Table of Contents

1. Main Components 3

1.1. Web App Structure 3

1.2. Benchmarking 5

2. Service Time and Service Demand 5

3. Model Validation 6

3.1. Throughput 6

3.2. Response time 11

4. Bottleneck and Optimal number of users 15

5. Improvement of Scalability level 15

5.1. Scaling 15

5.2. Caching 16

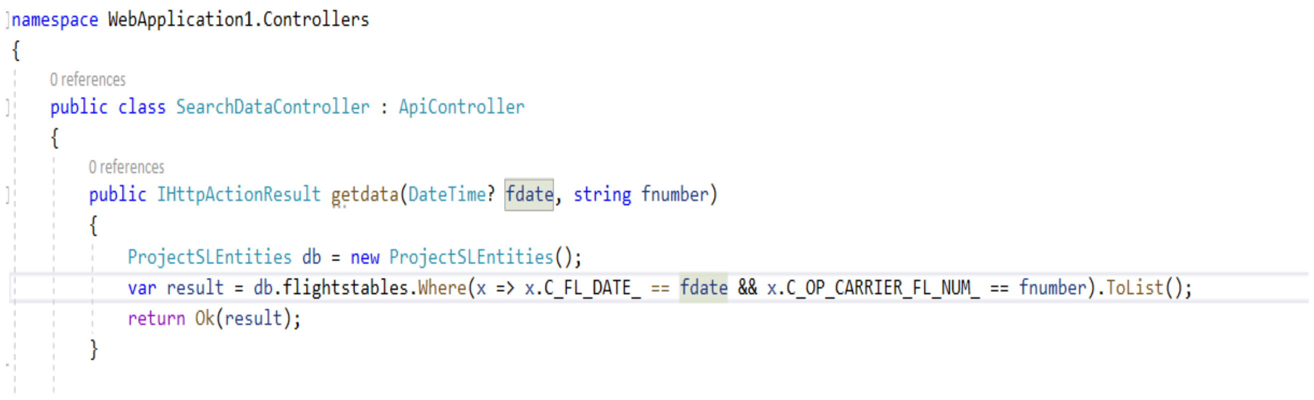
6. Conclusion 17

1. Main Components

1.1. Web App Structure

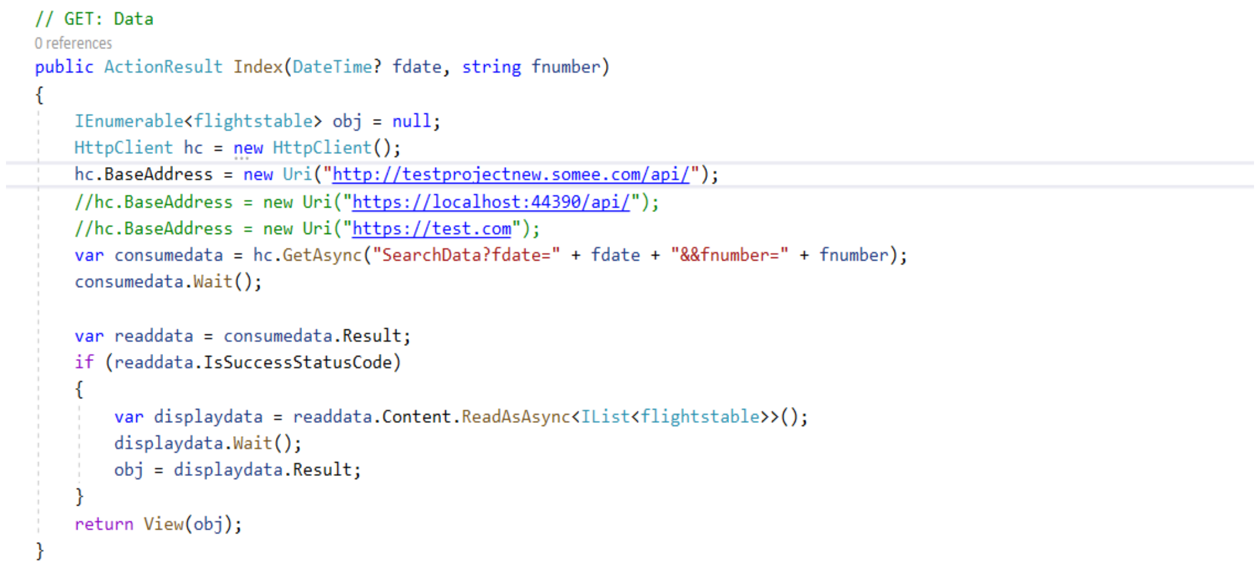
Our application is monolithic, which is hosted at a remote server. For front end we have we have used Asp.net MVC 5 framework with web API and for data storage we have used SQL. All of the data is hosted at a single server.

We have 3 different Queries, for the first query the user can search the record on the basis of Date and Flight number as shown in the fig 1.1(a) and the corresponding web API code is shown in the fig 1.1(b).



```
namespace WebApplication1.Controllers
{
    0 references
    public class SearchDataController : ApiController
    {
        0 references
        public IHttpActionResult getdata(DateTime? fdate, string fnumber)
        {
            ProjectSLEntities db = new ProjectSLEntities();
            var result = db.flightstables.Where(x => x.C_FL_DATE_ == fdate && x.C_OP_CARRIER_FL_NUM_ == fnumber).ToList();
            return Ok(result);
        }
```

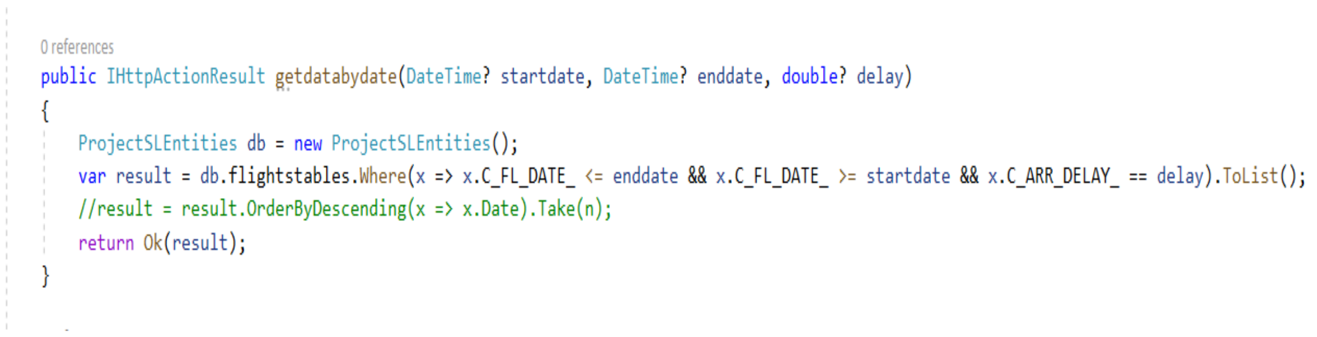
Fig 1.1(a)



```
// GET: Data
0 references
public ActionResult Index(DateTime? fdate, string fnumber)
{
    IEnumerable<flightstable> obj = null;
    HttpClient hc = new HttpClient();
    hc.BaseAddress = new Uri("http://testprojectnew.somee.com/api/");
    //hc.BaseAddress = new Uri("https://localhost:44390/api/");
    //hc.BaseAddress = new Uri("https://test.com");
    var consumedata = hc.GetAsync("SearchData?fdate=" + fdate + "&&fnumber=" + fnumber);
    consumedata.Wait();

    var readdata = consumedata.Result;
    if (readdata.IsSuccessStatusCode)
    {
        var displaydata = readdata.Content.ReadAsAsync<IList<flightstable>>();
        displaydata.Wait();
        obj = displaydata.Result;
    }
    return View(obj);
}
```

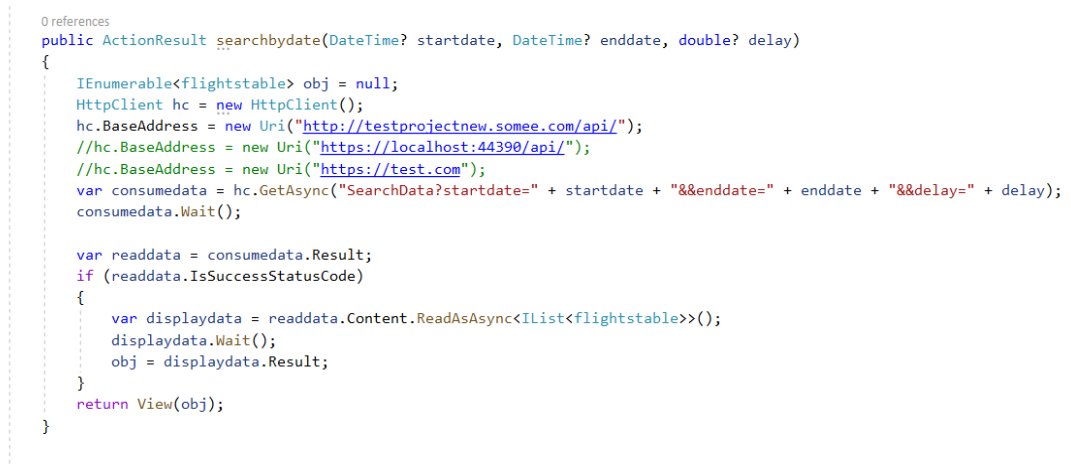
Fig 1.1(b)

For the second query the user can search the record on the basis of Date interval e.g. start date, end date and flight delay as shown in the fig 1.1(c) and the corresponding web API code is shown in the fig 1.1(d).



```
0 references
public IHttpActionResult getdatabydate(DateTime? startdate, DateTime? enddate, double? delay)
{
    ProjectSLEntities db = new ProjectSLEntities();
    var result = db.flightstables.Where(x => x.C_FL_DATE_ <= enddate && x.C_FL_DATE_ >= startdate && x.C_ARR_DELAY_ == delay).ToList();
    //result = result.OrderByDescending(x => x.Date).Take(n);
    return Ok(result);
}
```

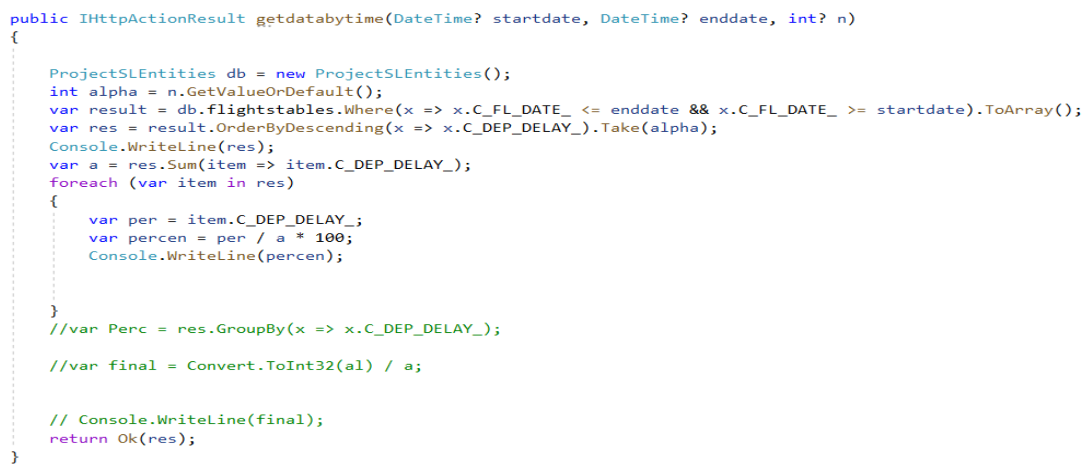
Fig 1.1(c)



```
0 references
public ActionResult searchbydate(DateTime? startdate, DateTime? enddate, double? delay)
{
    IEnumerable<flightstable> obj = null;
    HttpClient hc = new HttpClient();
    hc.BaseAddress = new Uri("http://testprojectnew.somee.com/api/");
    //hc.BaseAddress = new Uri("https://localhost:44390/api/");
    //hc.BaseAddress = new Uri("https://test.com");
    var consumedata = hc.GetAsync("SearchData?startdate=" + startdate + "&&enddate=" + enddate + "&&delay=" + delay);
    consumedata.Wait();

    var readdata = consumedata.Result;
    if (readdata.IsSuccessStatusCode)
    {
        var displaydata = readdata.Content.ReadAsAsync<IList<flightstable>>();
        displaydata.Wait();
        obj = displaydata.Result;
    }
    return View(obj);
}
```

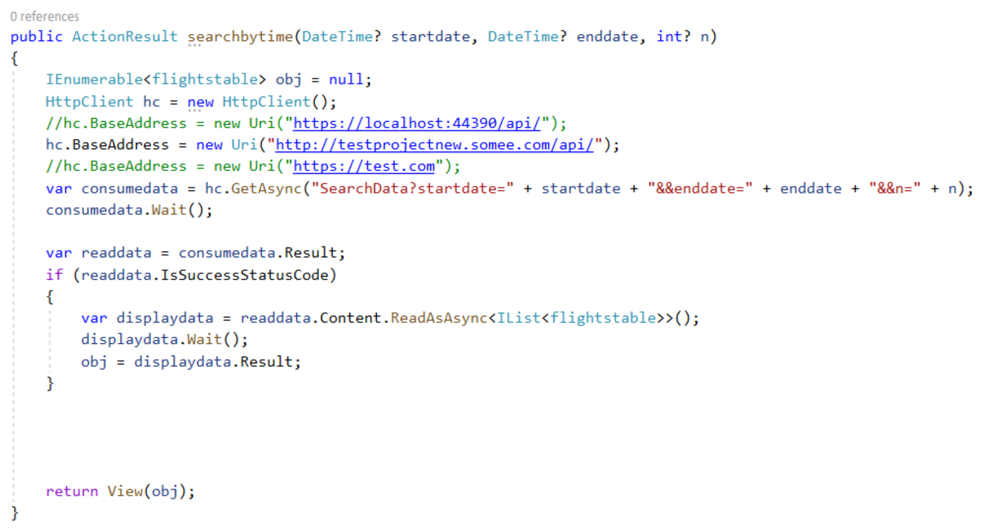
Fig 1.1(d)

For the third query the user can search the record on the basis of Date interval e.g. start date, end date and the number of records to be shown specified by number n in descending order e.g. departure delay as shown in the fig 1.1(e) and the corresponding web API code is shown in the fig 1.1(f).



```
public IHttpActionResult getdatabytime(DateTime? startdate, DateTime? enddate, int? n)
{

    ProjectSLEntities db = new ProjectSLEntities();
    int alpha = n.GetValueOrDefault();
    var result = db.flightstables.Where(x => x.C_FL_DATE_ <= enddate && x.C_FL_DATE_ >= startdate).ToArray();
    var res = result.OrderByDescending(x => x.C_DEP_DELAY_).Take(alpha);
    Console.WriteLine(res);
    var a = res.Sum(item => item.C_DEP_DELAY_);
    foreach (var item in res)
    {
        var per = item.C_DEP_DELAY_;
        var percen = per / a * 100;
        Console.WriteLine(percen);


    }
    //var Perc = res.GroupBy(x => x.C_DEP_DELAY_);

    //var final = Convert.ToInt32(al) / a;


    // Console.WriteLine(final);
    return Ok(res);
}
```

Fig 1.1(e)



```
0 references
public ActionResult searchbytime(DateTime? startdate, DateTime? enddate, int? n)
{
    IEnumerable<flightstable> obj = null;
    HttpClient hc = new HttpClient();
    //hc.BaseAddress = new Uri("https://localhost:44390/api/");
    hc.BaseAddress = new Uri("http://testprojectnew.somee.com/api/");
    //hc.BaseAddress = new Uri("https://test.com");
    var consumedata = hc.GetAsync("SearchData?startdate=" + startdate + "&&enddate=" + enddate + "&&n=" + n);
    consumedata.Wait();

    var readdata = consumedata.Result;
    if (readdata.IsSuccessStatusCode)
    {
        var displaydata = readdata.Content.ReadAsAsync<IList<flightstable>>();
        displaydata.Wait();
        obj = displaydata.Result;
    }



    return View(obj);
}
```

Fig 1.1(f)

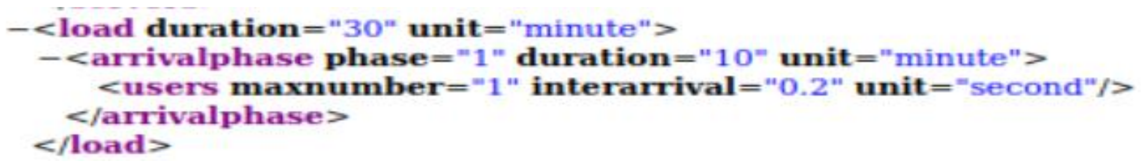
1.2. Benchmarking

In order to perform benchmarking experiment on the system we have used Tsung. To perform the benchmarking test we have divided our application into three stations.

- Infinite Delay server (Reference station)
- Load independent server
- Infinite Delay server

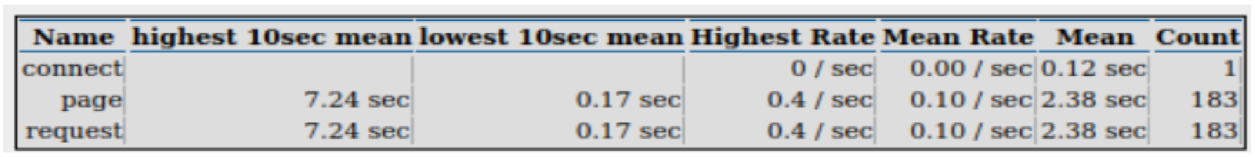
2. Service Time and Service Demand

For this part we set up a proxy in browser manually, first of all to record the sessions in xml file. Then we set up the main xml file by setting up the number of customers to 1 and load time to 30 minutes and also put the session data in the file to run the close loop test using Tsung.



```
–<load duration="30" unit="minute">
  –<arrivalphase phase="1" duration="10" unit="minute">
      <users maxnumber="1" interarrival="0.2" unit="second"/>
    </arrivalphase>
  </load>
```

After the completion of the test we get, the service time=2.38s



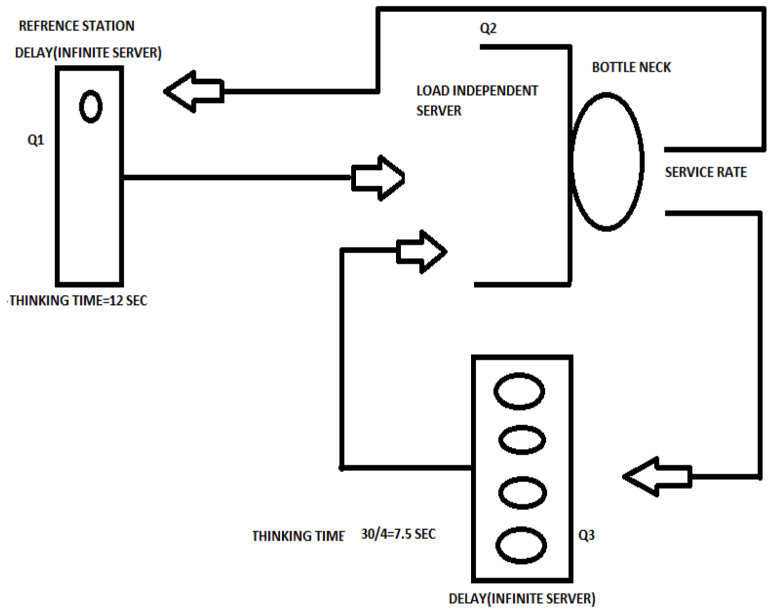
Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean Rate	Mean	Count
connect			0 / sec	0.00 / sec	0.12 sec	1
page	7.24 sec	0.17 sec	0.4 / sec	0.10 / sec	2.38 sec	183
request	7.24 sec	0.17 sec	0.4 / sec	0.10 / sec	2.38 sec	183

Next we calculated the Relative Visit Ratio,

Relative Visit Ratio=Number of Requests/load duration=183/30=6.1

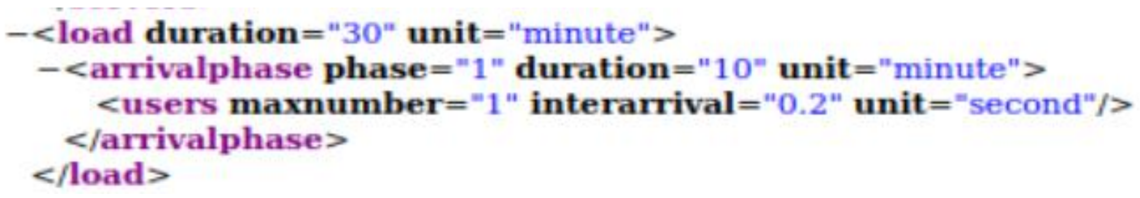
Similarly, we calculated Service Demand, Service Demand=6.1x2.38=14.51

3. Model Validation



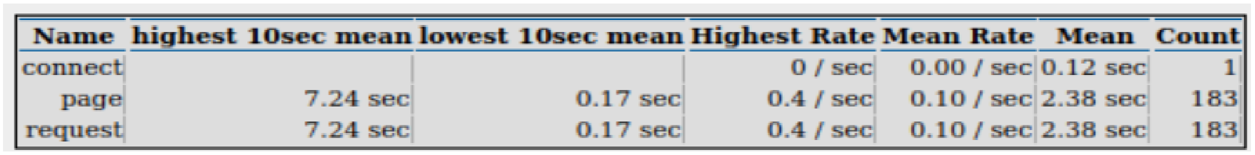
3.1. Throughput

Throughput can be defined as the number of requests a server can processes in a given unit of time. For 1 to 4 customers using Java modelling tool. We have set the service times 12s for the reference station which is the thinking time of the 1st station Q1 and the value of Visit would be 1. For the second station which is also the bottleneck of the system the service time is 2.38s which we got from Tsung and the relative visit ratio would be number of requests/load duration which we got by using Tsung which is equal to 183/30=6.1. Finally, for the last station which is also the infinite delay server. The service time would be 30/4=7.5s, which is the average of the thinking time excluding the first thinking time and the visit value would be 4.



```
-<load duration="30" unit="minute">
  -<arrivalphase phase="1" duration="10" unit="minute">
      <users maxnumber="1" interarrival="0.2" unit="second"/>
    </arrivalphase>
  </load>
```

After completion of the test by Tsung, we get the following values



Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean Rate	Mean	Count
connect			0 / sec	0.00 / sec	0.12 sec	1
page	7.24 sec	0.17 sec	0.4 / sec	0.10 / sec	2.38 sec	183
request	7.24 sec	0.17 sec	0.4 / sec	0.10 / sec	2.38 sec	183

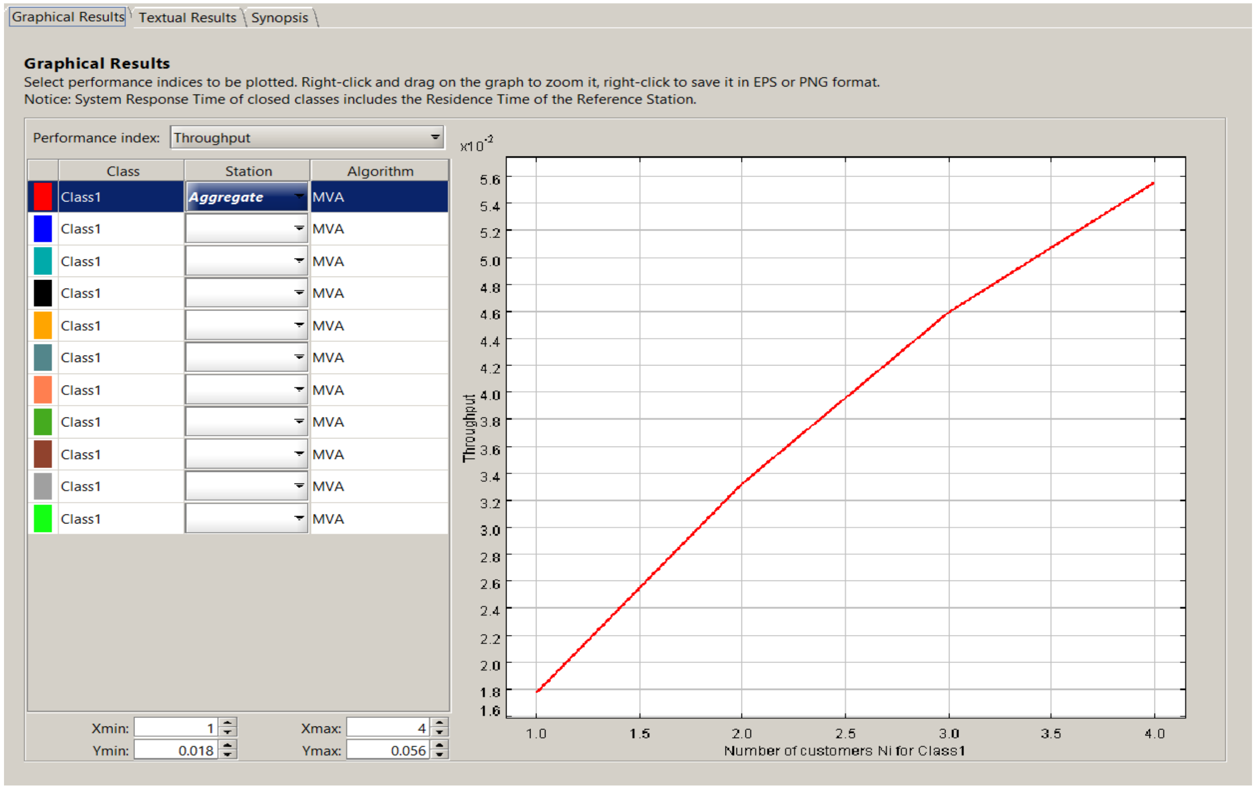
The service time=2.38s

Next we calculated the Relative Visit Ratio

Relative Visit Ratio=Number of Requests/load duration=183/30=6.1

Similarly, we calculated Service Demand, Service Demand=6.1x2.38=14.51

For customers 1 to 4. We got the following graph fig 3.1(a) and 3.1(b) for throughput using the Java modelling toot.



Graphical Results
Select performance indices to be plotted. Right-click and drag on the graph to zoom it, right-click to save it in EPS or PNG format.
Notice: System Response Time of closed classes includes the Residence Time of the Reference Station.

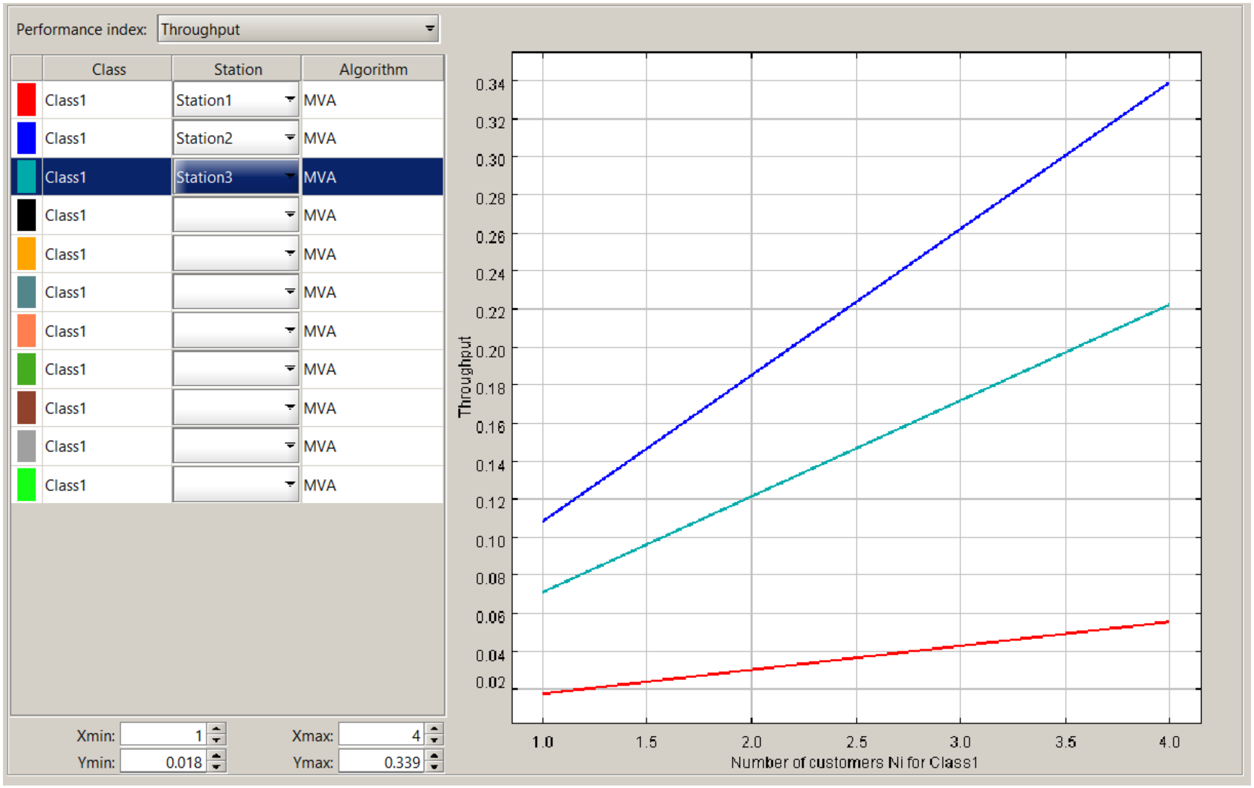
Performance index: Throughput

	Class	Station	Algorithm
🟥	Class1	Aggregate	MVA
🟦	Class1		MVA
🟦	Class1		MVA
⬛	Class1		MVA
🟧	Class1		MVA
🟦	Class1		MVA
🟧	Class1		MVA
🟩	Class1		MVA
🟫	Class1		MVA
⬜	Class1		MVA
🟩	Class1		MVA

Xmin: 1 Xmax: 4
Ymin: 0.018 Ymax: 0.056

Fig 3.1(a)

This Throughput is growing linearly with the number of jobs because the system has not reached its maximum capacity. This is shown in the fig 3.1(a) and 3.1(b)



Performance index: Throughput

	Class	Station	Algorithm
🟥	Class1	Station1	MVA
🟦	Class1	Station2	MVA
🟦	Class1	Station3	MVA
⬛	Class1		MVA
🟧	Class1		MVA
🟦	Class1		MVA
🟧	Class1		MVA
🟩	Class1		MVA
🟫	Class1		MVA
⬜	Class1		MVA
🟩	Class1		MVA

Xmin: 1 Xmax: 4
Ymin: 0.018 Ymax: 0.339

3.1(b)

By increasing the number of customers from 4 to 6, we got the graph 3.1(c). We can also see the saturation as number of jobs grows.

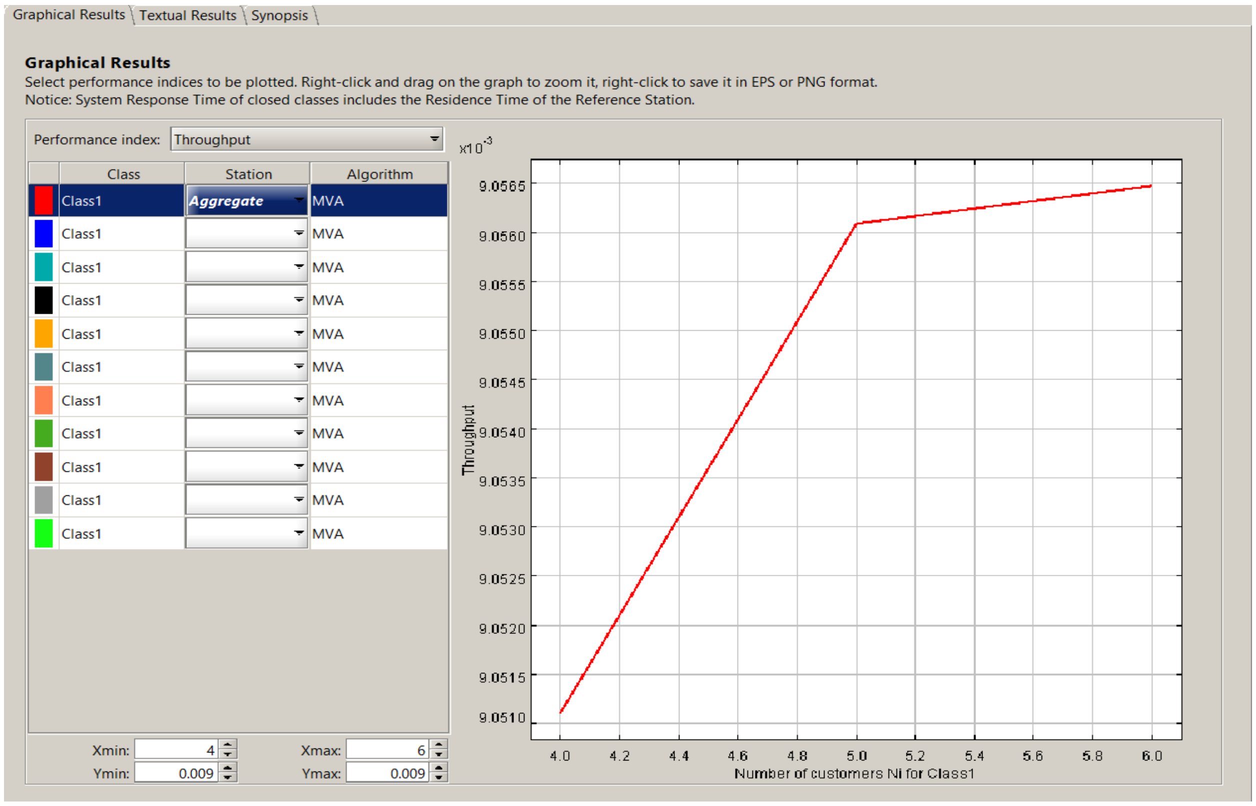
```
–<load duration="30" unit="minute">
  –<arrivalphase phase="1" duration="10" unit="minute">
      <users maxnumber="4" interarrival="0.2" unit="second"/>
    </arrivalphase>
  </load>
```

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean Rate	Mean	Count
connect			0 / sec	0.00 / sec	0.13 sec	4
page	57.63 sec	0.18 sec	0.9 / sec	0.28 / sec	6.60 sec	502
request	57.63 sec	0.18 sec	0.9 / sec	0.28 / sec	6.60 sec	502

Service rate=6.60s

Relative Visit Ratio=Number of Requests/load duration=502/30=16.73

Service Demand=16.73x6.60=110.418s



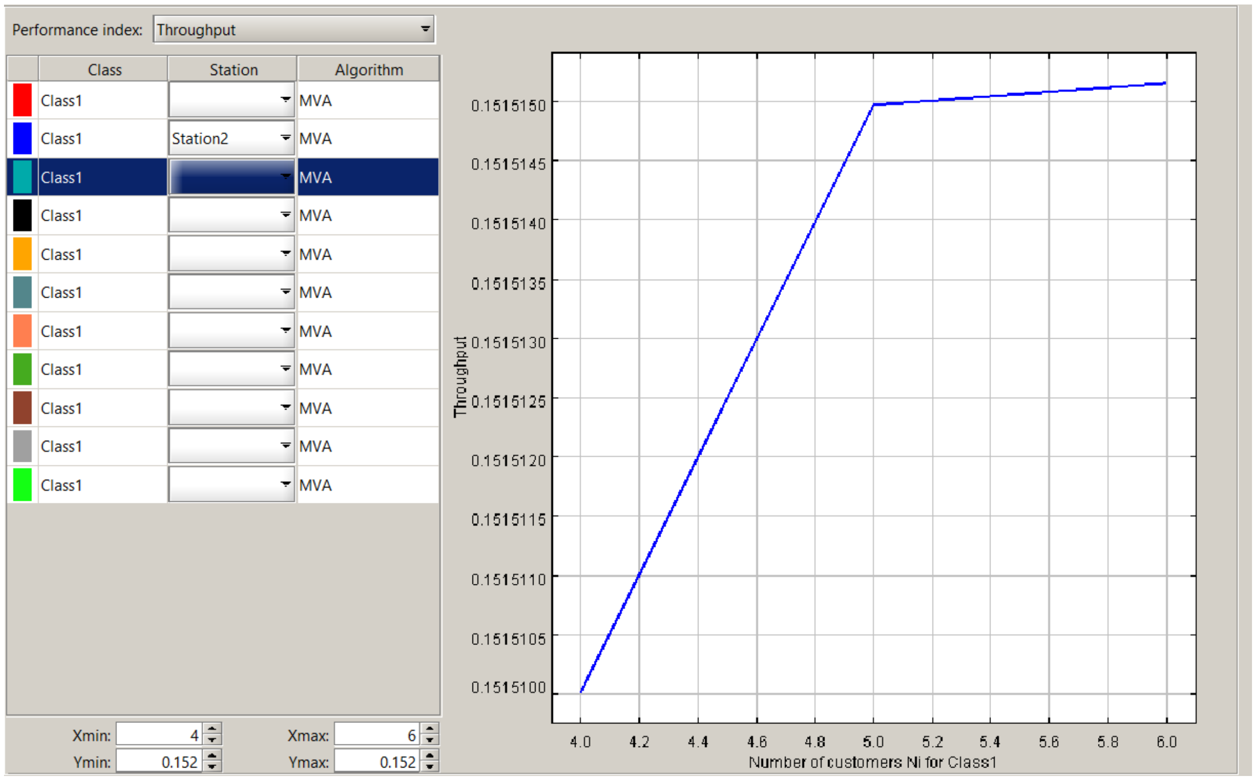
Graphical Results
Select performance indices to be plotted. Right-click and drag on the graph to zoom it, right-click to save it in EPS or PNG format.
Notice: System Response Time of closed classes includes the Residence Time of the Reference Station.

Performance index: Throughput

	Class	Station	Algorithm
🟥	Class1	Aggregate	MVA
🟦	Class1		MVA
🟦	Class1		MVA
⬛	Class1		MVA
🟧	Class1		MVA
⬛	Class1		MVA
🟧	Class1		MVA
🟩	Class1		MVA
🟫	Class1		MVA
⬜	Class1		MVA
🟩	Class1		MVA

Xmin: 4 Xmax: 6
Ymin: 0.009 Ymax: 0.009

Fig 3.1(c)



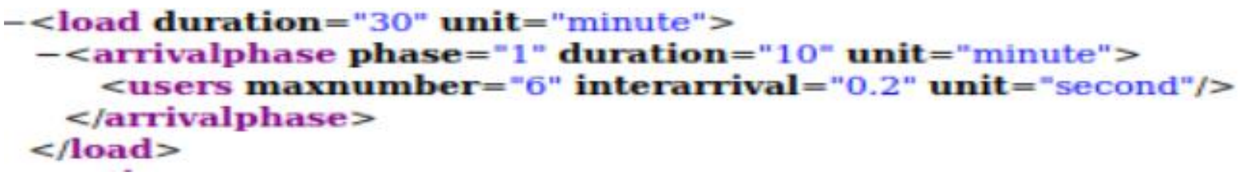
Performance index: Throughput

	Class	Station	Algorithm
🟥	Class1		MVA
🟦	Class1	Station2	MVA
🟦	Class1		MVA
⬛	Class1		MVA
🟧	Class1		MVA
⬛	Class1		MVA
🟧	Class1		MVA
🟩	Class1		MVA
🟫	Class1		MVA
⬜	Class1		MVA
🟩	Class1		MVA

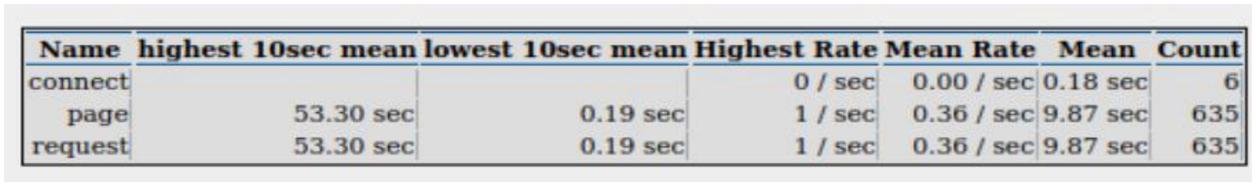
Xmin: 4 Xmax: 6
Ymin: 0.152 Ymax: 0.152

Fig 3.1(d)

In case, if we further increase the value of number of customers to 6 we get the following data.



```
-<load duration="30" unit="minute">
  -<arrivalphase phase="1" duration="10" unit="minute">
      <users maxnumber="6" interarrival="0.2" unit="second"/>
    </arrivalphase>
  </load>
```



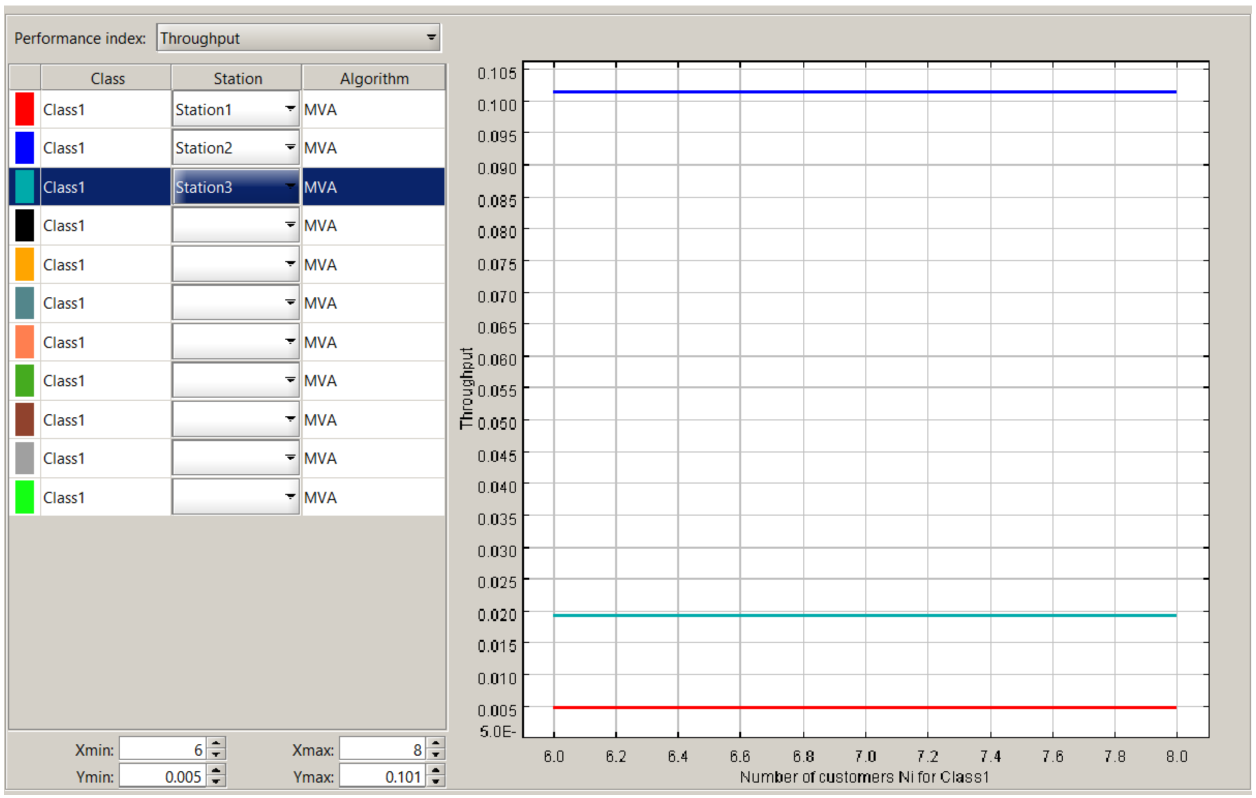
Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean Rate	Mean	Count
connect			0 / sec	0.00 / sec	0.18 sec	6
page	53.30 sec	0.19 sec	1 / sec	0.36 / sec	9.87 sec	635
request	53.30 sec	0.19 sec	1 / sec	0.36 / sec	9.87 sec	635

Service rate=9.87s

Relative Visit Ratio=Number of Requests/load duration=635/30=21.16

Service Demand=21.16.73x9.87=208.84s

By using the JMT modelling model by using the customer's parameters from 6 to 8 we get the following graphs as shown in the fig 3.1(e). The growth is horizontal. This is due to the fact that the system is beyond its maximum capacity.



Performance index: Throughput

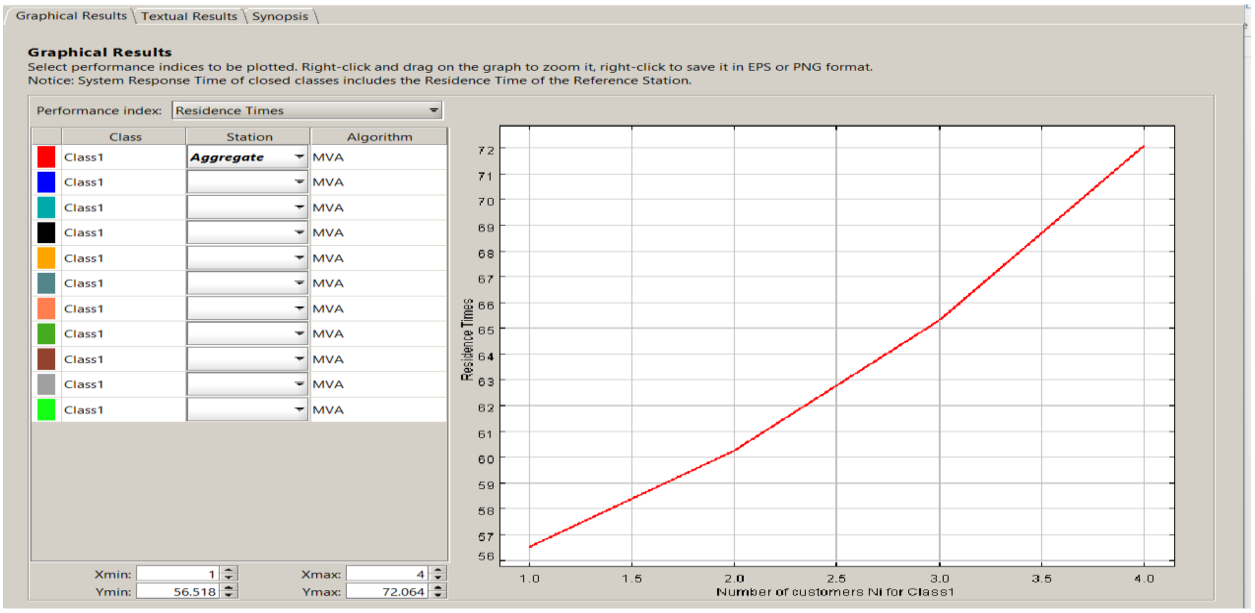
	Class	Station	Algorithm
🟥	Class1	Station1	MVA
🟦	Class1	Station2	MVA
🟦	Class1	Station3	MVA
⬛	Class1		MVA
🟧	Class1		MVA
🟦	Class1		MVA
🟧	Class1		MVA
🟩	Class1		MVA
🟫	Class1		MVA
⬜	Class1		MVA
🟩	Class1		MVA

Xmin: 6 Xmax: 8
Ymin: 0.005 Ymax: 0.101

Fig 3.1(e)

3.2. Response time

Response time is the time, required between when the user starts an action, and when the system starts to display the result. The response time shown by JMT for 1 to 4 customers is shown the fig. 3.2(a).



Graphical Results
Select performance indices to be plotted. Right-click and drag on the graph to zoom it, right-click to save it in EPS or PNG format.
Notice: System Response Time of closed classes includes the Residence Time of the Reference Station.

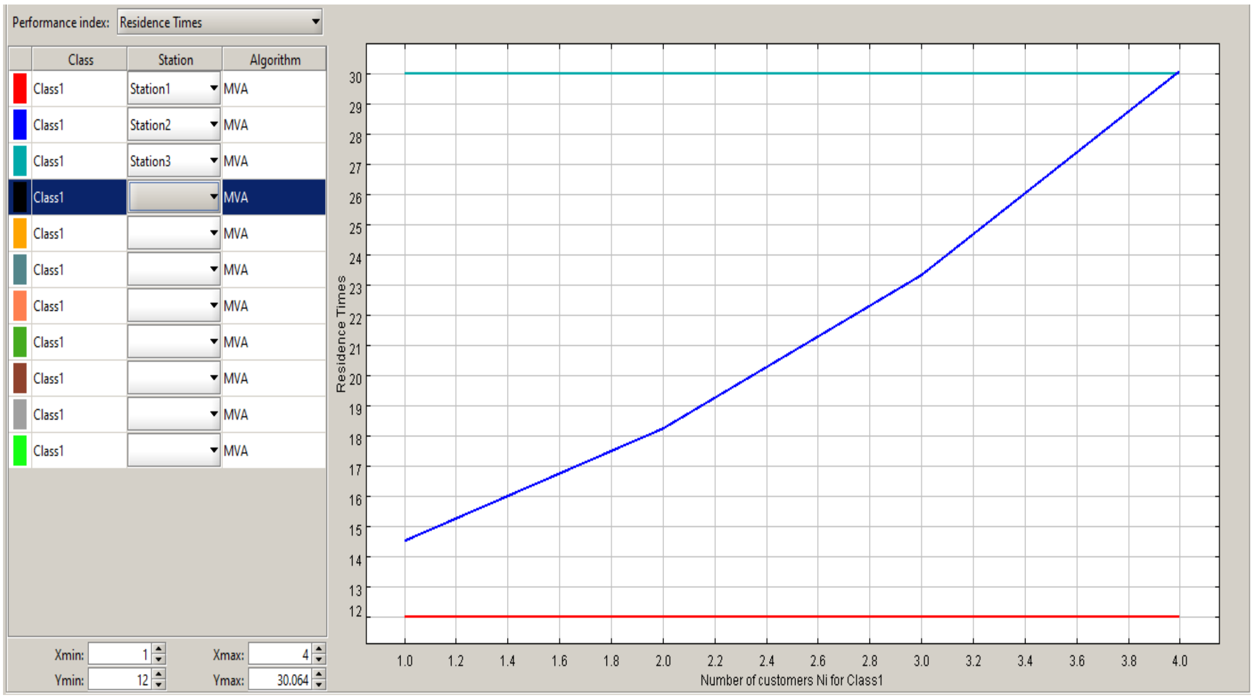
Performance index: Residence Times

	Class	Station	Algorithm
🟥	Class1	*Aggregate*	MVA
🟦	Class1		MVA
🟦	Class1		MVA
⬛	Class1		MVA
🟧	Class1		MVA
🟦	Class1		MVA
🟧	Class1		MVA
🟩	Class1		MVA
🟫	Class1		MVA
⬜	Class1		MVA
🟩	Class1		MVA

Xmin: 1 Xmax: 4
Ymin: 56.518 Ymax: 72.064

Fig 3.2(a)

Station 1 is constant because it is infinite server. Station 3 is also constant which is also infinite server. The response time at the Station 2 is increasing linearly which is also the bottleneck.



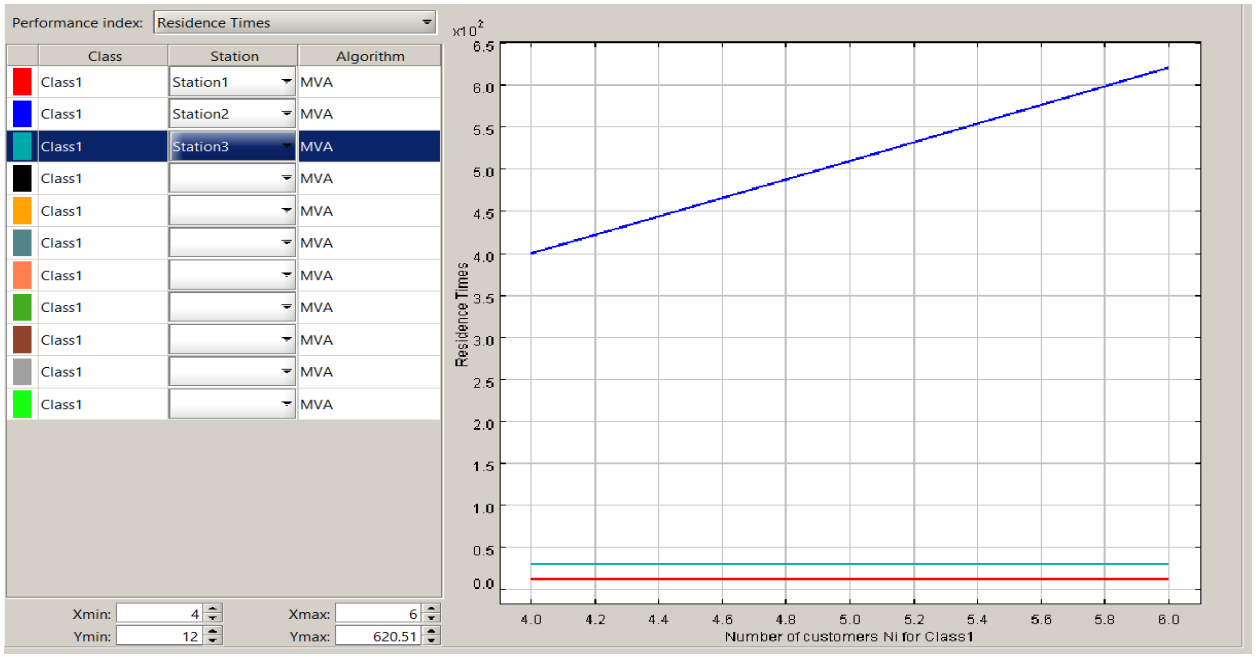
Performance index: Residence Times

	Class	Station	Algorithm
🟥	Class1	Station1	MVA
🟦	Class1	Station2	MVA
🟦	Class1	Station3	MVA
⬛	Class1		MVA
🟧	Class1		MVA
🟦	Class1		MVA
🟧	Class1		MVA
🟩	Class1		MVA
🟫	Class1		MVA
⬜	Class1		MVA
🟩	Class1		MVA

Xmin: 1 Xmax: 4
Ymin: 12 Ymax: 30.064

Fig 3.2(b)

By increasing the number of customers from 4 to 6 we got the graph 3.2(c) as we can see at the bottleneck station the response time is increasing while it is constant at the other two stations.



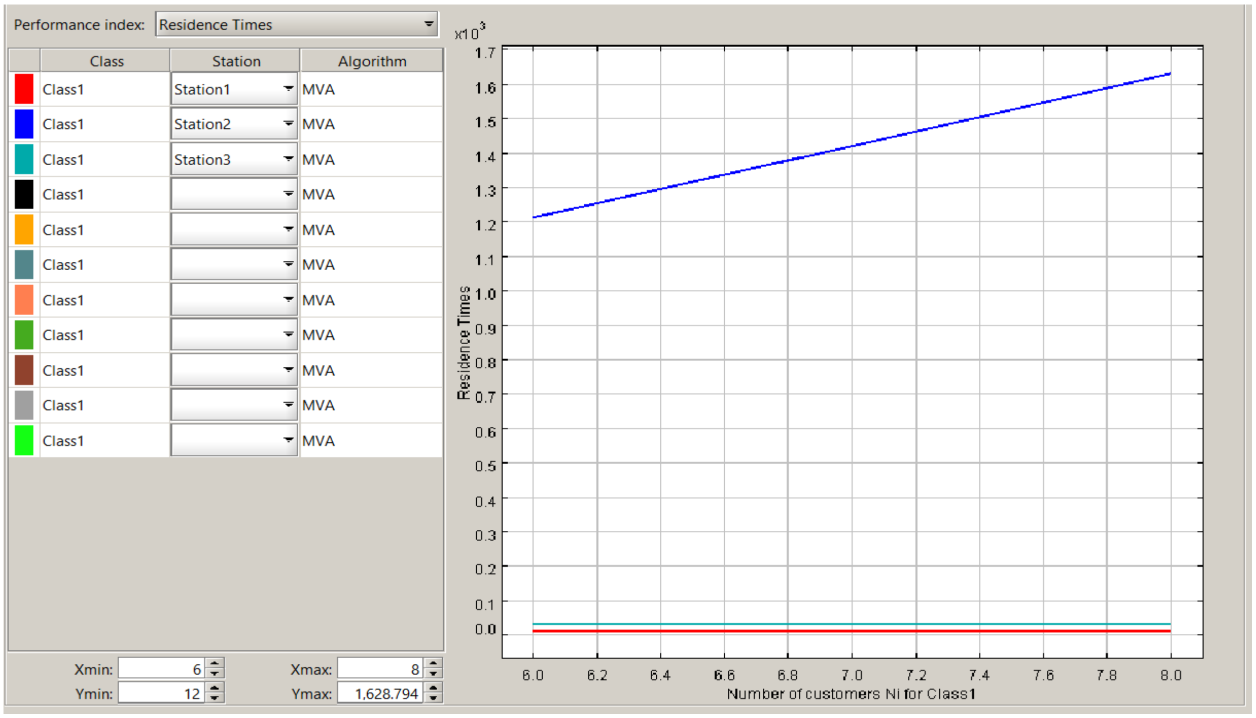
Performance index: Residence Times

	Class	Station	Algorithm
🟥	Class1	Station1	MVA
🟦	Class1	Station2	MVA
🟦	Class1	Station3	MVA
⬛	Class1		MVA
🟧	Class1		MVA
🟦	Class1		MVA
🟧	Class1		MVA
🟩	Class1		MVA
🟫	Class1		MVA
⬜	Class1		MVA
🟩	Class1		MVA

Xmin: 4 Xmax: 6
Ymin: 12 Ymax: 620.51

Fig 3.2(c)

Similarly, by increasing the customers from 6 to 8 we got the graph 3.2(d) as the response time keeps on increasing. Station 1 and Station 2 are constant.



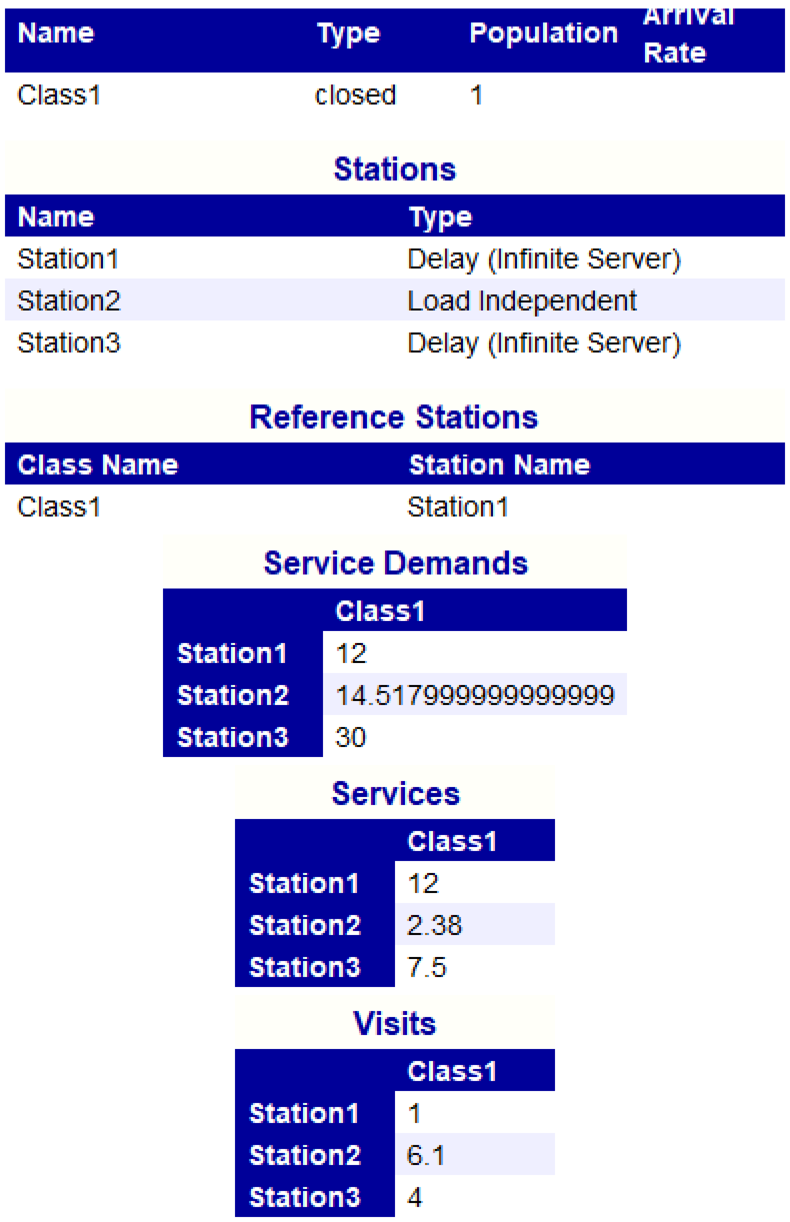
Performance index: Residence Times

	Class	Station	Algorithm
🟥	Class1	Station1	MVA
🟦	Class1	Station2	MVA
🟦	Class1	Station3	MVA
⬛	Class1		MVA
🟧	Class1		MVA
🟦	Class1		MVA
🟧	Class1		MVA
🟩	Class1		MVA
🟫	Class1		MVA
⬜	Class1		MVA
🟩	Class1		MVA

Xmin: 6 Xmax: 8
Ymin: 12 Ymax: 1,628.794

Fig 3.2(d)

Service demand calculated for 1 customer by using values of Tsung is 14.51, which is same in JMT as well.



Name	Type	Population	Arrival Rate
Class1	closed	1	

Stations

Name	Type
Station1	Delay (Infinite Server)
Station2	Load Independent
Station3	Delay (Infinite Server)

Reference Stations

Class Name	Station Name
Class1	Station1

Service Demands

	Class1
Station1	12
Station2	14.517999999999999
Station3	30

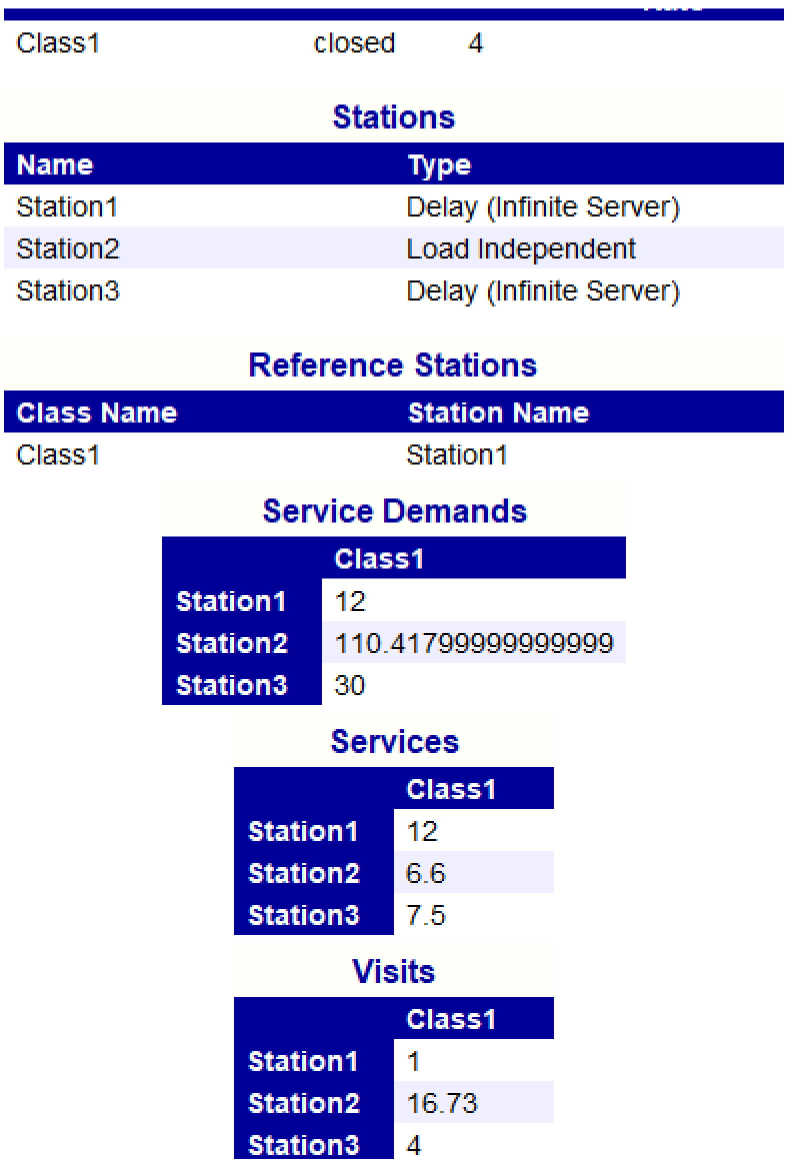
Services

	Class1
Station1	12
Station2	2.38
Station3	7.5

Visits

	Class1
Station1	1
Station2	6.1
Station3	4

Service demand calculated by using values of Tsung is 110.418s in case of 4 customers, which is same in JMT as well.



Class1	closed	4

Stations

Name	Type
Station1	Delay (Infinite Server)
Station2	Load Independent
Station3	Delay (Infinite Server)

Reference Stations

Class Name	Station Name
Class1	Station1

Service Demands

	Class1
Station1	12
Station2	110.41799999999999
Station3	30

Services

	Class1
Station1	12
Station2	6.6
Station3	7.5

Visits

	Class1
Station1	1
Station2	16.73
Station3	4

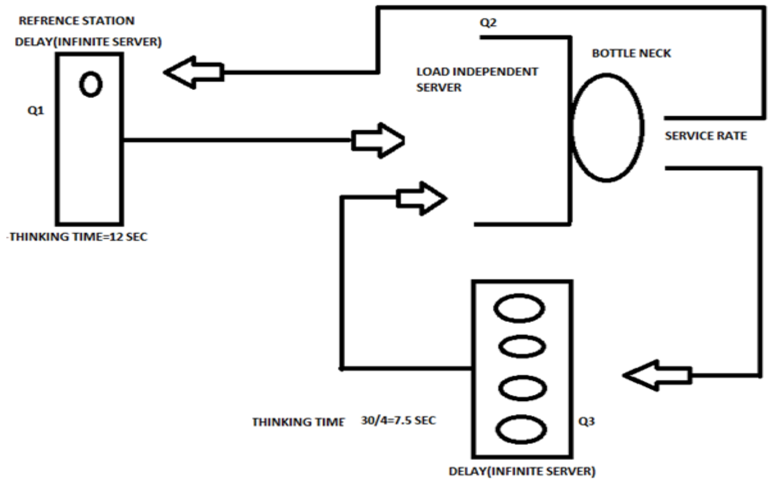
4. Bottleneck and Optimal number of users

Relative Visit Ratio=Number of Requests/load duration=183/30=6.1

Service Demand=6.1x2.38=14.51

Optimal number of Users=N=$\frac{Z+D}{Db}$= $\frac{42+14.51}{14.51}$=3.9

Our System has three stations, two systems have thinking times which cannot be our bottleneck. The remaining one is our bottleneck



5. Improvement of Scalability level

5.1. Scaling

For the improvement of scalability level there are several methods through which we can improve our scalability level e.g. scaling up and scaling down. We can improve our scalability level by adding either servers or databases horizontally or vertically as shown in Fig 5.1(a).

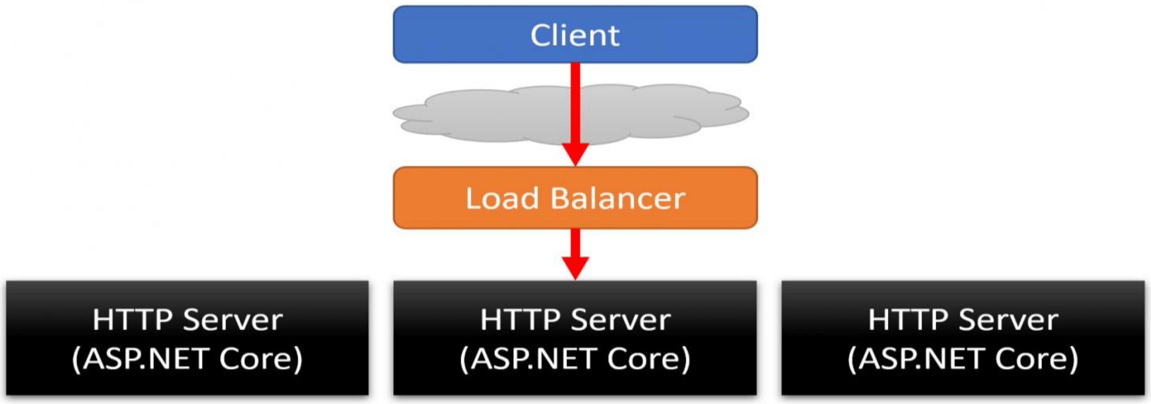

Client
Load Balancer
HTTP Server (ASP.NET Core)
HTTP Server (ASP.NET Core)
HTTP Server (ASP.NET Core)

Fig 5.1(a)

Similarly, to make the system even faster, we can use a load balancer, which then distributes the requests to the application equally. So, if we add load balancer then the load equally distributed. Moreover we can also, scale up the databases horizontally as shown in the Fig 5.1(b).

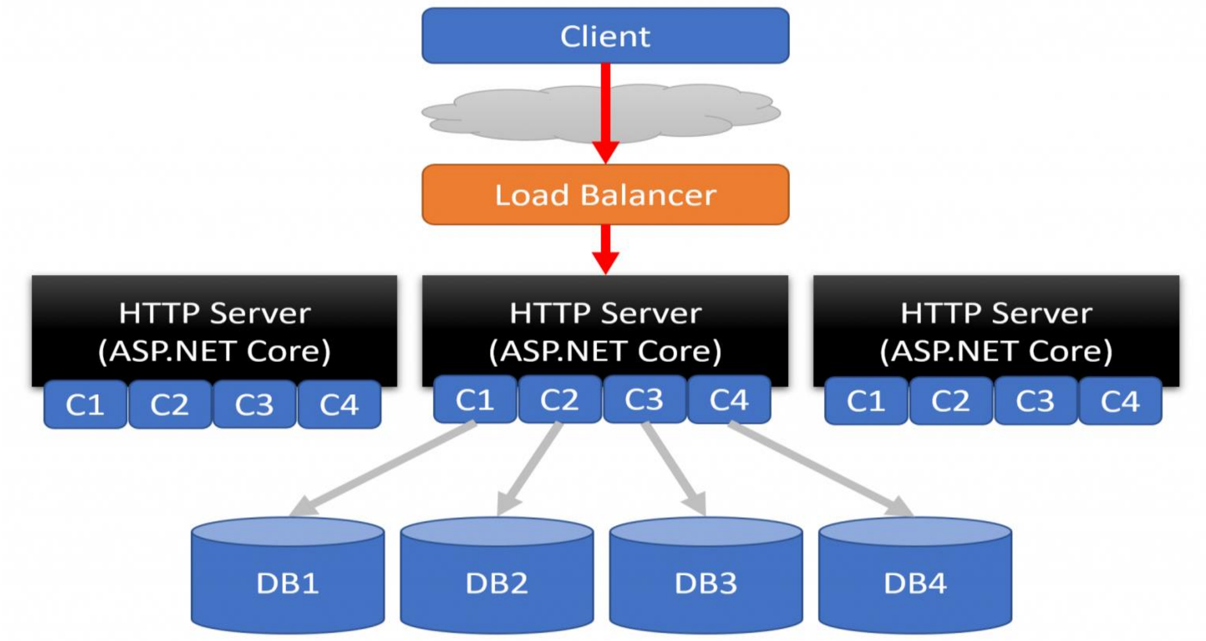

Client
Load Balancer
HTTP Server (ASP.NET Core)
C1
C2
C3
C4
HTTP Server (ASP.NET Core)
C1
C2
C3
C4
HTTP Server (ASP.NET Core)
C1
C2
C3
C4
DB1
DB2
DB3
DB4

Fig 5.1(b)

In the Fig 5.1(b), all the 3 servers are hosting all 4 boundaries (C1, C2, C3, and C4). However, we can create load balancing rules to create groups of servers that will only handle requests for a specific boundary. These are usually called Target Groups. After that we must increase the DB through which we can easily handle high load for example, in fig 5.1(b) that we have 4 DB through which we can handle huge load.

5.2. Caching

Caching is also another method for the improvement level of scalability. Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable our application to make vastly better use of the resources you already have, so that the data may be served faster during subsequent requests.

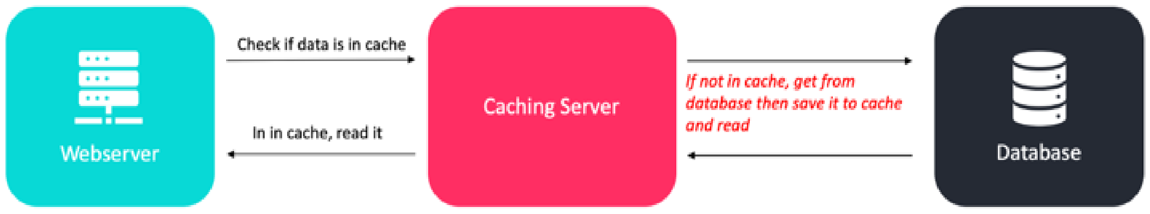

Webserver
Check if data is in cache
In in cache, read it
Caching Server
If not in cache, get from database then save it to cache and read
Database

Fig 5.2(a)

By making the use of caches with our servers, we can avoid reading the webpage or data directly from the server every time, in this way reducing the response time and the load on our server respectively. In this way we can make our application more scalable. Caching can be applied at many layers e.g. the database layer, web server layer and even network layer.

6. Conclusion

The application presented in this paper is built using Asp.net MVC 5 framework with web API and SQL database. The application is monolithic, all the data front end as well as the database is hosted at one place at a single server.

The web application presented in this paper was studied by creating a reasonable and accurate model, in order to analyze it with the help of theoretical and practical tools respectively, by making the use of Tsung and JMT. The benchmarking experiment was done using a PC as a client and our hosted app as remote server. With the help of operational analysis and JMT, it was possible to identify the performance of every type of query served by the application, and finding the bottleneck of the queuing network. In particular, we were able to find the bottleneck of the system, by assuming the closed interactive system.

Lastly, we studied different strategies to improve the scalability of the system notably, Scaling (Horizontally and vertically) and Caching.