda} &= \overrightarrow{\gamma}(I - Q)^{-1}
\end{aligned}$$

We can define the state of the network as being a tuple $\{(n_1, n_2, \ldots, n_N), n_i \geq 0\}$, where n_i is the current length of the queue at i.

An important result called **Jackson's Theorem** which solves the OQN Markov process. If the OQN is stable so that $\forall i : \rho_i < 1$ then the product-form solution of the model is given by:

$$\mathbb{P}(n_1, n_2, \ldots, n_N) = \prod_{i=1}^{N} (1 - \rho_i)\rho_i^{n_i} \tag{2.12}$$

This solution implies that each node can be reasoned about as a $M/M/1$ queue in isolation so, we can retrieve the mean number of jobs in a system:

$$\begin{aligned}
L_i &= \frac{\rho_i}{1 - \rho_i} \\
L &= \sum_{i=1}^{N} \frac{\rho_i}{1 - \rho_i}
\end{aligned}$$

Applying Little's Law (see 2.2.6) to a OQN gives us: $L_i = \lambda_i W_i$, so the mean waiting time at the queue i is:

$$W_i = \frac{\mu_i^{-1}}{1 - \rho_i} \tag{2.13}$$

If we have v_i representing the average number of times a job visits a node i while in the network and if γ is the total arrival rate into the network, $\gamma = \sum_i \gamma_i$ then $v_i = \frac{\lambda_i}{\gamma}$, so we obtain the visits as:

$$v_i = \frac{\gamma_i}{\gamma} + \sum_{j=1}^{N} v_j q_{ji} \tag{2.14}$$

We now define the visit vector $\overrightarrow{v} = \overrightarrow{\lambda}/\gamma$, and we can get the network mean response time as:

$$W = \sum_{i=1}^{N} v_i W_i \tag{2.15}$$

If we apply Little's Law to the whole network we get: $L = \gamma W$, so the mean response time can be written as:

$$W = \frac{L}{\gamma} = \frac{1}{\gamma} \sum_{i=1}^{N} \frac{\rho_i}{1 - \rho_i} \tag{2.16}$$

Modeling cloud computing components

For the sake of simplicity in this project we will be modeling different cloud computing components as an $M/M/1$ queue. For our response times we will have:

$$T(\lambda) = T_0 + \frac{1}{\mu + \lambda} \tag{2.17}$$

T_0 is the time taken when there is only one request in the queue and also includes other delays that may be not related to the response time for this particular component.

Maximum Likelihood Estimate (MLE)

Likelihood enables us to see how likely a particular distribution is given our particular data set. It is the product of individual probabilities for each data sample.

Our observed data points are represented as: $x = \{x_1, x_2, \ldots, x_n\}$. We must choose a value of θ for which the probability of the observed data x is maximised. The negative logarithm of the likelihood function is called an error function and maximising the likelihood is the same as minimising the error function, so likelihood is defined by:

$$\mathcal{L}\left(\theta \mid x\right) = p\left(x \mid \theta\right) = \prod_{i=1}^{n} p(x_i \mid \theta) \tag{2.18}$$

Essentially we are estimating the parameter values which maximises the likelihood function, so:

$$\theta_{MLE} = argmax\ \mathcal{L}(\theta \mid x) \tag{2.19}$$

The value of θ_{MLE} is known as the maximum likelihood estimate (MLE). However we consider the log-likelihood function to be more convenient than the likelihood function:

$$\ell(\theta \mid x) = \log\{L(\theta \mid x)\}$$

$\log(x)$ is a monotonically increasing function, so maximising ℓ maximises L. The log likelihood function can be written as:

$$\ell(\theta \mid x) = \sum_{i=1}^{n} \log\left\{p_{x|\theta}(x_i)\right\} \tag{2.20}$$

Solving for:

$$\frac{\partial}{\partial\theta}\ell(\theta) = 0$$

yields the Maximum Likelihood Estimate if:

$$\frac{\partial^2}{\partial\theta^2}\ell(\theta) < 0$$

Maximum Likelihood Estimate for an exponential distribution

The MLE for an exponential distribution is given by:

$$\mathcal{L}(\lambda) = \prod_{i=1}^{n} \lambda e^{-\lambda x_i} = \lambda^n e^{\left(-\lambda \sum_{i=1}^{n} x_i\right)} = \lambda^n e^{-\lambda n \bar{x}}$$

The derivative of the log-likelihood is:

$$\frac{d}{d\lambda} \ln \mathcal{L}(\lambda) = \frac{d}{d\lambda}(n \ln(\lambda) - \lambda n \bar{x}) = \frac{n}{\lambda} - n\bar{x}$$

This then gives us an MLE parameter for the exponential distribution:

$$\lambda_{MLE} = \frac{1}{\bar{x}} \tag{2.21}$$

2.2.7 Tools

Apache Benchmark (ab)

Apache Benchmark (ab) [9] is a tool used for benchmarking a HTTP server. It particularly helps to show how many requests per second a HTTP server is capable of serving.

Java Modelling Tools (JMT)

Java Modelling Tools [3] is open source suite for performance evaluation, capacity planning and modeling of computer systems. Lots of state-of-the-art algorithms are implemented to solve queuing network models. We can define models through a wizard or with a graphical friendly user interface.

There are lots of sub-tools avaliable to solving different problems in the area of queuing theory however the tool that is of main interest to us is JSIM which is a discrete-event simulator for the analysis of queueing networks.

We can simply input our queuing network model graphically together with the arrival and service time parameters and the tool is able to provide performance indices such as throughputs, utilizations, response times and queue lengths. A screenshot of the tool is shown in figure 2.4.

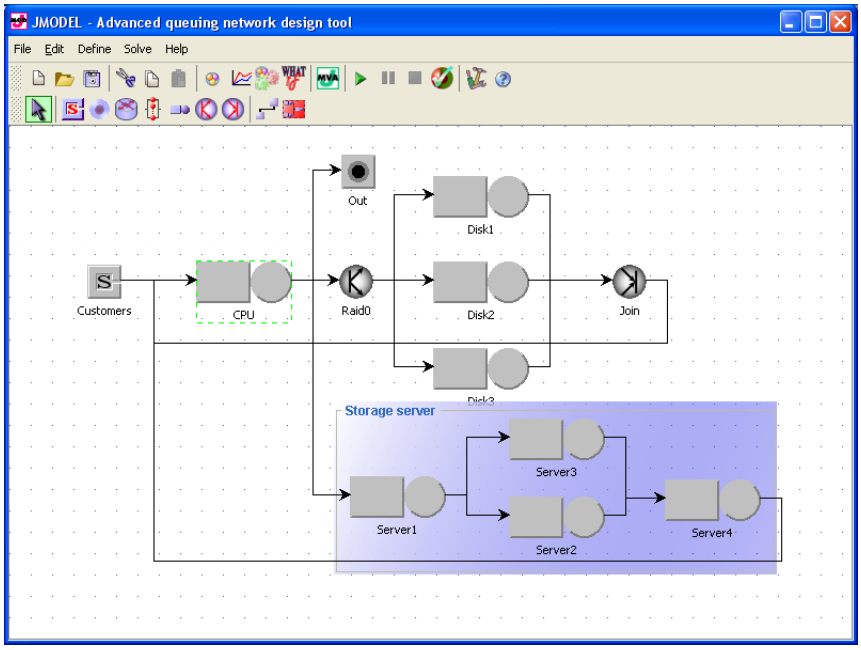


Figure 2.4: Screenshot of the JSIM Tool (from JMT)

2.2.8 Modeling cloud service usage

Agner Karup Erlang worked at a telephone exchange and he was one of the first to suggest modeling the telephone exchange using queuing networks [8]. He came up with the idea that if calls were made at random they follow a Poisson distribution

As far as the implementation for this project is concerned we want to model customers who interact with cloud services as open queuing networks.

We will be using queuing theory to model customer arrival for cloud computing services in this project and would like to evaluate how different cloud services react to different inter-arrival times.

Useful information that we can derive from the cloud providers is the service time and also the system response time which is the time a customer takes in the system. It would be interesting to model different bursts of traffic and to simulate a scaling problem for software systems.

Chapter 3

Overview and Architecture

The code for the project is primarily written in Python, an object-oriented, dynamically typed language.

There are three main phases for this project. The first one is developing a service time evaluation tool to test response times for different individual cloud components.

The second phase is to derive parameters for these components so that they effectively model $M/M/1$ queues.

The third phase is to use these parameter values in a queuing network tool to try and build more complex cloud computing models. We will be using the Java Modelling Tool for this.

We will be focussing on modeling two main cloud computing components for this project:

- Web servers
- Data storage (databases)

3.1 Data storage

Most cloud computing applications and websites have some sort of data storage aspect as they store some of their users data or generate content that they will somehow use in the future.

The function of reading and writing data from a datastore forms a bulky part of any

application. In our service evaluation tool we aim to gain service times for both the reading and writing of data.

In order to be consistent we are using JavaScript Object Notation (JSON) for data storage operations throughout this project. JSON [7] is a text based data-interchange format used to represent data structures and associative arrays.

3.2 Web servers and Hypertext Transfer Protocol Server (HTTP Server)

A web server is used to listen to requests that are coming into it via the internet, and take action either by replying or to take some internal action which can be in the same system (e.g change the value of a variable) or another system which it is connected to (e.g. write some value to a database).

HTTP is a protocol used by the World Wide Web. It defines how media and information is formatted and sent.

At its core HTTP has a request-response model that makes it ideal for a client-server architecture as the client sends the server a request and the server then responds to the client with either data or other meta-information regarding the response. Most modern browsers support the HTTP protocol and one of the most famous HTTP servers Apache HTTP Server (”httpd”) is used by more than 100 million websites [14] around the world, making this the one of the primary protocols of the internet.

In this project we will be getting response times from HTTP servers to keep our experiments as consistent as possible. In addition there are many tools that support the HTTP protocol. The request-response model also makes the calculation of server response times much easier.

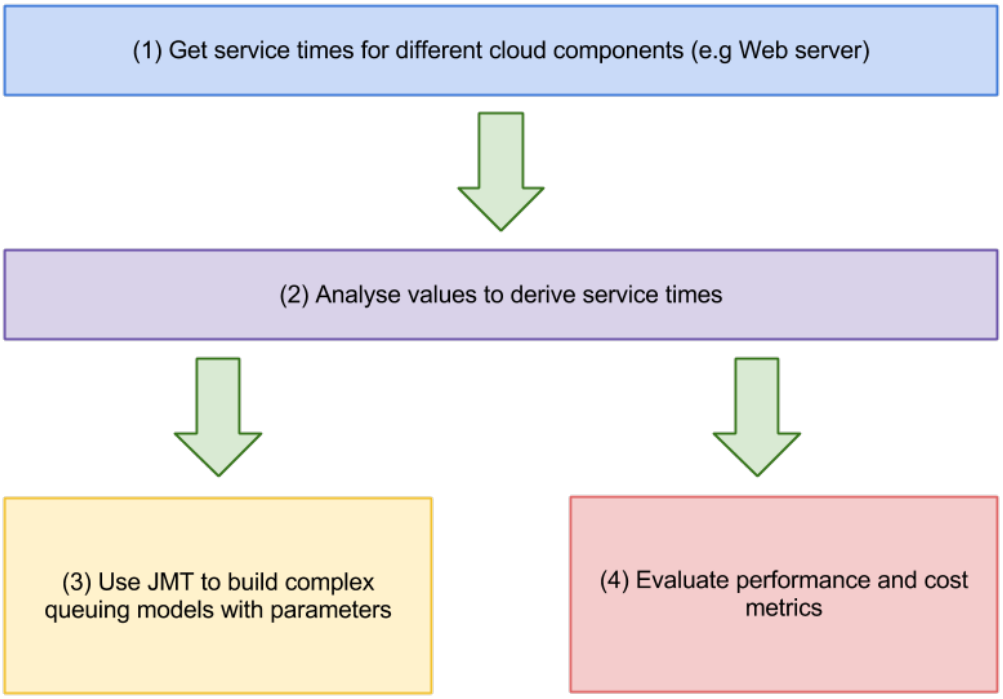


Figure 3.1: Workflow for the project

Chapter 4

Service time evaluation tool

The service time evaluation tool (STET) is a command line tool used to generate queries to components and retrieve response times for these queries. The response times are stored in a text file. As mentioned in Section 3 we will be focussing on retrieving web server and database component response times.

4.1 Generation of random data and Twitter

One of the first problems that had to be solved in this project is modeling the storage of data in a cloud service to effectively get some response times. The first barrier was automatic generation of data on the fly when running tests to write to a data store.

Cloud applications typically store data with certain properties:

- Data is stored in a structured format, as records - databases are ideal for storing this type of data.
- Each record has a fixed number of (option, value) tuples - the values have a specific type (e.g. integer or string).
- Each (option, value) tuple has a constraint on the size of the value, usually stemming from constraints on physical storage (e.g. integer values can be restricted to a physical 32-bit constraint).
- Some options can be allowed to have a null value.

To demonstrate the properties mentioned above, an example of typical types of data that a cloud service can store is user records where we store 3 (option, value) tuples -

name, age and name of pet (can be null or not null). The name and age fields can vary in length up to a limit and the name of a pet can also take a null value.

On top of these properties it is also likely that the size of each record will vary with a small standard deviation (for example in our example above where we were storing user records we can have variation in people's names length which consequently accounts for this standard deviation).

It's not very easy to generate structured data using the $random()$ function that is inbuilt into almost every programming language. Instead we decided that it would be good if we could use an API that can stream data that conforms to the above properties as we demand. We decided to use Twitter for this purpose.

4.1.1 Twitter

Twitter is a social networking service allowing users to follow each other and consequently receive updates from users they are following. These updates are in the form of Tweets [18] - a 140 character message posted on the social network. Tweets also contain metadata about the location of where the tweet was posted, user information and lots more fields.

Figure 4.1: A sample tweet

For our service evaluation tool we are using tweets for the data storage. Our data is coming from the Twitter Streaming API [19] that provides a certain fraction of all Tweets at a certain moment, together with all their meta data in JSON format.

Using Twitter data has helped to solve the problem of generating random data for storage.

4.2 Database service times

To analyse service times of databases using our tool we implemented support for two database systems, PostgreSQL and MongoDB. By using the service time tool we can

specify which database to connect to as well as how many reads and writes to conduct. The writes involve storing tweet meta data to the database and the reading involves querying the database and getting a result. Response times for both these operations is logged to a file.

An example usage of this tool is as follows:

```
> python Performance.py -d mongo -i 2000 -s 1000
```

This command will open a connection to MongoDB, insert 2000 tweets in the database and also perform 1000 read operations on the database. Response times for each action will be logged.

The response time for an action is calculated as follows:

```
...
before = DateTime.now()
# Perform some database action
Database.write(TweetData)
after = DateTime.now()
response_time = after - before
```

4.2.1 PostgreSQL

PostgreSQL (Postgres) is an open-source object-relational database management system (ORDBMS). It has inbuilt native datatypes used for storing data. For queries it uses Structured Query Language (SQL) which is based upon relational algebra and is specially designed for manipulation of structured data. Some sample SQL queries (which are actually used in the code base) are below:

```
CREATE TABLE tweets (
  id bigserial primary key,
  json tweet
);

SELECT * FROM tweets WHERE id = 12;
```

PostgreSQL also supports insertion of native JSON (see 3.1) objects that is ideal for Tweet objects that are coming in as JSON.

4.2.2 MongoDB

MongoDB is an open-source NoSQL document oriented database. It's a fairly new product and has left the convention of using the SQL language for data manipulation, it uses its own data manipulation language. MongoDB stores JSON (see 3.1) objects as opposed to raw datatypes. An example of a JSON object is:

```
{
  "name" : "Wayne"
  "favorite_language" : "Handel-C",
  "cars_owned" : ["Mercedes-Benz", "Ferrari"]
}
```

Insertion of JSON data to both PostgreSQL and MongoDB is very important as we are inserting similar types of data which makes the service time evaluation comparison between both databases fairer.

We are treating our database as an $M/M/1$ queue with an arrival rate from the Twitter Streaming API as an approximate Poisson distribution with parameter λ.

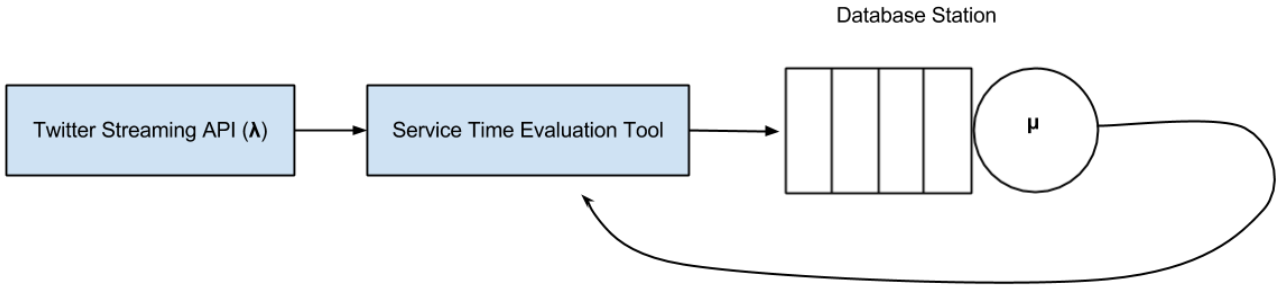


Figure 4.2: Modeling a database in the tool

4.3 Web server service times

As part of the service time evaluation tool we also wanted to test response times for web servers, and again to make the evaluation time comparison as fair as possible we created a web server that can be easily deployed to different cloud services and used used for testing.

There are two main types of methods are used between client-server communication, these methods are specified in the HTTP (see 3.2) specification:

- GET : requests data from a particular source

- POST : submits data for processing to a resource

There are other methods which are also in the HTTP specification (i.e. PUT and DELETE), however they are not as commonly used as GET and POST methods.

Similar to the database response time calculation, web server response time calculations are carried out as follows:

```
...
before = DateTime.now()
# Perform a web server query
HTTP_connection.request("GET", url)
request = HTTP_connection.get_response()
after = DateTime.now()
response_time = after - before
```

A sample usage of this tool is as follows:

```
> python Performance.py -u localhost -p 80 -g 3000 -x 2000 -n 30
```

This will connect to the server on the url localhost (u) with port 3000 (p) with 30 (n) concurrent connections and will perform 3000 (g) get requests and 2000 (x) post requests for each concurrent connection. The data for the post requests is passed in a configuration file. (See appendix for full usage options).

Our webserver is also modeled as an $M/M/1$ queue in the tool, and we can have n concurrent connections to the webserver (specified in the tool parameters) each with a Poisson distribution rate. The vector of parameters is $[\lambda_1, \lambda_2, \ldots, \lambda_N]$. We can use the result in 2.2.2 to calculate the Poisson arrival rate for the web server station queue as being $\lambda_z = \sum_{i=0}^{N} \lambda_i$.

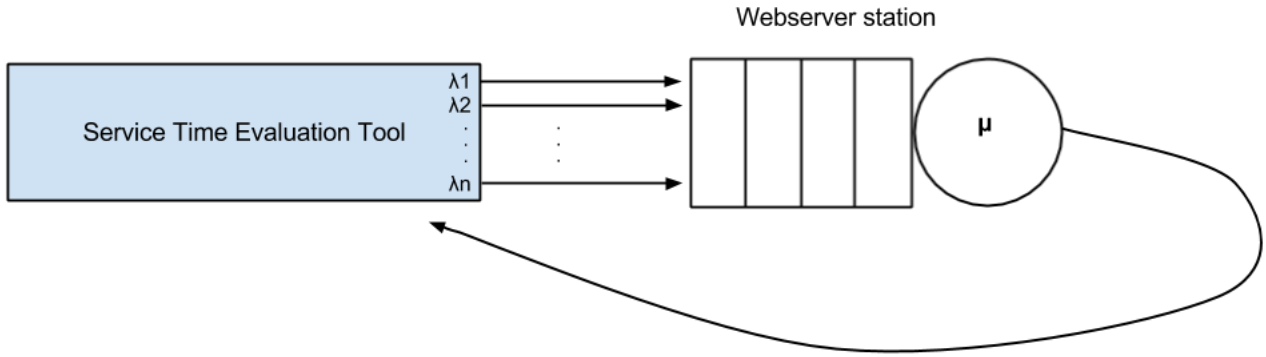


Figure 4.3: Modeling a webserver in the tool

4.4 Sample webserver for testing out tool

We implemented a simple web server to test out the service time tool on various platforms. The web server is written in Node.js [12] which is a platform designed to write non-blocking (asynchronous) and event based networking applications in the JavaScript language. We used a library called Express.js which makes it very easy to code a web server. The pseudocode in JavaScript callback function style [15] of our simple web server is as follows:

```
WebServer webserver;
webserver.listen_on_port(3000)

webserver.get('/', function(Request request, Response response){
  /*  Do some actions */

  /* Compose a response */
  send(response)
});

webserver.post('/posturl', function(Request request, Response response){
  post_data = request.parameters;
  /* Do something with post_data */

  /* Compose a response */
  send(response)
});
```

Node.js contains a built in web server so no additional software such as Apache HTTP server or Nginx is needed, making configuration to get the web server running effortless. The web server was deployed on various virtual machine instances, this will be described in depth in later chapters.

4.5 Architecture of the service time evaluation tool

As described above the service time tool developed is quite flexible as it carries out read and write operations to a database and also performs concurrent POST and GET requests to a specified web server to get response times. The overall architecture of the tool is depicted in figure 4.4 where the direction of arrows represent the flow of information.

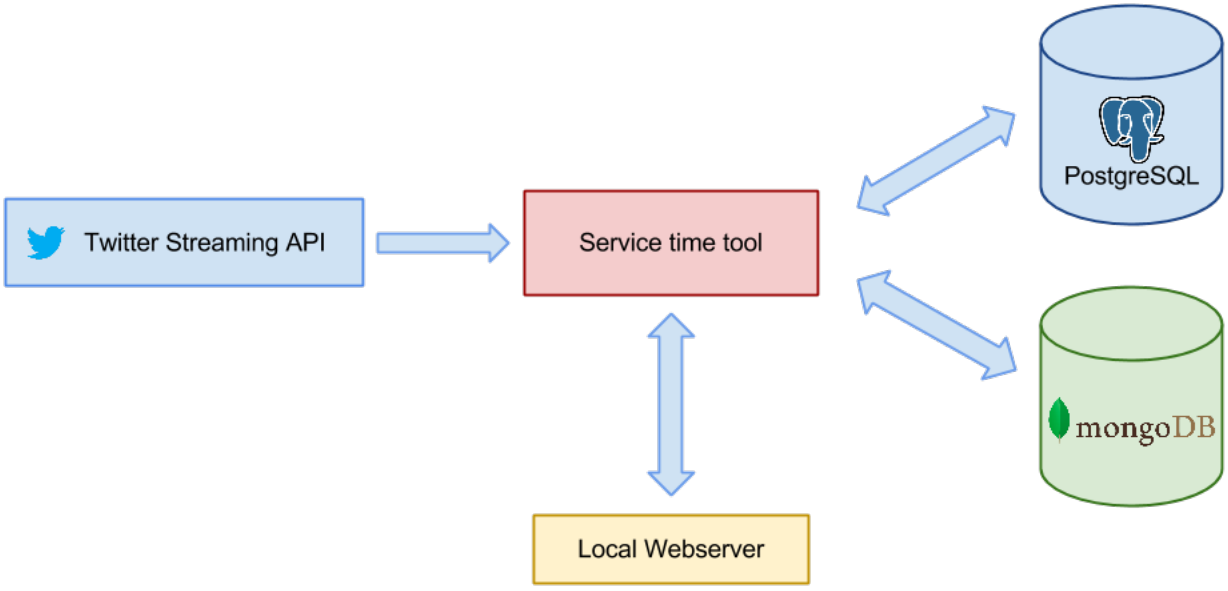


Figure 4.4: Architecture for the service evaluation tool

Chapter 5

Analysis of service times

5.1 Modeling Twitter Streaming API inter-arrival times

In order to model what probability distribution that the Twitter Streaming API resembles best we took the inter-arrival times between tweet arrivals from the Twitter API when no operations were being carried out on the Tweets, so just the streaming was ongoing, in pseudocode this calculation is represented as follows:

```
data = []
before = DateTime.now()
for tweet in TwitterAPIStream:
 now = DateTime.now()
 interarrival_time = now - before
 data += interarrival_time
 before = now
```

A sample of more than 8000 tweet arrival events was taken and the inter-arrival times (in seconds) are plotted as an empirical cumulative density graph in figure 5.1. The plot looks very similar to an exponential distribution and we can try to work out an estimate of the rate parameter using the maximum likelihood estimator (see 2.2.6).

For the exponential distribution the maximum likelihood estimate is given by $\lambda_{MLE} = \frac{1}{\bar{x}}$ (described in section 2.2.6).

For our data points that yields a rate parameter $\lambda_{MLE} = 492.872$.

As we have modeled the inter-arrival time between successive tweets as an exponential distribution we can conversely say that the arrival process of tweets follows a Poisson distribution.

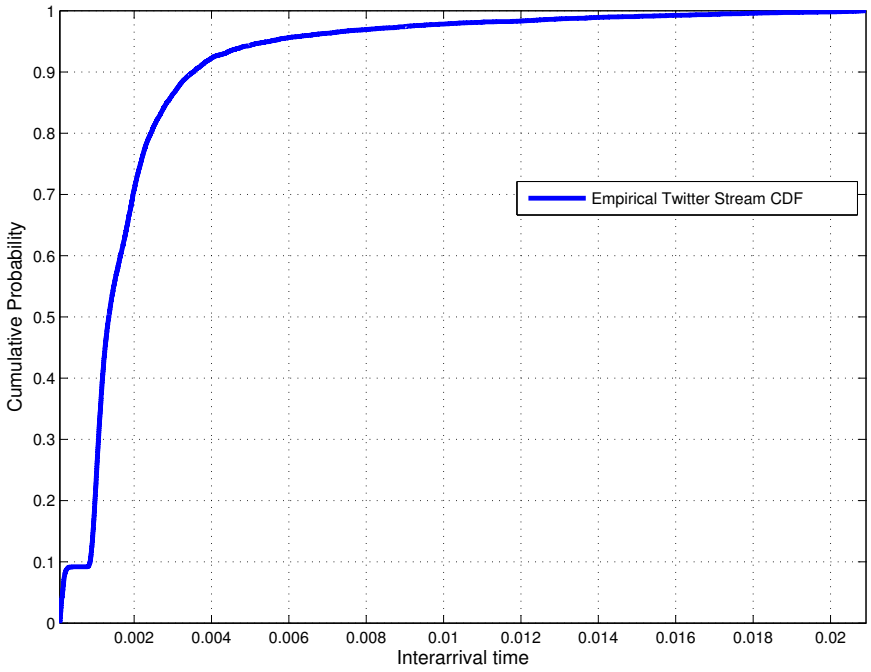


Figure 5.1: Empirical Cumulative Density Plot for inter-arrival time between Tweets

5.2 Running simulations on Amazon EC2 instances

We are going to run our tests on 2 types of Amazon EC2 (described in 2.1.1) instances (**micro** and **extra-large**). Technical specifications of the instances are listed in table 5.1. All the instances that we use have a 64-bit architecture. In addition the cost of the EC2 instances are listed in table 5.2.

The vCPU column stands for number of virtual CPUs and ECU stands for Elastic Compute Unit [16] that is the equivalent of CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

We must also note that all our response time data has a **minimum value** that is non-zero, which can be thought to approximately represent the T_0 value in equation 2.17.

All probability density graphs in this section have been shifted for the data set x by a factor of $\min x$, as it is easier to fit this to a probability distribution. The reason that this shift has taken place is because of the T_0 term in equation 2.17. All PDF graphs contain a line representing the probability density function derived from MLE for the data points.

Instance Type	vCPU	ECU	Memory (GB)	Storage (GB)	Network Performance
Micro	1	upto 2	0.615	Variable	Low
Small	1	1	1.7	1 x 160	Low
Medium	1	2	3.75	1 x 410	Moderate
Large	2	4	7.5	2 x 420	Moderate
Extra-large	4	8	15	4 x 420	High

Table 5.1: EC2 instance specifications.

Instance Type	Instance-hour cost ($)
Micro	$0.025
Small	$0.065
Medium	$0.130
Large	$0.260
Extra-large	$0.520

Table 5.2: EC2 instance pricing per hour.

5.3 MongoDB service time

To get the service time calculations of a MongoDB database we carried out 5000 reads and 5000 writes to the database. As stated above we used tweet data for our write operations. A histogram plot of this experiment on an EC2 extra-large instance is shown in figure 5.2.

Clearly we can see that there are two probability distributions, one that is centered around the 0 to 1×10^{-4} second range and another that is centered around the 3 to 4×10^{-4} second range and these correspond to the read and write distributions. There is no point in treating the read and write operations as being from the same distribution and trying to find a suitable distribution with a parameter as they have different parameters.

We can separate both the read and write data, if we carry out some additional processing then we can create a probability density estimation for the read and write operations. We can get an estimated probability density graph that is shown in figure 5.3 for MongoDB write operations and in figure 5.4 for read operations.

To calculate the parameter for the service time for components we can use the maximum likelihood estimate (described in 2.2.6). We will also be assuming that the distribution is **exponentially** distributed so that we can model an $M/M/1$ queue.

Applying the formula described in section 2.2.6 we compute the parameter for the exponential distribution given the results in table 5.3 for the mean of the data. Additional results in table C.2.

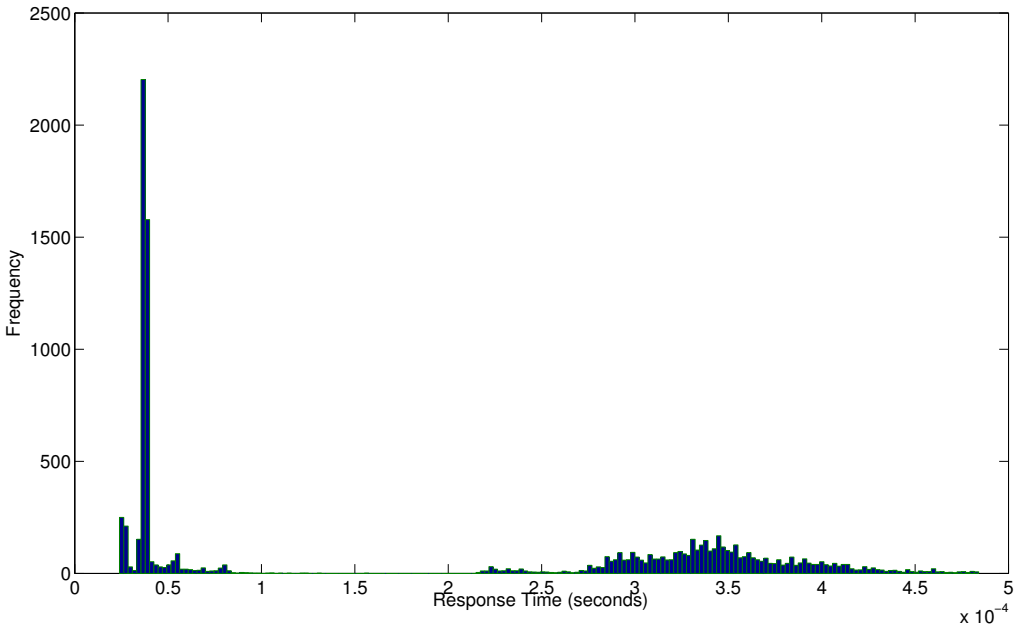


Figure 5.2: Response time frequencies for 5000 reads and 5000 writes to MongoDB on an extra-large EC2 instance

	Read operations	Write operations
Micro Instance ($\bar{x}$)	2.3091×10^{-5}	2.8500×10^{-4}
Extra Large Instance ($\bar{x}$)	1.4727×10^{-5}	1.5539×10^{-4}

Table 5.3: Mean values for MongoDB reads and writes

5.4 PostgreSQL service time

We carried out similar tests to above for PostgresSQL involving 5000 reads and 5000 writes to the database for both micro and extra-large EC2 instances. Our results for 5000 reads and 5000 writes are plotted in a histogram in figure 5.5.

Once again there seem to be two clear distributions and separating them we can build a an estimated probability distribution function to get figure 5.6 representing the PDF for reads and figure 5.7 representing the PDF for writes.

We conducted a similar calculation to above, to get the service time by using the log likelihood estimate. Results shown in table C.2.

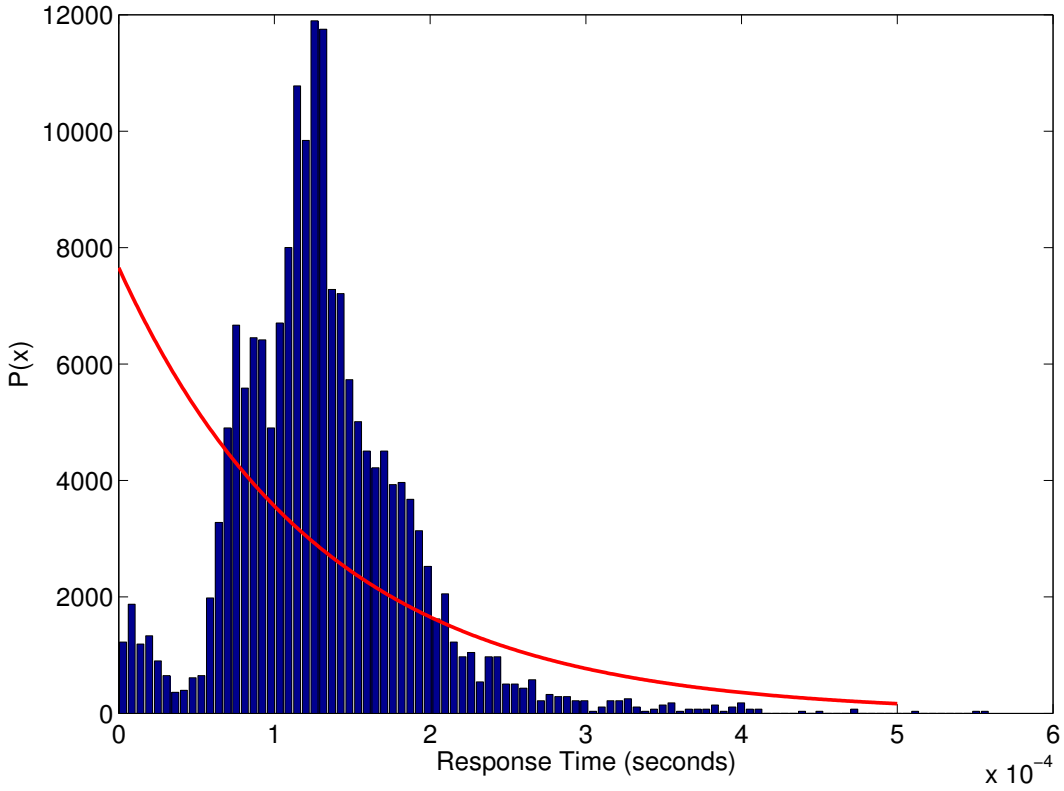


Figure 5.3: PDF for writes on MongoDB on a extra-large EC2 instance

5.5 Web server service time

To get the service time parameters of a HTTP web server we carried out 5000 POST and 5000 GET requests to the web server. The web server used was a simple Node.js implementation as described in section 4.4. We conducted the tests on both a **micro** and **extra-large** EC2 instance.

To inspect our data initially we conducted a mix of POST and GET requests to and figure 5.8 shows a histogram of our service times for these requests (to a micro EC2 instance). There were 10000 requests, consisting of 5000 POST and 5000 GET requests.

The data shows that we might have two independent distributions as there seems to be more density of data points around the 0 to 0.005 second range and another dense region of data points around the 0.0075 and 0.015 range. This suggests that GET and POST requests have different service times.

We consequently decomposed the data (for the micro EC2 instance results) to investigate and a plot of the estimated probability density function of GET requests is shown in figure 5.9 and for POST requests this is shown in figure 5.10.

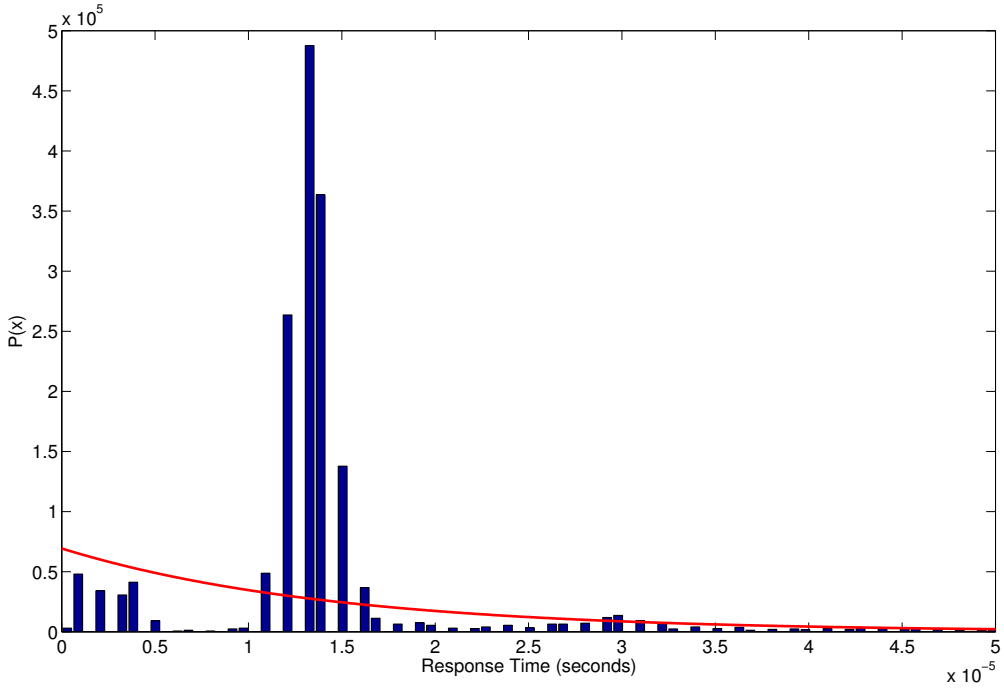


Figure 5.4: PDF for reads on MongoDB on a extra-large EC2 instance

Similar to the calculations of the database components we tried to fit the service time data to an exponential distribution using the log-likelihood estimate so that we can investigate the web server as an $M/M/1$ queue. The results of means for the POST and GET requests is shown in table 5.4, we can use this to calculate the service time parameters shown in table C.2.

	GET request	POST request
Micro Instance ($\bar{x}$)	2.3091×10^{-5}	2.8500×10^{-4}
Extra Large Instance ($\bar{x}$)	1.4727×10^{-5}	1.5539×10^{-4}

Table 5.4: Mean values for (Node.js) web server requests

All the results for the service times are listed in Appendix C.

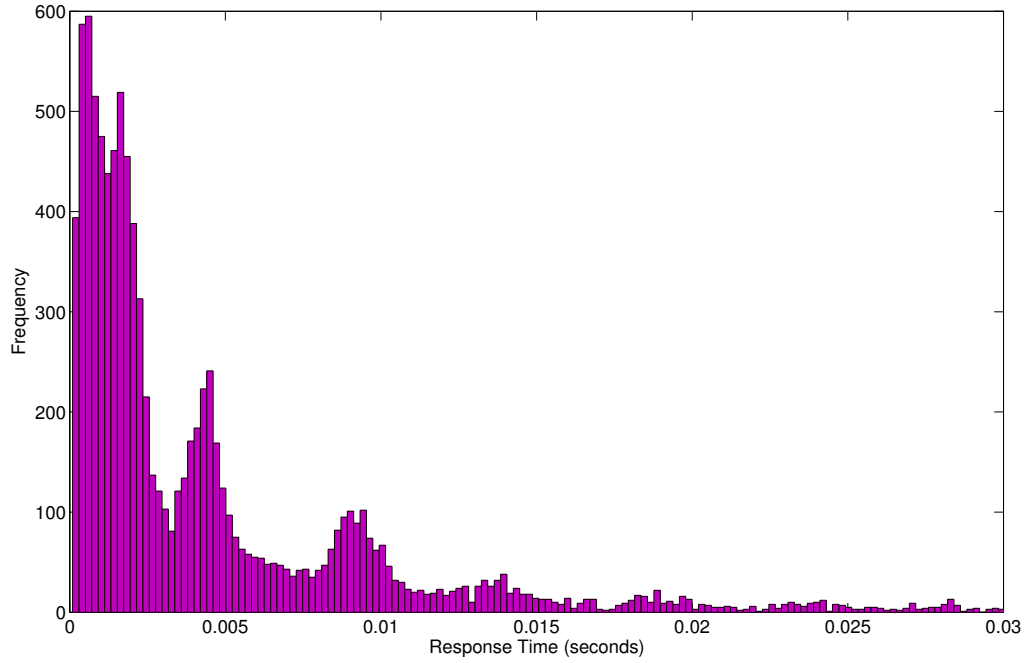


Figure 5.5: Response time frequencies for 5000 read and 5000 write requests to Postgres on a micro EC2 instance

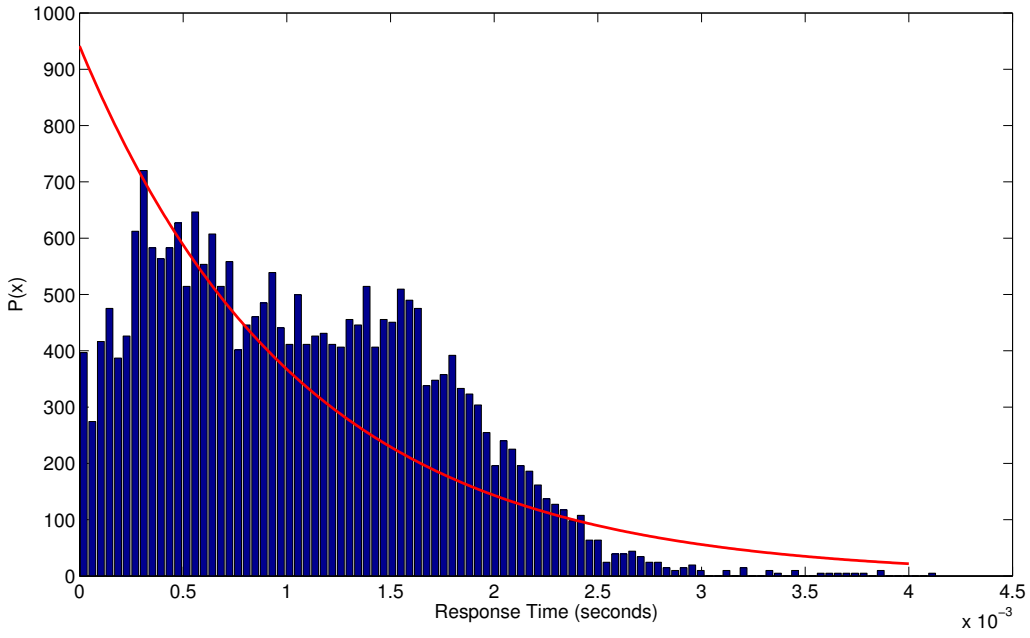


Figure 5.6: PDF for reads on a Postgres on a micro EC2 instance

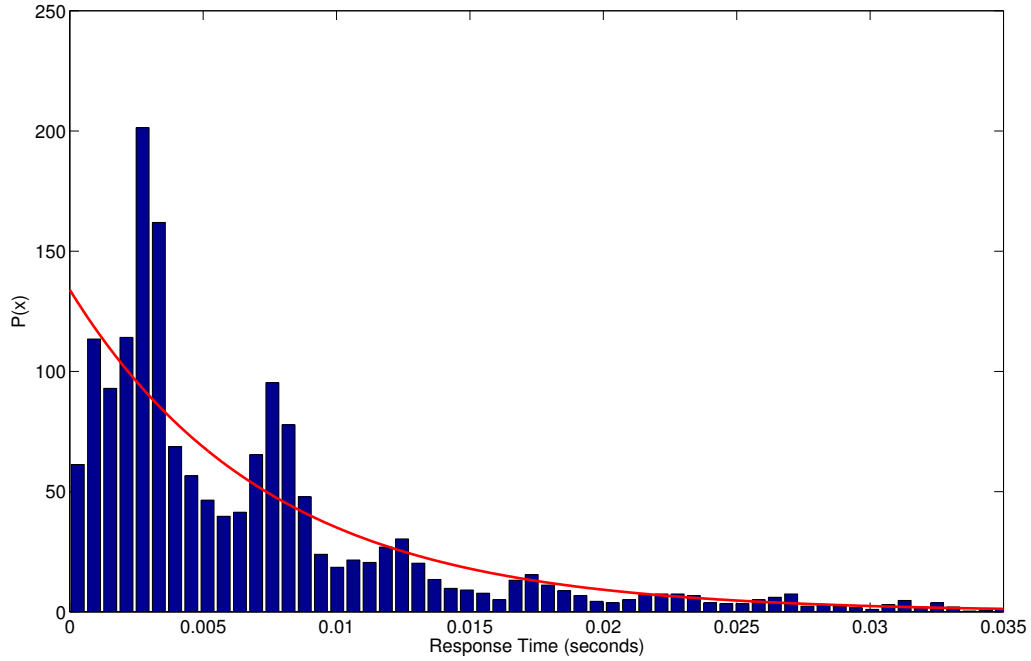


Figure 5.7: PDF for writes to Postgres on a micro EC2 instance

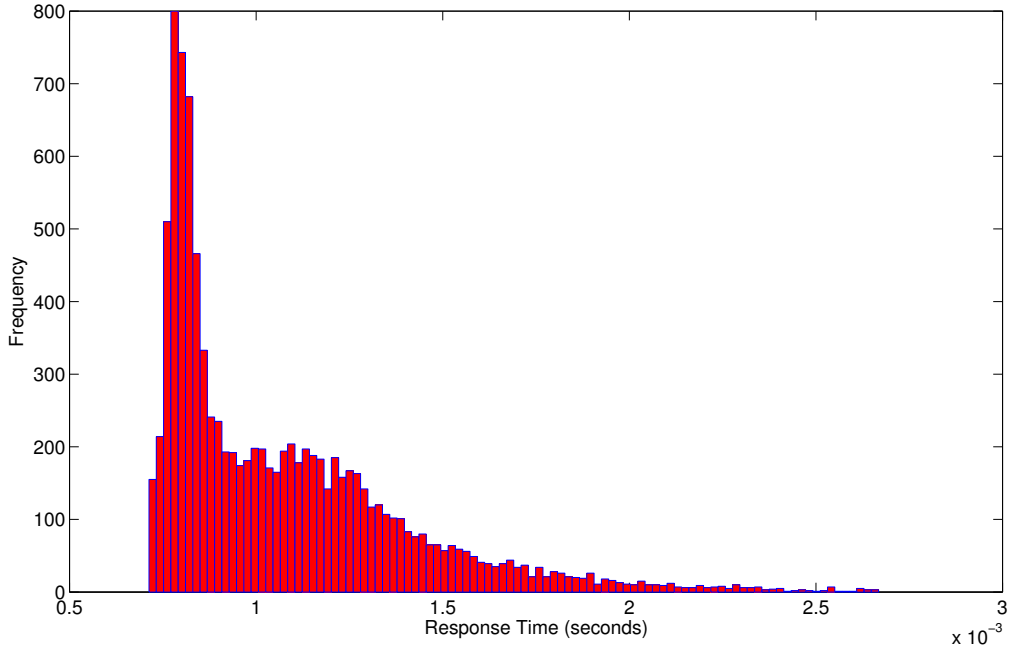


Figure 5.8: Response time frequencies for 5000 GET and 5000 POST requests on a micro EC2 instance

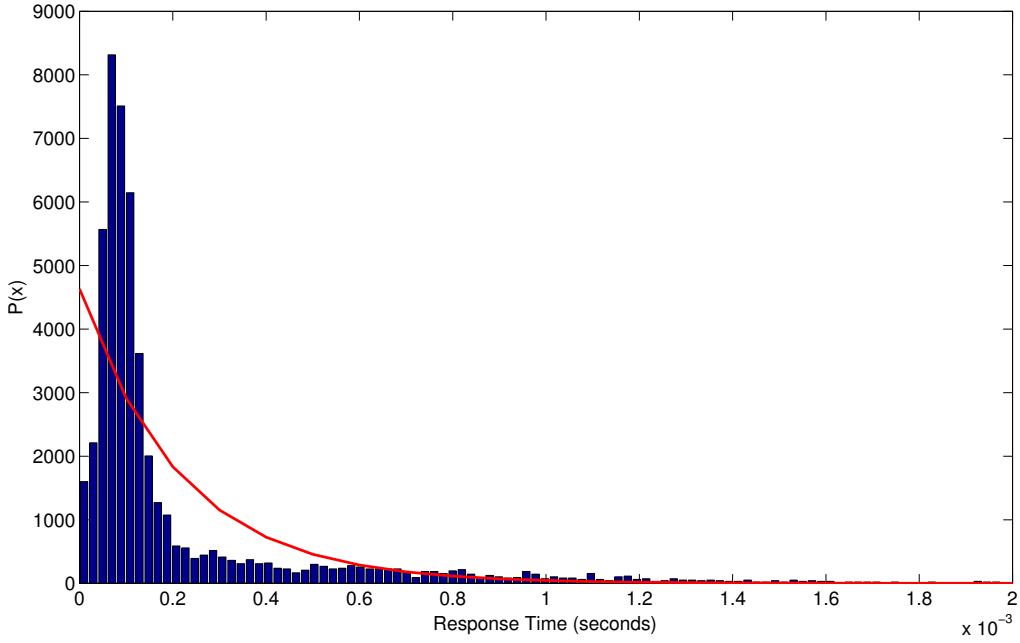


Figure 5.9: PDF for GET requests to a Node.js server on a micro EC2 instance

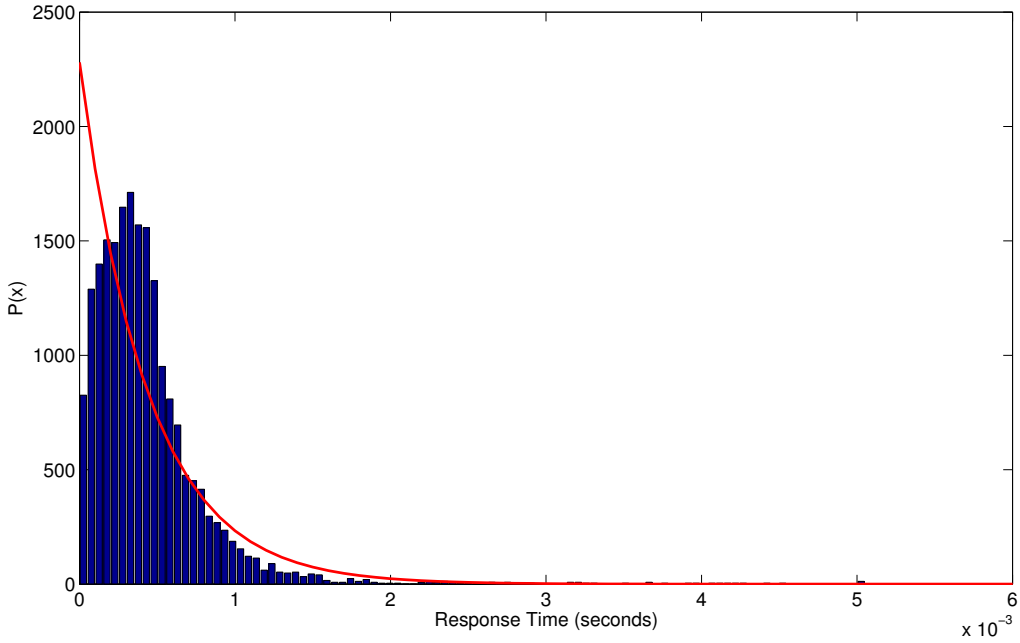


Figure 5.10: PDF for POST requests to a Node.js server on a micro EC2 instance

Chapter 6

Modeling using Java Modeling Tools

We wanted to carry out some analysis using the JMT tool mentioned in 2.2.7. Results for the analysis carried out with JMT are shown in appendix D.

For the parameters we used the micro-instance (Node.js) web server service times (for POST requests) and for the database we were using PostgreSQL (write operation) service times for micro and extra-large EC2 instances. This is a realistic cloud setup as applications usually send a HTTP POST request when they want to store some data to the cloud.

To test out the tool we inputted the parameters for a simple queuing network involving only one component. The setup is shown in figure 6.1.

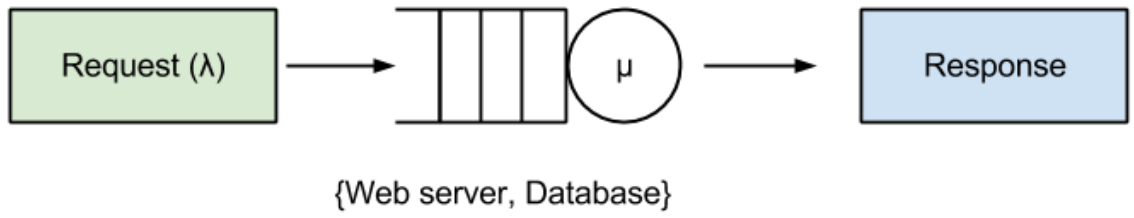


Figure 6.1: One component queuing network

After this we began by modeling a very simple cloud computing setup which is shown in figure 6.2.

We quickly saw that at a particular parameters for the incoming Poisson process (λ)

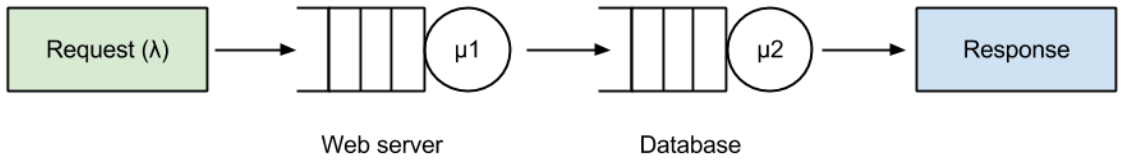


Figure 6.2: Simple cloud computing queuing model

we get a point where the system response time increases exponentially (see results in appendix D) and this is because of the bottlenecks of the database node (for our particular parameters). Consequently we investigated the effects of using different database configurations, an alternate database configuration is shown in figure 6.3.

These threshold values can be thought to represent the point when it become infeasible to use the system any more, given the traffic as the system response time for all the users slows down exponentially.

We investigated changing the database configurations as follows:

- 1 micro-instance web server connected to 1 micro instance database
- 1 micro-instance web server connected to 1 extra-large instance database
- 1 micro-instance web server connected to 2 micro instance databases
- 1 micro-instance web server connected to 4 micro instance databases
- 1 micro-instance web server connected to 10 micro instance databases

Results for the threshold value of (the arrival process) λ at which the system response time starts increasing exponentially are listed in table 6.1.

Web server config	**Database config**	**Threshold**
1 × micro	1 × micro	116
1 × micro	1 × extra-large	1160
1 × micro	2 × micro	235
1 × micro	4 × micro	380
1 × micro	10 × micro	1200

Table 6.1: λ values for service time threshold

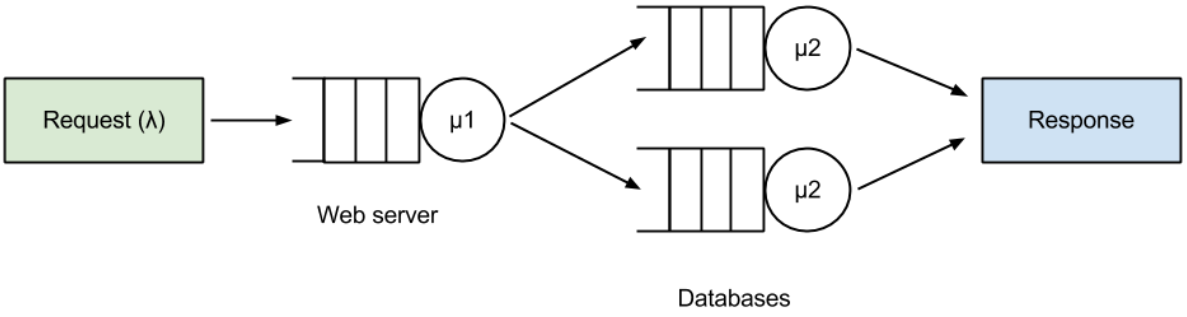


Figure 6.3: More complex cloud computing queuing model with two databases

Chapter 7

Evaluation

After we successfully evaluated the service times for the Node.js webserver, PostgreSQL and MongoDB databases for both micro and extra-large EC2 instances and different types of operations on these cloud components (i.e. POST/GET requests for the web server and read/write operations for the database) we now want to see what conclusions we can draw from our results.

One of the major learning points for this project was that when people refer to a cloud service as a single entity, it hardly represents the reality, as all these cloud services are in reality composed of much more complex individual components and hierarchies, and in order to be able to carry out any type of performance evaluation we must look at individual components.

Even when we break down the individual components we find that when we try to find some sort of model the response data does not give a clear model, and this is shown by the MongoDB response time histogram (figure 5.2). It's only when we drill down to individual operations on a component (such as a write to a database or a GET request to a web server) can you really start to build a coherent model of the component.

7.1 Performance vs Cost

It's quite obvious from our service time data (table C.2) that performance wise the extra-large EC2 instance outperforms the micro instance in every measure, however it's more interesting to calculate if the increase in cost is proportional to the increase in performance.

To compare the cost of using different EC2 instances we can calculate the cost per response, we use: μ in seconds $\times$ Cost of EC2 instance (per second).

Response time description	Ratio Extra-Large vs Micro
Postgres	
Read	12.0627
Write	2.0855
MongoDB	
Read	13.2626
Write	2.4096
Node.js web server	
GET	4.1911
POST	3.6523

Table 7.1: Cost per request ratio for EC2 program operations

After the calculation of the cost per response for micro and extra-large instances we can calculate the ratio of cost per response in extra-large instances vs cost per response in micro instances.

The results for these ratios is shown in table 7.1.

Analysing this table we can see that extra-large instances are more expensive than micro instances in every single response time cost than the micro instances. **So actually the increase in performance is not proportionally worth the cost**.

This particularly stands for database reads (MongoDB and Postgres) where the costs for extra-large instances are more than 12 times higher than for micro instances.

For the database writes the cost per request in an extra-large instance is more than double that than for a micro instance.

The cost of running a web server in an extra-large instance is more than 3.5 times higher than a micro instance (using the data for both POST and GET requests).

7.2 Further analysis using JMT

Even though all the evidence in 7.1 shows that an extra-large EC2 instance is more expensive given all our operations by at least double, what we can try to do is glue together some cheaper components to yield us similar performance benchmarks to the more expensive and better performing components.

Let's refer back to table 6.1, we can see that if we have an extra-large EC2 instance running Postgres then we have a threshold value of 1160. If we want to replicate this performance using micro-instances then we need approximately 10 small instances (the table shows 10 micro Postgres instances yielding a threshold of value of 1200).

7.3 Performance Evaluation of Cloud Services

What we have seen so far is that the cost per request is more expensive on a larger cloud instance than on a smaller cloud instance. The size here refers to the instance having more CPU's, more memory and more elastic compute units.

So why should we use larger instances when there is a clear cost advantage in the smaller instances? Well we have seen that as the parameter of the Poisson process or as the arrival rate increases the system can no longer handle the traffic of requests and we need to switch to larger instances.

Cloud service providers (Amazon.com Inc.) in our case takes advantage of this and in order for us to be able to handle the increased amount of requests in computing power, be it a database or web server, we are forced to upgrade to a larger instance or face a crippling cloud system that is unresponsive.

7.4 Limitations

There were a few limiting factors to this project.

We were modeling the cloud computing components as an $M/M/1$ queue. This meant that we were constantly trying to fit our service times to an exponential distribution, which may not have been the case. One of the places where I felt that the exponential distribution was not representative of the service time is for the MongoDB writes, referring to figure 5.3 we can see that the data does not quite have the negative-exponential shape.

Chapter 8

Conclusions and Future work

8.1 Conclusion

In this project we successfully managed to:

- Build a tool to receive response times for the database and webserver components of a cloud computing service (in particular MongoDB and Postgres for databases and HTTP webservers).
- Fitted distributions to different response times (using maximum-likelihood) to get the rate parameter.
- Modeled different cloud computing components as $M/M/1$ queues and found threshold values for certain configurations of these queuing networks. We were able to build complex network models that would be hard to build in practice. (e.g. connecting a web server to 20 databases).
- Evaluated the performance of different queuing models and also conducted a cost analysis method for calculating performance vs cost for different cloud computing options (Amazon EC2 instance types).

8.2 Improving the Service Time Evaluation Tool

The service evaluation tool was limited by certain factors. We only conducted an implementation of Postgres, MongoDB and HTTP web server. There are many different components of cloud computing that that would have been interesting to model so that we can generate queuing networks that are much more dynamic and realistic.

For example we could have looked at modeling Content Delivery Networks (CDNs) or Load Balancers, both of which are popular components in the cloud computing space.

Our tool only dealt with read and write operations to databases, we could have gained more insights by looking at other database operations such as update.

8.3 Improving the spectrum of tests conducted

The tests we conducted were only on micro and extra-large EC2 instances. It would have been beneficial to also include small, medium and large instances.

We also only looked at Amazon Web Services as our cloud provider it would also have been interesting to carry out similar analysis on other cloud service providers.

8.4 Future work

This project has three distinct parts:

1. Generation of service times
2. Analysis of service times to fit a distribution
3. Performance and cost analysis of individual components and of the whole system

A future proposal for improving this project would be bringing all the above parts into one simple and easy to use tool.

Other minor future work that can be undertaken is to increase the supported database types. We could also increase the number of functions of the components that we are able to analyse (e.g. adding support for database update query analysis).

Bibliography

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53:50–58, April 2010.

[2] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260, April 1975.

[3] M Bertoli, G Casale, and G Serazzri.

[4] Paul J. Burke. The output of a queuing system. *Operations Research*, 4(6):699–704, 1956.

[5] Panagiotis Christias. Unix man pages: ssh (), 1999.

[6] Thomas Claburn. Amazon ec2 outage hobbles websites, April 2011.

[7] D. Crockford. Json, 2006.

[8] AK Erlang. Nyt tidsskrift for matematik. *Journal of the ACM*, 1909.

[9] The Apache Software Foundation. Apache http server benchmarking tool, 2013.

[10] Pete Harrison. Performance evaluation. 2013.

[11] Pete Harrison. Performance evaluation, 2013.

[12] Inc. Joyent. Node.js, 2013.

[13] D. G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 1953.

[14] Netcraft. February 2009 web server survey, 2009.

[15] Mozilla Developer Network and individual contributors. Creating javascript callbacks in components, March 2011.

[16] Jason Read. What is an ecu? cpu benchmarking in the cloud, May 2010.

[17] K Sigman. A proof of the queueing formula l = λw, 2009.

[18] Twitter. Posting a tweet, 2013.

[19] Twitter. Twitter streaming apis, 2013.

[20] Eric W. Weisstein. Binomial distribution.

[21] Eric W. Weisstein. Exponential distribution.

Appendix A

Amazon Web Services

The **Amazon Web Console** is used for managing Amazon Web Services through a simple web based interface.

Amazon EC2 - Amazon Elastic Compute Cloud, this is a facility to rent out virtual machines. A unit of EC2 is called an instance and before acquiring an instance you can choose different options relating to the instance.

ECU - Elastic Compute Unit is an abstraction of a computing resource. An Elastic Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

Appendix B

Service Time Evaluation Tool usage

The service evaluation tool was tested out in the Mac OS X Mountain Lion and Ubuntu 12.04 operating system distributions. To run the tool you must have the following dependencies installed (most are available by default):

- Python
- Git (for checking out the code)
- PostgreSQL
- MongoDB
- Node.js

For **Python** we need to additionally install the twython, pymongo, psycopg2 libraries. For **Node.js** we just need the express library. Also ensure that you have the libpq-dev and python-dev libraries installed on your machine.

The code can be checked out from `https://github.com/jamalzkhan/PECS.git`, to start using the tool go into the main directory and carry out the following commands:

```
> cd peformance_framework
> mkdir logs
```

You also need to configure the config.conf file with information such as your Twitter API access tokens. An example configuration file has already been created in the code base but requires editing. Please refer to B.1 or B.2 to setup the configuration file for database or HTTP web server evaluations respectively.

B.1 Running database evaluations

Before beginning to use the tool please ensure that the PostgreSQL or MongoDB server is runnning to avoid any exceptions.

If you are using the tool for evaluating performance of PostgreSQL databases then you should create a database and a new user.

```
> sudo -u postgres createuser --superuser $USER
> sudo -u postgres createdb performance
```

The following should be included in your configuration file if you want to conduct a database evaluation.

- Twitter API access tokens
- Postgres username and password
- Database and table name that will be used by the evaluation tool. Please do not use an existing table name if you want to use it for purposes other than this profiling tool, Unless you really know what you are doing!

For PostgreSQL to create a table automatically you can carry out the following command:

```
> Python Performance.py -d postgres -c
```

The following database specific flags are used by the tool:

- -d database_name (Use either postgres or mongo)
- -i inserts (Number of write operations to carry out)
- -s selects (Number of read operations to carry out)
- -c (Automatically create a table, only use with the -d postgres flag)

An example usage of the tool is:

```
> Python Performance.py -d postgres -i 200 -s 300
```

This will insert 200 Tweets into the specified PostgreSQL instance and also carry out 300 read operations on the table where the Tweets are stored.

All the response times for the tool are stored in the ”logs/” folder.

B.2 Running HTTP web server evaluations

Please ensure that a web server is switched on if you want to conduct performance evaluation for a HTTP web server.

The following should be included in your configuration file if you want to conduct a HTTP web server evaluation.

- The URL to which you want to POST data to and also the URL you want to perform GET requests to
- Post data that you would like to send to the web server when conducting POST requests

The following web server specific flags are used by the tool:

- -u url (URL of the web server, e.g. localhost)
- -p port (Port at which the webserver is operating)
- -g gets (Number of GET requests to perform)
- -x posts (Number of POST requests to perform)
- -n threads (Number of concurrent connections to make)

The tool will conduct the post request and get requests multiplied by the concurrent connections. An example usage is:

```
> python Performance.py -u localhost -p 80 -g 3000 -x 2000 -n 30
```

This performs 3000 GET requests and 2000 POST requests multiplied by 30 concurrent threads on the localhost HTTP web server at port 80.

Also note that the Node.js webserver in the code base has a GET and POST method implemented in the "/" and"/posturl" path respectively, so you can use this as a starting point.

Appendix C

Service time evaluation for cloud computing components

C.1 Minimum T values

These values correspond to the T_0 values which are described in 2.17 and are displayed in table C.1.

Component	**Instance type**	**Operations**	T_0
MongoDB	Micro		
		Read	2.1000×10^{-5}
		Write	2.8500×10^{-4}
	Extra-large		
		Read	2.4000×10^{-5}
		Write	2.1700×10^{-4}
Postgres	Micro		
		Read	9.5000×10^{-5}
		Write	1.4×10^{-3}
	Extra-large		
		Read	1.7000×10^{-4}
		Write	1.3×10^{-3}
Node.js web server	Micro		
		GET	7.1300×10^{-4}
		POST	7.9800×10^{-4}
	Extra-large		
		GET	8.9900×10^{-4}
		POST	1.0×10^{-3}

Table C.1: T_0 calculated values

C.2 Calculated service time parameters

Component	Instance type	Operations	μ_{MLE}
MongoDB	Micro		
		Read	43.300
		Write	0.745
	Extra-large		
		Read	67.902
		Write	6.4354
Postgres	Micro		
		Read	0.9091
		Write	0.1266
	Extra-large		
		Read	1.5684
		Write	1.2624
Node.js web server	Micro		
		GET	4.3693
		POST	1.9773
	Extra-large		
		GET	21.6828
		POST	11.2618

Table C.2: μ_{MLE} calculated values

Appendix D

JMT Modeling Results

Our results for different configurations of queuing networks are below. We were using the parameters (see C.2) for a Node.js web server POST requests and for the PostgreSQL write operations for the database.

The graphs show the system response time and there is a clear mark for each of them where the service time increases exponentially. This is what is referred to as the threshold.

All the values listed below are not to scale, this is as the input parameters to the tool were scaled as well. The values calculated in table 6.1 have been scaled back appropriately.

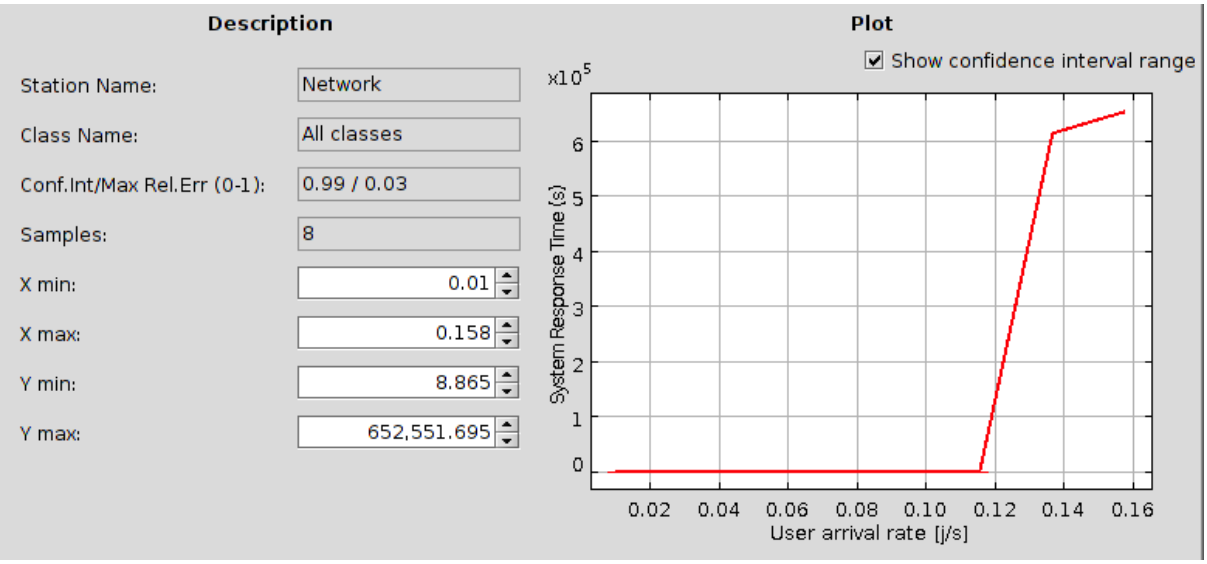


Figure D.1: 1 micro instance web server and 1 micro instance database

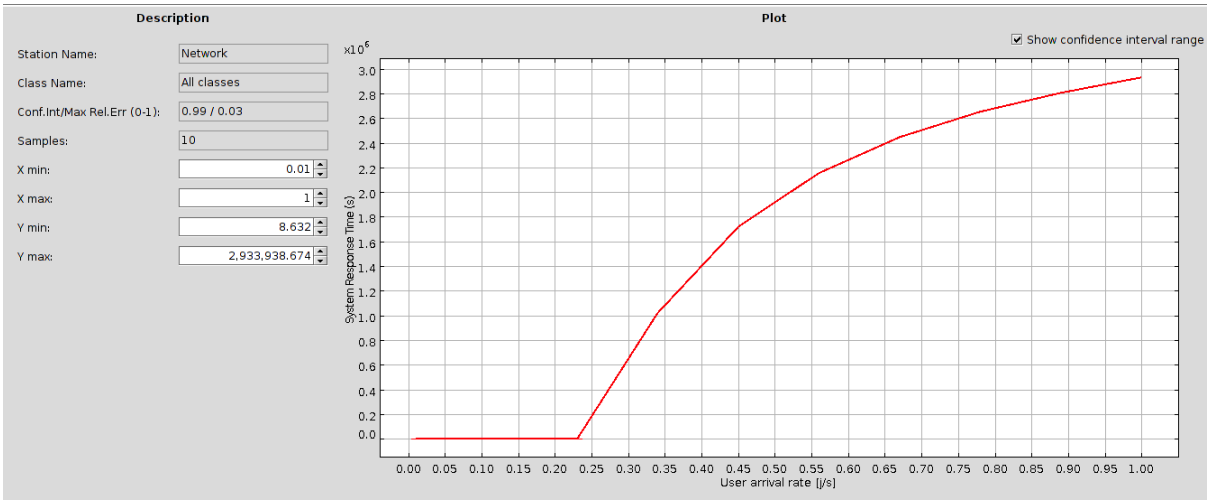


Figure D.2: 1 micro instance web server and 2 micro instance databases

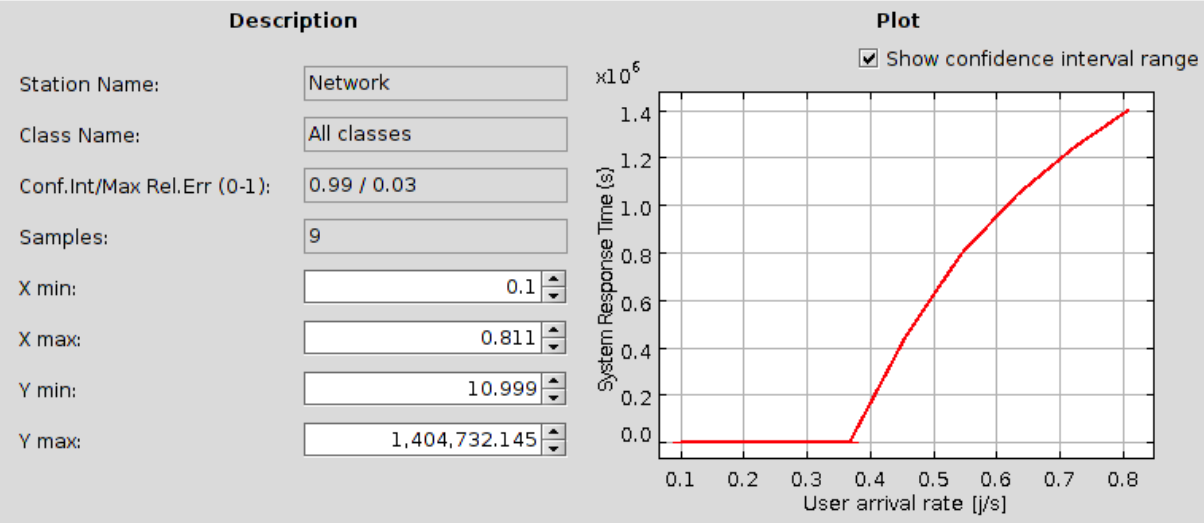


Figure D.3: 1 micro instance web server and 4 micro instance databases

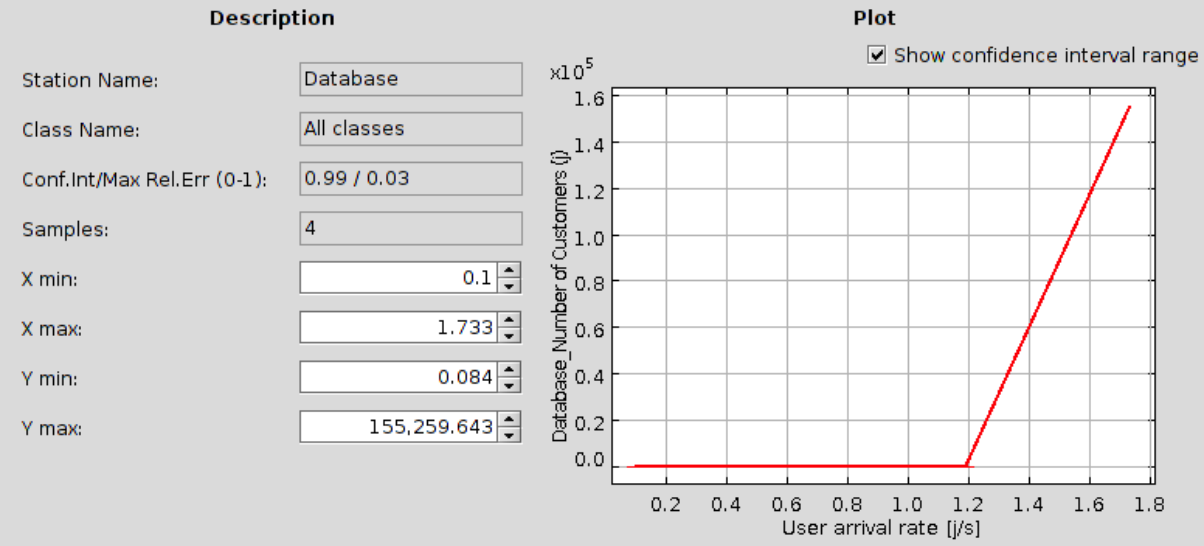


Figure D.4: 1 micro instance web server and 10 micro instance databases

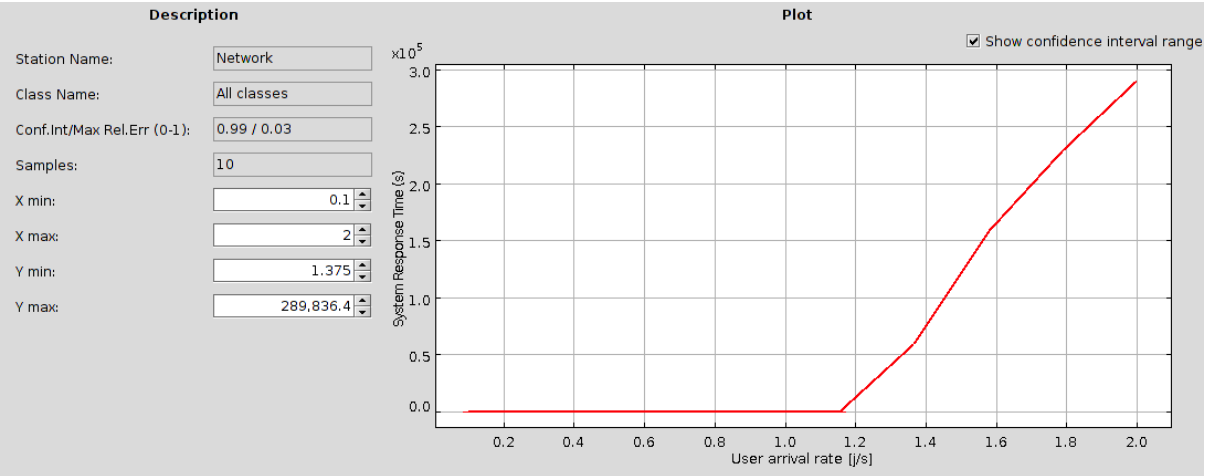


Figure D.5: 1 micro instance web server and 1 extra-large instance database