A Case Study in Web Application Performance Measurement

by

Nitin Goyal
B.Tech., Baldev Ram Mirdha Institute of Technology, 2011

A Master Project Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

A Case Study In Web Application Performance Measurement

by

Nitin Goyal
B.Tech., Baldev Ram Mirdha Institute of Technology, 2011

Supervisory Committee

Dr. Daniel Hoffman, Supervisor

(Department of Computer Science)

Dr. Sudhakar Ganti, Departmental Member

(Department of Computer Science)

Supervisory Committee

Dr. Daniel Hoffman, Supervisor

(Department of Computer Science)

Dr. Sudhakar Ganti, Departmental Member

(Department of Computer Science)

Abstract

The Computational Quiz Generation (CQG) system is a web application that provides online programming quizzes. CQG has been used in CSC 111, CSC 116, CSC 361, SEng 265 and SEng360. In the future we want to use CGQ in larger sections but due to the unavailability of performance metrics on CQG, it would be risky. We want to get quantitative performance data. We are interested in identifying maximum number of users supported stably by CQG, quiz start up time and if Java questions are expensive. Hence performance testing was conducted on CQG using Apache JMeter. Several tests were conducted to collect quantitative performance data relating to speed, stability and scalability. This project is a deployment of the test infrastructure on CGQ that would benefit the stakeholders in CQG to better determine and understand problems related to the maximum number of supported users, start up delays, expensive questions, etc. Experimental results have shown that the quiz start up time is high and depends on the size of the question library. It was also found that Java questions are much more expensive to use than C, C++ and Python. Performance testing has also uncovered the modules in CGQ that requiring optimization.

Table of Contents

Table of Figures

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Daniel Hoffman for the useful comments, remarks and engagement through the learning process of this master's report. I would like to thank him for suggesting the project topic and for the support on the way.

Furthermore, I would like to thank my fiancée and parents who have supported me throughout the entire process, both by keeping me harmonious and helping me putting pieces together. I will be grateful forever for your love.

1.0 Introduction

1.1 CQG system

Computational Quiz Generation (*CQG*) is an online quiz generation framework focussing on code reading. Figure 1 shows a Python quiz where the student entered the standard output. The student then clicks the *Check answer* button and the quiz returns a *Correct* message as the entered value is correct.

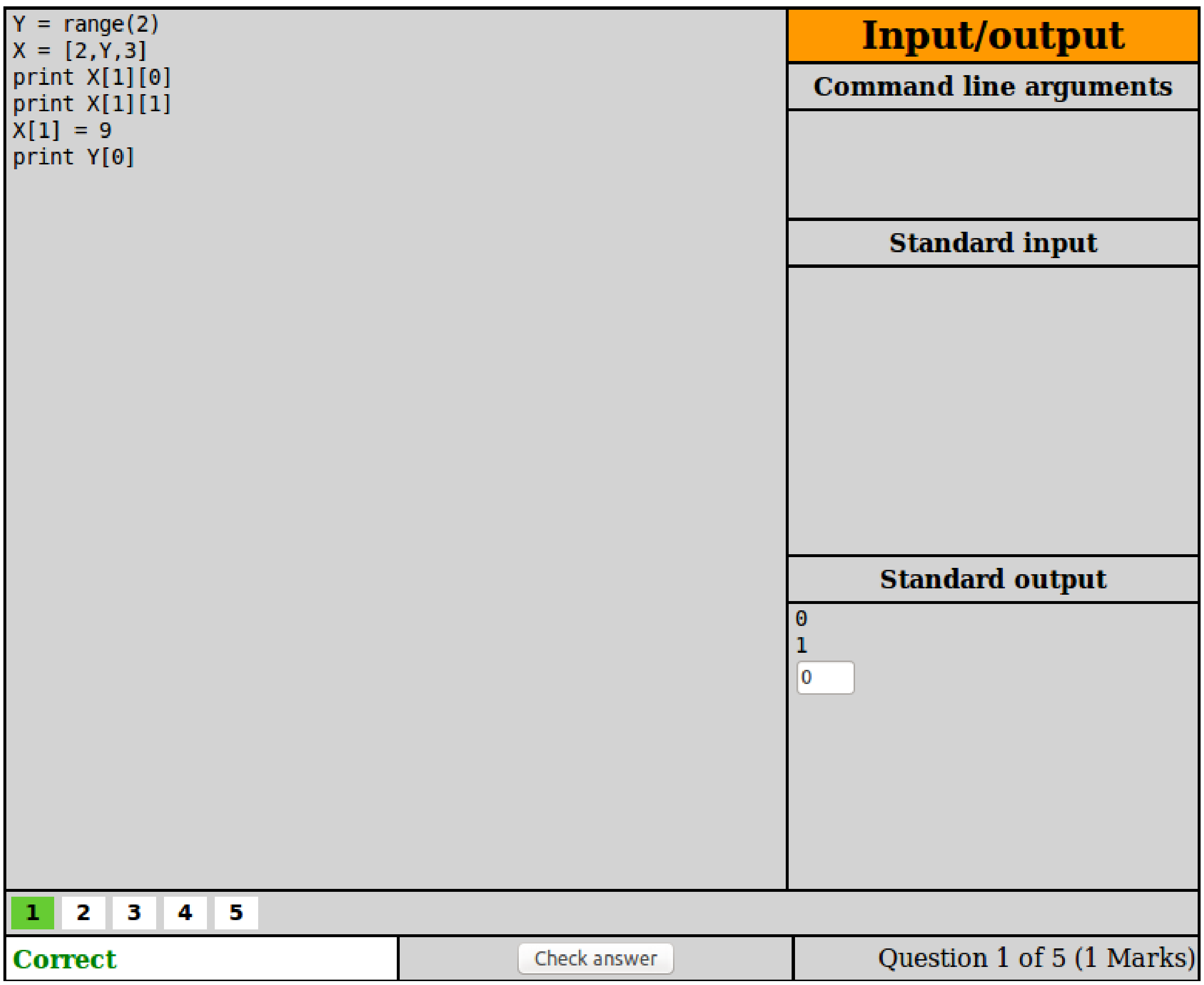


Figure 1 : CGQ Quiz in Python

CQG is implemented with HTML forms and Python. CQG has been used in Computer Science (CSC 111, CSC116 and CSC 361) and Software Engineering (SEng 265, SEng 360) course offerings. CGQ quizzes have been developed for C, C++, Java and Python languages.

1.2 Problem statement

While CQG has been used in many CSC and SEng courses, we have no quantitative performance data on CQG. In particular we are interested in finding out the performance metrics for the following questions:

1.2.1 Number of users supported (Scalability and Stability)

We want to identify the number of users CQG can support stably. A Performance metric for the number of users supported and the corresponding delays and CPU utilizations will help us to better configure the hosting servers.

1.2.2 Expected Start up delays (Speed)

CQG quizzes experience delays while starting up for the first time. We are interested in identifying and quantifying these delays. This is important to know as we want to minimize the start-up delays to make CQG quizzes load faster. We are also interested in identifying the actions that causes these delays in CQG.

1.2.3 Question cost by language

CQG supports programming quizzes for C, C++, Java and Python. Answer checking in CQG is usually very fast as the code is precompiled. However we are interested in testing whether the performance metrics are different for different languages. We expect that Java questions are expensive due to the startup time and high memory usage of JVM as compared to C, C++ or Python.

1.2.4 Cost of quiz logging

For each user in CQG an XML log file is generated. These log files captures users actions for answer submission or moving to a next question. For each action performed by the user a write operation is triggered on the XML file. We are interested in identifying if this approach of logging actions is expensive or not.

1.3 Use in large sections

The above mentioned problems are important as we want to use CGQ in larger sections. To do so we need more information about CQG behaviour under load.

With the knowledge of these performance metrics it would be helpful to better estimate the load on the server. This information can be advantageous to make better decisions about server configuration. Performance testing will also identify the places of improvement in CGQ so that it can be optimized in the future.

1.4 Solution approach

To better understand and measure CQG performance we tested it using *Apache JMeter*.

The types of performance testing we conducted are:

1. *Load testing:* Checks the application's ability to perform under anticipated user loads.

2. *Stress testing*: Involves testing the application under extreme workloads to identify the breaking point of the application.

1.5 Experimental results

The results from the performance tests have provided us quantitative data relating to *number of users*, *minimum/maximum Response time*, size of *HTTP GET/Reply* and *CPU utilization.*

1.6 My Contribution

I conducted Performance testing of CQG using Apache JMeter to analyse its scalability and load endurance capacity. The contributions are:

- Evaluation of CGQ start-up lag time
- Determination of the maximum number of users CQG could support stably
- Identification of which questions are expensive
- Measurement of quiz logging cost
- Determination of the effect of question library size on quiz start up time:

JMeter components are applied and understood by doing a few initial experiments and then later mapped to CQG accordingly. The primary focus while creating the test runs was to identify the answers for the problems discussed in the previous section. After conducting the measurements and analysis we were able to provide quantitative data on the performance characteristics of CQG.

1.7 Organisation of the report

In Chapter 2, *Apache JMeter* and *CQG* are introduced with the definition of components and concepts used in the experiments. Information for *HTTP*, *TCP* and *HTML* is also presented to form the background for networking concepts. Chapter 3 describes the experimental design used in the performance measurement and presents the initial experiments conducted to understand JMeter features. Chapter 4 presents the experiment results, Chapter 5 provides the conclusion from the analysis of the experiment results and Chapter 6 presents the Future work.

2.0 Background

2.1 JMeter

2.1.1 Introduction

Apache *JMeter* is an *Apache project* [1] that provides a load testing tool for analyzing and measuring the performance of a variety of services, with a focus on web applications. JMeter can be used to generate a variety of loads on a server by generating HTTP requests that hit the specified server. JMeter supports variable *parameterization*, *assertions* (response validation), *per Thread cookies*, configuration variables and a variety of report generation features.

A test plan can describes the steps that JMeter will execute when run. A complete test plan can have one or more thread groups, logic controllers, listeners and timers.

2.1.2 Thread group

A *thread group* consists of controllers and samplers under it. There are certain controls defined in a thread group are:

- *Number of threads*: It can be considered as the number of users.
- *Ramp-up period*: The time taken by the JMeter to start the total number of threads. For example, if there are 10 threads, and the ramp up period is 50 seconds, then each thread will start 5=50/10 seconds after the previous thread has begun.

2.1.3 Sampler

The *Sampler* tells the JMeter to send the request to a specified server and wait for a response. We are using HTTP Request sampler, which allows JMeter to send an HTTP/HTTPS request.

2.1.4 Timer

Timer are used to introduce delay before each sampler. Without a timer, JMeter might overwhelm the server by making too many requests in a very short amount of time. We are using *Constant Timer* and *Uniform Random Timer* for our experiments.

2.1.5 Listener

The *Listener* provides access to the information that JMeter has collected about the test case while JMeter runs. We are using *View Results in Table* and *Summary Report* listener for our experiments.

2.1.5.1 View Result in Table

The concept of latency and sample time can be illustrated by a timing diagram as shown in Figure 1.

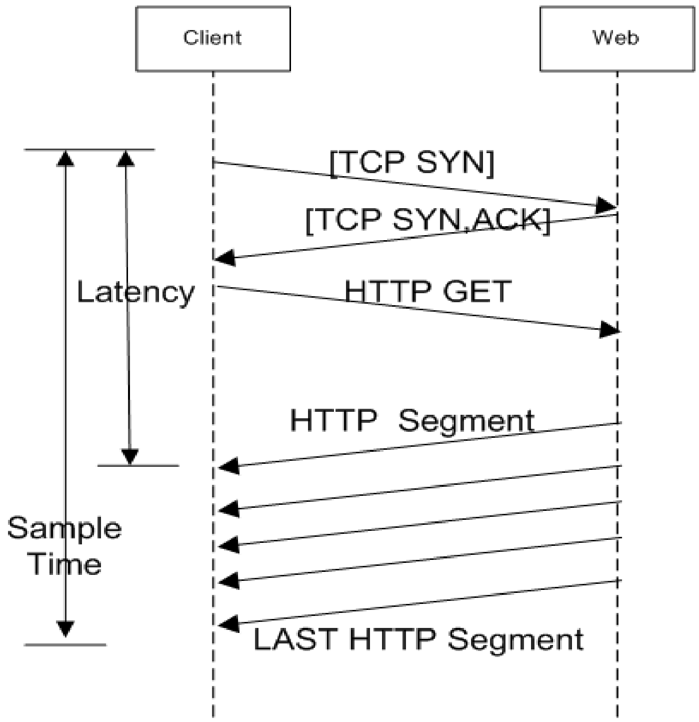
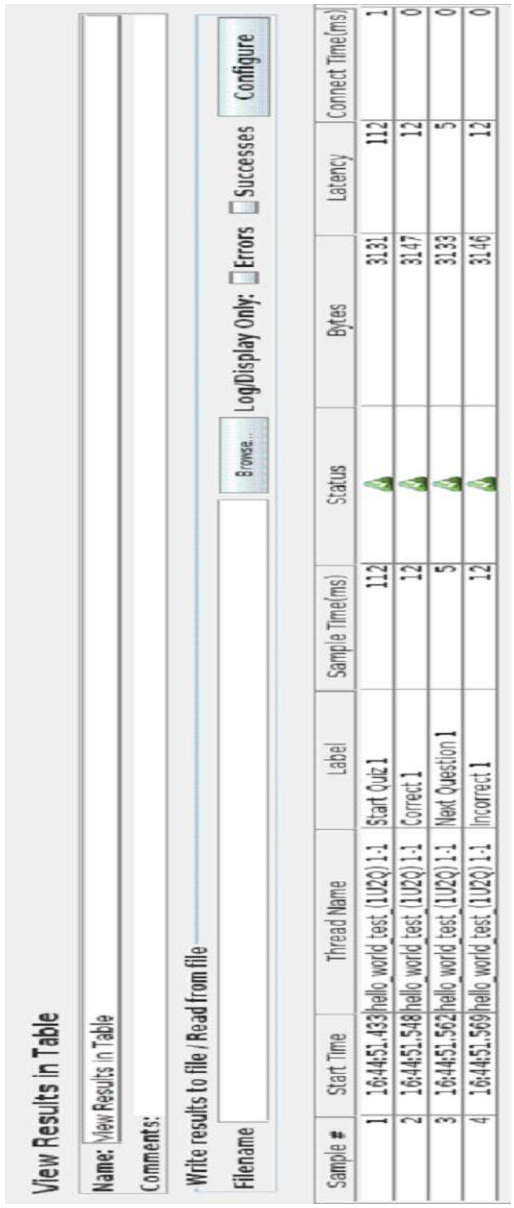


Figure 2 : Latency-Sample time Diagram

The columns contained in Result table as shown in Figure 2 can be defined as:

- *Sample time*: The time from invoking the request to the last byte of the response coming back.
- *Bytes*: The size of the data in the sample response returned from the server.
- *Latency Time*: The time from invoking the request to the first packet of the response coming back.
- *Connection Time*: The time taken to establish the connection with including SSL handshake.

The columns Sample #, Start Time, Thread Name, Label and Status are not used in this report.



Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Latency	Connect Time(ms)
1	16:44:51.433	hello world test (1U2Q) 1-1	Start Quiz 1	112		3131	112	1
2	16:44:51.548	hello world test (1U2Q) 1-1	Correct 1	12		3147	12	0
3	16:44:51.562	hello world test (1U2Q) 1-1	Next Question 1	5		3133	5	0
4	16:44:51.569	hello world test (1U2Q) 1-1	Incorrect 1	12		3146	12	0

Figure 3 : View Results in Table

To confirm the correctness of results as shown in Figure 2. We created a simple CQG quiz containing two questions. The Submit and next question are emulated using the HTTP request sampler. Verification of the columns of View Result Table related to latency and sample time is done with the help of a packet sniffing tool called Wireshark as shown in Figure 3.

As seen in Figure 2, Sample 1 indicates the Sample time, Latency as 112 ms which can be confirmed from the Wireshark experiment as shown in Figure 3 with packet number 8.

It indicates the time as approximately 111 ms excluding the 1 ms connection time that can add up to 112 ms, in our case Latency and Sample time is the same as only one HTTP segment gets returns from the server. Bytes represents the size of the response for the request from the server. Connection time is 1 ms as the connection was very fast.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.2	TCP	74	54762 > http-alt [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=435267787 TSecr=
2	0.000032000	10.0.0.2	10.0.0.1	TCP	74	http-alt > 54762 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=50344
3	0.000154000	10.0.0.1	10.0.0.2	TCP	66	54762 > http-alt [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=435267787 TSecr=503446
4	0.000650000	10.0.0.1	10.0.0.2	HTTP	208	GET /cqg/quiz?spec=111/hello_world_1 HTTP/1.1
5	0.000663000	10.0.0.2	10.0.0.1	TCP	66	http-alt > 54762 [ACK] Seq=1 Ack=143 Win=30080 Len=0 TSval=503446 TSecr=435267788
6	0.111440000	10.0.0.2	10.0.0.1	TCP	463	[TCP segment of a reassembled PDU]
7	0.111464000	10.0.0.2	10.0.0.1	TCP	1514	[TCP segment of a reassembled PDU]
8	0.111523000	10.0.0.2	10.0.0.1	HTTP	1352	HTTP/1.1 200 OK (text/html)
9	0.111604000	10.0.0.1	10.0.0.2	TCP	66	54762 > http-alt [ACK] Seq=143 Ack=398 Win=30336 Len=0 TSval=435267815 TSecr=503473
10	0.111664000	10.0.0.1	10.0.0.2	TCP	66	54762 > http-alt [ACK] Seq=143 Ack=1846 Win=33280 Len=0 TSval=435267815 TSecr=503473
11	0.111786000	10.0.0.1	10.0.0.2	TCP	66	54762 > http-alt [ACK] Seq=143 Ack=3132 Win=36096 Len=0 TSval=435267815 TSecr=503473
12	0.116052000	10.0.0.1	10.0.0.2	HTTP	300	GET /cqg/quiz?%24out=hello+world&but=Check+answer&currqn=0 HTTP/1.1
13	0.116061000	10.0.0.2	10.0.0.1	TCP	66	http-alt > 54762 [ACK] Seq=3132 Ack=377 Win=31104 Len=0 TSval=503475 TSecr=435267816
14	0.126170000	10.0.0.2	10.0.0.1	TCP	463	[TCP segment of a reassembled PDU]
15	0.126197000	10.0.0.2	10.0.0.1	TCP	1514	[TCP segment of a reassembled PDU]
16	0.126257000	10.0.0.2	10.0.0.1	HTTP	1368	HTTP/1.1 200 OK (text/html)

Figure 4: Wireshark Experiment to confirm JMeter latency & Sample time

2.1.5.2 Summary Report

The *Summary report* contains a row for each differently named request in the test. The Summary report provides information about the minimum/maximum response time and throughput.

- *Average*: The mean response time in milliseconds for a particular HTTP request.
- *Min*: Minimum response time taken by the request.
- *Max*: Maximum response time taken by the request.
- *Throughput*: The number of requests per unit of time that are sent to the server under test.

We are particularly focused on identifying the HTTP requests for which the Maximum response time is greater than 5 seconds. These requests will provide information about the CQG load time.

2.2 CQG

CQG offers quizzes in practice and marked mode. In Practice mode, there is no User authentication and quiz logging. Quizzes under marked mode are authenticated by the login credentials provided to the students at the beginning of the term. Marked quizzes are logged on the server for each action performed by the students on the client. Quiz logs are then used for marks calculation.

In terms of CQG, we define the JMeter variables which are used to perform different experiments on the server with varying load and number of users.

- *Number of threads*: Number of students/users attempting the quiz
- *Ramp-up Period*: Time taken from quiz start to see the first question.
- *Timer*: Delay between each pair of submit actions.

2.3 HTTP

Hypertext Transfer Protocol is an underlying protocol used by World Wide Web. It defines how messages are formatted and transmitted over the internet. It also formulates the specification of the actions that web servers and browsers should take in response to various commands.

2.3.1 GET

Prominent request methods in HTTP are *GET*, *POST* and *PUT*. CQG uses only GET method to interact with the server. This can be confirmed by packet no. 4 of Figure 4. A request containing the GET method has name/value pairs in the URL which requests data from a specified resource.

2.3.2 HTTP Persistent connection

Persistent connection or *HTTP Keep-alive* is the idea of using a single *TCP* connection to send and receive multiple HTTP request/response. CQG uses HTTP 1.1 under which all the connections are considered persistent unless declared otherwise. JMeter has the functionality to define HTTP requests with the Keep-alive tag that is responsible for persistent connections. HTTP 1.1 behaviour can be confirmed from packet 4 of Figure 4. A TCP connection is established only once in the beginning of the quiz as shown in packet 1 and 2. This connection is then used by the subsequent HTTP requests in packet 12 and 16.

No.	Time	Source	Destination	Protocol	Lengtł	Info
1	0.000000000	10.0.0.1	10.0.0.2	TCP	74	54762 > http-alt [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=435267787 TSecr=0 WS=128
2	0.000032000	10.0.0.2	10.0.0.1	TCP	74	http-alt > 54762 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=503446 TSecr=435267787 WS=128
3	0.000154000	10.0.0.1	10.0.0.2	TCP	66	54762 > http-alt [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=435267787 TSecr=503446
4	0.000650000	10.0.0.1	10.0.0.2	HTTP	208	GET /cqg/quiz?spec=111/hello_world_1 HTTP/1.1
5	0.000663000	10.0.0.2	10.0.0.1	TCP	66	http-alt > 54762 [ACK] Seq=1 Ack=143 Win=30080 Len=0 TSval=503446 TSecr=435267788
6	0.111440000	10.0.0.2	10.0.0.1	TCP	463	[TCP segment of a reassembled PDU]
7	0.111464000	10.0.0.2	10.0.0.1	TCP	1514	[TCP segment of a reassembled PDU]
8	0.111523000	10.0.0.2	10.0.0.1	HTTP	1352	HTTP/1.1 200 OK (text/html)
9	0.111604000	10.0.0.1	10.0.0.2	TCP	66	54762 > http-alt [ACK] Seq=143 Ack=398 Win=30336 Len=0 TSval=435267815 TSecr=503473
10	0.111664000	10.0.0.1	10.0.0.2	TCP	66	54762 > http-alt [ACK] Seq=143 Ack=1846 Win=33280 Len=0 TSval=435267815 TSecr=503473
11	0.111786000	10.0.0.1	10.0.0.2	TCP	66	54762 > http-alt [ACK] Seq=143 Ack=3132 Win=36096 Len=0 TSval=435267815 TSecr=503473
12	0.116052000	10.0.0.1	10.0.0.2	HTTP	300	GET /cqg/quiz?%24out=hello+world&but=Check+answer&currqn=0 HTTP/1.1
13	0.116061000	10.0.0.2	10.0.0.1	TCP	66	http-alt > 54762 [ACK] Seq=3132 Ack=377 Win=31104 Len=0 TSval=503475 TSecr=435267816
14	0.126170000	10.0.0.2	10.0.0.1	TCP	463	[TCP segment of a reassembled PDU]
15	0.126197000	10.0.0.2	10.0.0.1	TCP	1514	[TCP segment of a reassembled PDU]
16	0.126257000	10.0.0.2	10.0.0.1	HTTP	1368	HTTP/1.1 200 OK (text/html)
17	0.126399000	10.0.0.1	10.0.0.2	TCP	66	54762 > http-alt [ACK] Seq=377 Ack=4977 Win=41856 Len=0 TSval=435267819 TSecr=503477
18	0.130293000	10.0.0.1	10.0.0.2	HTTP	264	GET /cqg/quiz?currqn=1 HTTP/1.1

Figure 5: Wireshark Experiment to confirm HTTP and TCP (3-way handshake)

2.4 TCP

TCP enables two hosts to establish a connection and exchange streams of data. Using Wireshark we confirmed that a single TCP connection is used to handle multiple quiz submit presses as shown in Figure 4.

2.4.1 Three-Way Handshake

A three-step method is used in a TCP/IP network to create a connection. This connection requires both client and server to exchange *SYN* and *ACK* packets before actual data communication begins. The Three-way handshake is shown in Figure 5 and confirmed from packets 1, 2 and 3 of Figure 4.

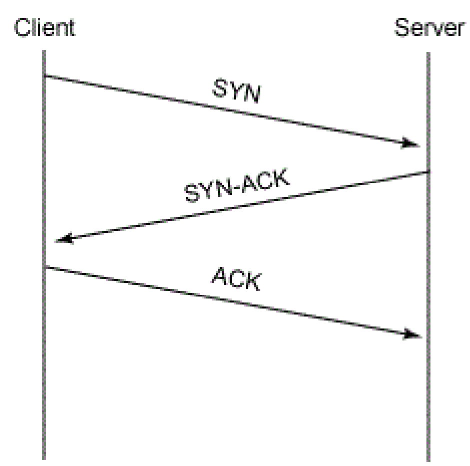


Figure 6: TCP three way handshake

2.4.2 PDU (Protocol Data Unit)

PDU is the information delivered as a unit among peer entities of network and that may contain control information, such as address information or user data [4].

2.5 HTML

HTML or Hyper Text Mark-up language in CQG is very light-weight and does not contain any images or JavaScript. HTML forms are used with no client side embedded code and are generated on the server side using web2py .Each HTML page in CQG contains textboxes for entering the expected input or output and buttons for submitting/checking the answers and switching between questions.

3.0 Experimental Design

3.1 Dominant control variables

The control variables that are used in the experiments to vary the load are:

- Ramp-up
- Timer
- Number of threads
- Quiz Content: Quizzes containing C, Python and Java questions.

3.2 Test Setup

Performance tests are run with two different setups:

3.2.1 Local server setup

In this setup, one machine (*Asia*) is the test server on which CQG is running over port 8081. Asia is connected by a cross-over cable to another machine (*India*) on which JMeter is running to generate traffic. Figure 6 shows the Local server setup.

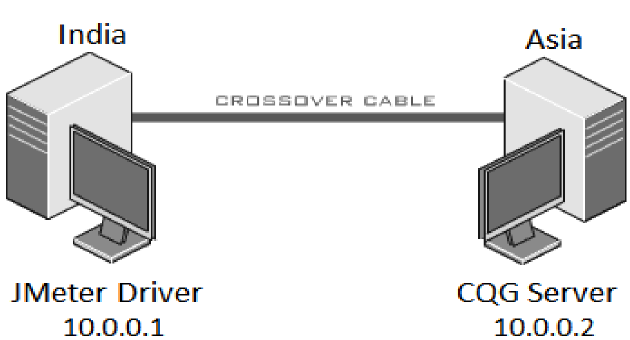


Figure 7 : Local server setup using crossover cable

Configuration of machines in local setup is shown in Table 1.

Machine name	India	Asia
Processor	Intel i7-2600 @3.40 GHz	Intel Core 2 Duo @2.33 GHz
RAM	4 GB	2 GB
Operating System	Ubuntu 14.04	Ubuntu 14.04
CPU Core	8	2

Table 1: Local Server Setup

3.2.2 SEng server setup

In this setup, a virtual server (*cqg.seng.uvic.ca*) is deployed using *Proxmox*. *Proxmox* is a server virtualization management solution. This server is publically accessible and runs CQG on port 8081. Machine *India* is running JMeter which targets the virtual server. Figure 7 shows the SEng server setup.

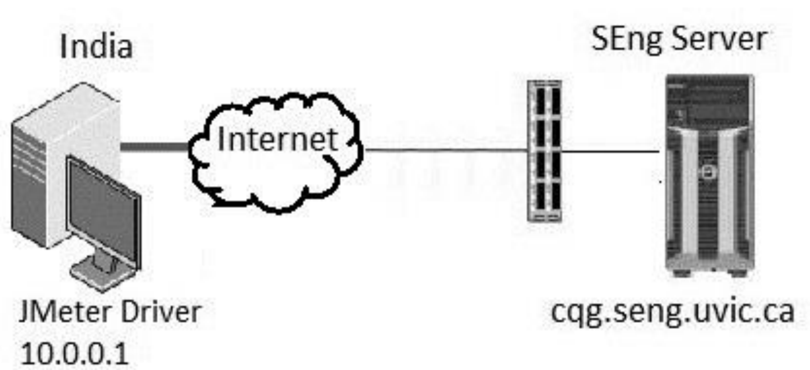


Figure 8: SEng Server Setup

Configuration of the server is shown in Table 2.

Server Name	cqg.seng.uvic.ca
Processor	KVM 64 bit
RAM	Variable (512 MB – 1 GB)
Operating System	Scientific Linux 6.7
CPU Core	1 core

Table 2: SEng Server Configuration

3.3 JMeter Delay Experiment

To measure the JMeter delay accuracy we conducted several experiments. We tested *Constant Timer* and *Uniform Random Timer* using Wireshark.

3.3.1 Constant Timer

Constant timer introduces a fixed delay between consecutive requests of the same thread. This is useful when we want to have each thread pause for the same amount of time. The configuration used for this experiment is:

- *HTTP Sampler* : 3 identical requests to the CQG static page
- *Number of Threads* : 1
- *Thread Delay (ms)* : 1, 10, 100, 1000

We ran several tests using different Thread delays and measured the delay accuracy using Wireshark. We found the Constant Timer accurate. A Wireshark dump for a thread delay of 100 ms in Constant timer is shown in Figure 8. GET requests from Packet no. 4, 10, 15 and 20 shows the delay of 100 ms.

0

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.2	TCP	74	40696 > http-alt [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=195809499 TSecr=0 WS=128
2	0.000031000	10.0.0.2	10.0.0.1	TCP	74	http-alt > 40696 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=109011759 TSecr=195809499 WS=128
3	0.000131000	10.0.0.1	10.0.0.2	TCP	66	40696 > http-alt [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=195809499 TSecr=109011759
4	0.000374000	10.0.0.1	10.0.0.2	HTTP	194	GET /cqg/default/index HTTP/1.1
5	0.000386000	10.0.0.2	10.0.0.1	TCP	66	http-alt > 40696 [ACK] Seq=1 Ack=129 Win=30080 Len=0 TSval=109011759 TSecr=195809499
6	0.002092000	10.0.0.2	10.0.0.1	TCP	462	[TCP segment of a reassembled PDU]
7	0.002156000	10.0.0.2	10.0.0.1	HTTP	424	HTTP/1.1 200 OK (text/html)
8	0.002229000	10.0.0.1	10.0.0.2	TCP	66	40696 > http-alt [ACK] Seq=129 Ack=397 Win=30336 Len=0 TSval=195809500 TSecr=109011759
9	0.002238000	10.0.0.1	10.0.0.2	TCP	66	40696 > http-alt [ACK] Seq=129 Ack=755 Win=31360 Len=0 TSval=195809500 TSecr=109011759
10	0.105709000	10.0.0.1	10.0.0.2	HTTP	194	GET /cqg/default/index HTTP/1.1
11	0.107097000	10.0.0.2	10.0.0.1	TCP	462	[TCP segment of a reassembled PDU]
12	0.107158000	10.0.0.2	10.0.0.1	HTTP	424	HTTP/1.1 200 OK (text/html)
13	0.107241000	10.0.0.1	10.0.0.2	TCP	66	40696 > http-alt [ACK] Seq=257 Ack=1151 Win=32512 Len=0 TSval=195809526 TSecr=109011785
14	0.107247000	10.0.0.1	10.0.0.2	TCP	66	40696 > http-alt [ACK] Seq=257 Ack=1509 Win=33536 Len=0 TSval=195809526 TSecr=109011785
15	0.210703000	10.0.0.1	10.0.0.2	HTTP	194	GET /cqg/default/index HTTP/1.1
16	0.212081000	10.0.0.2	10.0.0.1	TCP	462	[TCP segment of a reassembled PDU]
17	0.212141000	10.0.0.2	10.0.0.1	HTTP	424	HTTP/1.1 200 OK (text/html)
18	0.212251000	10.0.0.1	10.0.0.2	TCP	66	40696 > http-alt [ACK] Seq=385 Ack=1905 Win=34560 Len=0 TSval=195809552 TSecr=109011812
19	0.212258000	10.0.0.1	10.0.0.2	TCP	66	40696 > http-alt [ACK] Seq=385 Ack=2263 Win=35712 Len=0 TSval=195809552 TSecr=109011812
20	0.317080000	10.0.0.1	10.0.0.2	HTTP	194	GET /cqg/default/index HTTP/1.1
21	0.318404000	10.0.0.2	10.0.0.1	TCP	462	[TCP segment of a reassembled PDU]
22	0.318463000	10.0.0.2	10.0.0.1	HTTP	424	HTTP/1.1 200 OK (text/html)

Figure 9 : Constant Timer - Wireshark Dump

3.3.2 Uniform Random Timer

This timer pauses each thread request for a random amount of time. It will delay consecutive requests of the same thread by a random interval within lower and upper bounds. Uniform Random Timer consists of two components:

- *Random Delay maximum (ms)*: Maximum random number of milliseconds to pause.
- *Constant Delay Offset (ms)*: Number of milliseconds to pause in addition to the random delay.

Total delay is the sum of the Random value and constant offset value.

Example: If *Constant delay offset* is 1000 ms and *Random Delay maximum* is 200 ms than all threads will be delayed between 1000 ms and 1200 ms.

The configuration used for this experiment is:

- *HTTP Sampler* : 3 identical requests to the CQG static Page
- *Number of Threads* : 1
- *Constant Delay Offset/Random Delay Maximum (ms)* : 1000/1, 1000/10, 1000/100

We ran several experiments using the defined configuration and found Uniform Random Timer to be very accurate and random. A Wireshark dump for the experiment with *Constant Delay Offset/Random Delay Maximum* of 1000/100 ms is shown in Figure 9. GET requests from Packet no. 4, 10, 15 and 20 shows random delay of 1059, 1029 and 1022 ms.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.2	TCP	74	40711 > http-alt [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=195928501 TSecr=0 WS=128
2	0.000033000	10.0.0.2	10.0.0.1	TCP	74	http-alt > 40711 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=109130757 TSecr=195928501 WS=128
3	0.000154000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=195928501 TSecr=109130757
4	0.000583000	10.0.0.1	10.0.0.2	HTTP	194	GET /cqg/default/index HTTP/1.1
5	0.000595000	10.0.0.2	10.0.0.1	TCP	66	http-alt > 40711 [ACK] Seq=1 Ack=129 Win=30080 Len=0 TSval=109130757 TSecr=195928501
6	0.002327000	10.0.0.2	10.0.0.1	TCP	462	[TCP segment of a reassembled PDU]
7	0.002391000	10.0.0.2	10.0.0.1	HTTP	424	HTTP/1.1 200 OK (text/html)
8	0.002489000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=129 Ack=397 Win=30336 Len=0 TSval=195928501 TSecr=109130758
9	0.002499000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=129 Ack=755 Win=31360 Len=0 TSval=195928501 TSecr=109130758
10	1.059846000	10.0.0.1	10.0.0.2	HTTP	194	GET /cqg/default/index HTTP/1.1
11	1.062375000	10.0.0.2	10.0.0.1	TCP	462	[TCP segment of a reassembled PDU]
12	1.062443000	10.0.0.2	10.0.0.1	HTTP	424	HTTP/1.1 200 OK (text/html)
13	1.062553000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=257 Ack=1151 Win=32512 Len=0 TSval=195928766 TSecr=109131023
14	1.062565000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=257 Ack=1509 Win=33536 Len=0 TSval=195928766 TSecr=109131023
15	2.089764000	10.0.0.1	10.0.0.2	HTTP	194	GET /cqg/default/index HTTP/1.1
16	2.091615000	10.0.0.2	10.0.0.1	TCP	462	[TCP segment of a reassembled PDU]
17	2.091682000	10.0.0.2	10.0.0.1	HTTP	424	HTTP/1.1 200 OK (text/html)
18	2.091789000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=385 Ack=1905 Win=34560 Len=0 TSval=195929024 TSecr=109131280
19	2.091801000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=385 Ack=2263 Win=35712 Len=0 TSval=195929024 TSecr=109131280
20	3.111970000	10.0.0.1	10.0.0.2	HTTP	194	GET /cqg/default/index HTTP/1.1
21	3.113794000	10.0.0.2	10.0.0.1	TCP	462	[TCP segment of a reassembled PDU]
22	3.113861000	10.0.0.2	10.0.0.1	HTTP	424	HTTP/1.1 200 OK (text/html)
23	3.113963000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=513 Ack=2659 Win=36736 Len=0 TSval=195929279 TSecr=109131535
24	3.113974000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=513 Ack=3017 Win=37888 Len=0 TSval=195929279 TSecr=109131535
25	4.210221000	10.0.0.1	10.0.0.2	HTTP	194	GET /cqg/default/index HTTP/1.1
26	4.212031000	10.0.0.2	10.0.0.1	TCP	462	[TCP segment of a reassembled PDU]
27	4.212099000	10.0.0.2	10.0.0.1	HTTP	424	HTTP/1.1 200 OK (text/html)
28	4.212206000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=641 Ack=3413 Win=38912 Len=0 TSval=195929554 TSecr=109131810
29	4.212217000	10.0.0.1	10.0.0.2	TCP	66	40711 > http-alt [ACK] Seq=641 Ack=3771 Win=39936 Len=0 TSval=195929554 TSecr=109131810

Figure 10 : Uniform Random Timer: Wireshark Dump

3.4 Test Run's design

Each test run contains one thread group. Each thread group will contain one thread per student. We divide the quiz into N chunks, where one chunk represents an HTTP request that hits the specified server. We are using same questions, answers, and order and inter-submit delay in each thread group.

We conducted several experiments on the local and SEng server. Experiments on the local and SEng server are identical in the configuration. In each experiment we changed either the number of threads or the quiz language (C, Python and Java). The constant and varying parameters in the tests are:

1. Constant parameters

   Ramp-up time: 30 seconds

   Constant Timer: 100 ms

   Number of questions in the quiz: 2

2. Varying parameters

   Number of Threads: 30, 100-190 (difference of 10), 200-1000 (difference of 100)

   Language: C, Python and Java

   Quiz Authentication: True or False

   Quiz Logging: True or False

4.0 Experimental Results

In this section, we present the results that will enable us to answer the questions that are introduced in the Introduction.

Table 3 shows the results analysed from running experiments on local server for C language CQG quiz.

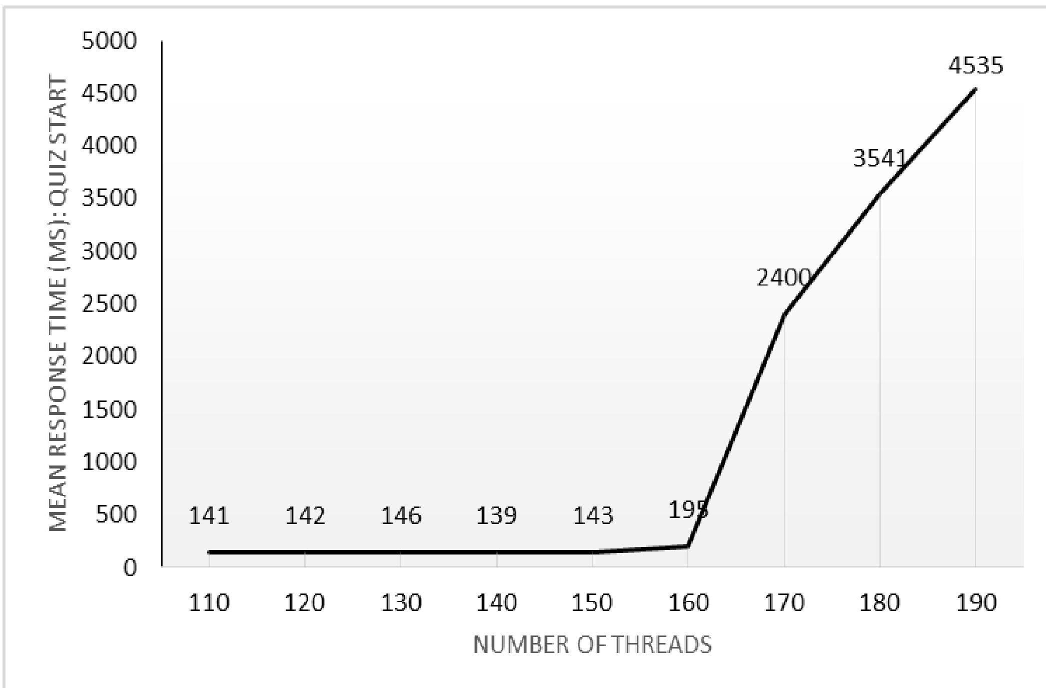
Mean Response time(ms)											
No. of threads:	30	100	200	300	400	500	600	700	800	900	1000
StartQuiz	121	129	5826	15807	25911	35985	46078	56173	56450	76810	86902
Authentication	10	9	28	28	30	29	29	29	28	29	28
AnswerCheck-Q1	12	18	31	34	32	34	33	34	33	33	34
NextQuestion	5	5	12	12	12	12	11	11	11	11	11
AnswerCheck-Q2	13	12	33	33	34	34	33	35	36	33	34

Minimum Response time(ms)											
No. of threads:	30	100	200	300	400	500	600	700	800	900	1000
StartQuiz	118	125	589	850	1108	1281	1453	1359	1451	1411	1348
Authentication	9	9	11	11	11	10	11	10	8	8	8
AnswerCheck-Q1	12	12	12	13	16	13	14	17	13	14	13
NextQuestion	5	4	4	4	4	4	4	4	4	4	4
AnswerCheck-Q2	12	12	12	12	13	13	13	13	13	13	13

Maximum Response time(ms)											
No. of threads:	30	100	200	300	400	500	600	700	800	900	1000
StartQuiz	193	243	10674	30425	50418	70549	90810	110798	111148	151857	171796
Authentication	15	16	94	96	91	93	187	110	127	97	113
AnswerCheck-Q1	14	27	81	108	83	113	123	111	101	119	137
NextQuestion	7	15	58	62	49	49	60	65	56	59	76
AnswerCheck-Q2	23	29	80	115	89	119	126	110	108	101	145

Table 3 : Mean response time in Local server using C quizzes

As can been be seen from Table 3, Mean response time increases rapidly from 100 threads to 200 threads. To narrow down the specific number of thread at which the response time is increased rapidly we conducted experiments with threads in the range of 110-190 with difference of 10 as shown in Graph 1. We found that at 170 threads the mean response time increases rapidly.



Graph 1 : Increase in mean response time at Quiz Start for 170 threads.

We conducted experiments using different languages for the CQG quizzes (C, Python and Java). Table 4 shows the difference in the response time for these languages. It was found that the answer checking is expensive in Java quizzes as compared to C and Python. The difference between *mean response* times can be seen from the rows with label *AnswerCheck-Q1* and *AnswerCheck-Q1* in section 1, 2 and 3.

Mean response time(ms) for C quiz											
No. of threads:	30	100	200	300	400	500	600	700	800	900	1000
StartQuiz	121	129	5826	15807	25911	35985	46078	56173	56450	76810	86902
Authentication	10	9	28	28	30	29	29	29	28	29	28
AnswerCheck-Q1	**12**	**18**	**31**	**34**	**32**	**34**	**33**	**34**	**33**	**33**	**34**
NextQuestion	5	5	12	12	12	12	11	11	11	11	11
AnswerCheck-Q2	**13**	**12**	**33**	**33**	**34**	**34**	**33**	**35**	**36**	**33**	**34**

Mean response time(ms) for Python quiz											
No. of threads:	30	100	200	300	400	500	600	700	800	900	1000
StartQuiz	121	131	4970	14703	24310	34002	43903	53512	63355	73304	83108
Authentication	10	9	29	29	27	28	29	29	27	29	28
AnswerCheck-Q1	**26**	**33**	**54**	**48**	**51**	**51**	**51**	**52**	**52**	**51**	**51**
NextQuestion	5	4	11	11	11	11	11	12	12	11	11
AnswerCheck-Q2	**5**	**4**	**15**	**13**	**13**	**12**	**13**	**14**	**13**	**14**	**12**

Mean response time(ms) for Java quiz											
No. of threads:	30	100	200	300	400	500	600	700	800	900	1000
StartQuiz	122	150	5631	16151	26950	37137	48793	59459	70023	79332	88976
Authentication	10	10	33	30	31	32	32	32	33	33	33
AnswerCheck-Q1	**108**	**127**	**368**	**381**	**376**	**382**	**386**	**389**	**408**	**384**	**379**
NextQuestion	5	4	11	11	12	12	11	12	378	12	11
AnswerCheck-Q2	**109**	**142**	**363**	**372**	**368**	**380**	**380**	**383**	**397**	**378**	**375**

Table 4: Difference between mean response time for C, Python and Java quizzes on Local server.

To check if quiz logging has any effect on performance we conducted experiments using C language quizzes on SEng server. Table 6 shows the results with quiz logging and without it.

Mean response time(ms) **without quiz logging**					
No. of Threads	30	100	400	700	1000
StartQuiz	115	120	18680	39211	62497
Authentication	31	31	106	104	103
AnswerCheck-Q1	26	45	121	124	124
NextQuestion	13	14	56	50	47
AnswerCheck-Q2	27	45	135	131	130
Mean response time (ms) **with quiz logging**					
No. of Threads	30	100	400	700	1000
StartQuiz	150	180	18710	39856	63917
Authentication	36	38	120	113	119
AnswerCheck-Q1	31	51	134	126	131
NextQuestion	16	16	51	49	54
AnswerCheck-Q2	30	45	137	134	141

Table 5: Difference in response time for quizzes with and without logging

As discussed earlier in the experimental design, we used two kinds of test setups (Local server and SEng server). We conducted identical experiments on these setups to identify if the performance of CQG is dependent on the server configuration. Table 7 confirms our hypothesis that performance is related to server configuration as the mean response time is different. It was found that the mean time to start up the quiz is less in SEng server as compared to the local server. However careful analysis confirms that after the quiz start up performance is low in SEng server.

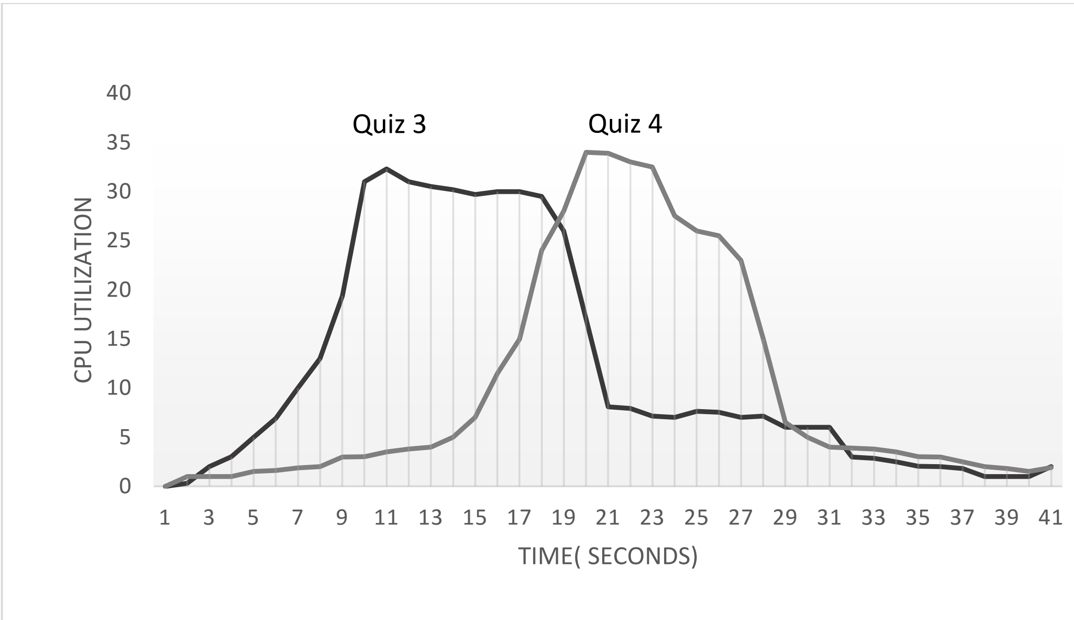
Mean response time(ms) for C quizzes on Local Server											
No. of threads:	30	100	200	300	400	500	600	700	800	900	1000
StartQuiz	121	129	5826	15807	25911	35985	46078	56173	56450	76810	86902
Authentication	10	9	28	28	30	29	29	29	28	29	28
AnswerCheck-Q1	12	18	31	34	32	34	33	34	33	33	34
NextQuestion	5	5	12	12	12	12	11	11	11	11	11
AnswerCheck-Q2	13	12	33	33	34	34	33	35	36	33	34

Mean response time(ms) for C quizzes on Seng Server											
No. of threads:	30	100	200	300	400	500	600	700	800	900	1000
StartQuiz	115	120	4147	8628	18680	26462	30753	39211	48743	52524	62497
Authentication	31	31	124	106	106	110	107	104	103	102	103
AnswerCheck-Q1	26	45	150	121	121	125	130	124	122	120	124
NextQuestion	13	14	52	51	56	49	47	50	47	47	47
AnswerCheck-Q2	27	45	150	129	135	134	133	131	127	125	130

Table 6: Difference in the performance of Local and SEng server

To identify the CPU utilization of the CQG quizzes we captured the server's CPU usage. CPU information was captured on the SEng server while students were doing the marked quiz in

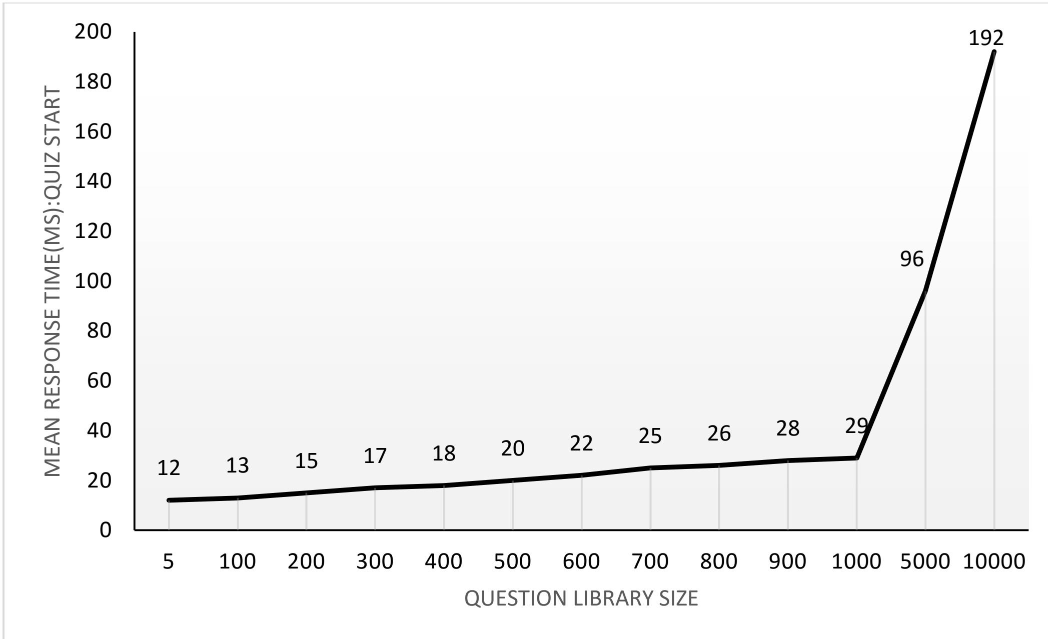
SEng 265. We found that CPU usage increases rapidly at the start of the quiz. Graph 2 shows the CPU utilization on the SEng server while running quizzes for approximately 35 students.



Graph 2: CPU Utilization over time

In the end, we want to identify and reduce the high quiz start-up time. To identify that we conducted experiments by varying the question library size. It was our hypothesis that question library size was closely related to the high quiz start up times.

We conducted experiments with 50 threads using quizzes for C. We found that quiz start up time is related to the size of the question library. Graph 2 shows the results from the experiments.



Graph 3: Relation between question library size and mean response time for quiz start

5.0 Conclusion

We can conclude from the performance testing of CQG that:

- It can support up to roughly 1000 users stably. The only bottleneck of high quiz start up time while increasing number of users can be minimized by choosing a small and effective question library.
- Java questions are expensive and should be used while considering the low performance of answer checking.
- Quiz logging has minimal effect on the performance of CQG.
- Configuration of server setup (RAM, disk) should be chosen according with expected number of user and load.

6.0 Future Work

We know that the quiz start up time is high when the question library size is big. In the future CQG quiz start up time can be reduced by identifying the cause and applying the appropriate patch. Each patch can be measured using the test framework implemented in this project until the caused is identified.

Furthermore, other quiz types like networking and multiple choice can be measured and analysed as our hypothesis is that these quiz types are fast and cheap.

7.0 References

The sources are listed in the order in which they are cited in the report, as in the following book and article.

[1] Apache Project : http://www.apache.org/
[2] http://www.webopedia.com/TERM/T/TCP.html
[3] https://en.wikipedia.org/w/index.php?title=Hypertext_Transfer_Protocol&redirect=no
[4] https://en.wikipedia.org/wiki/Protocol_data_unit