Advanced Systems Lab (Fall'15) – Second Milestone

Name: *Yifan Su*
Legi number: *14-927-040*

Grading

Section	**Points**
1	
2	
3.1	
3.2	
4	
5	
Total	

1 System as One Unit

In this section, a M/M/1 queuing model of the whole system is presented based on the trace obtained during the stability experiment to build a coarse grained model of the real system. The model provides us with a rough simulation of the behavior of the system. The source code to build M/M/1 model can be found in *ethz.asl.statistics.MM1.java*. Recall that the stability experiment was performed on two middleware instances and one database instance using 32, 48 and 64 clients with a fixed sending rate of 100 reqs/sec. The raw data can be found in *logs/stability*. Two parameters (the mean arrival rate λ and the mean service rate μ) are needed to build the model.

To obtain the mean service time, the logs collected in middleware side during the stability experiment were used. Each middleware instance logged the time spent in the request/response queue/task or the database queue/task. The time spent in a queue does not count as service time which should be ignored when calculating average service time. Therefore, to get the mean service time of a single request, I added the time spent in the frontend tasks (both request and response) and database task together. Table 1 shows the detailed service time and waiting time for a single request with 48 clients.

Micro Part	Time (ms)
Service Time	1.4393
Waiting Time	0.4599
Total Time	1.8992

Table 1: Average time spent in micro parts of a single request processing

A service time of 1.4393 ms gives a service rate of $\mu = 1000/1.4393 = 695$ reqs/sec of a single backend worker. The real system in stability experiment contains two middleware instances with ten backend workers each. Hence the system can be modelled as 20 parallel M/M/1 queues with a service rate μ of each single bworker. Requests from clients are considered to be distributed equally among bworkers on both middleware instances. The utilization U, mean number of jobs $E(n)$ and calculated response time $E(w)$ of the system is as shown in eq. (1), eq. (2) and eq. (3).

$$U = \rho = \frac{\lambda}{m\mu} = \frac{\lambda}{\mu} \tag{1}$$

$$E(n) = \frac{\rho}{1-\rho} = \frac{\lambda}{\mu - \lambda} \tag{2}$$

$$E(w) = \frac{1/\mu}{1-\rho} = \frac{1}{\mu - \lambda} \tag{3}$$

Figure 1 shows the calculated response time of the model, the measured response time of the stability experiment as well as the maximum throughput experiment. As the figure illustrates, the coarse model of the system is quite similar to the real behavior of the real system especially at small arrival rate. After the arrival rate reaches 8000 reqs/sec, the model seems to overestimate the actual performance since the response time of the database increases much faster than the model assumes due to the bottleneck of the maximum throughput of the system. The measured values follow the same trend of the model and grow exponentially as the utilization of the system comes closer to 1, because as the arrival rate increases more requests have to be queued which in turn results in a higher queue time. Moreover, the measured response time of the stability experiment is similar to the value of the maximum throughput experiment at the same arrival rate.

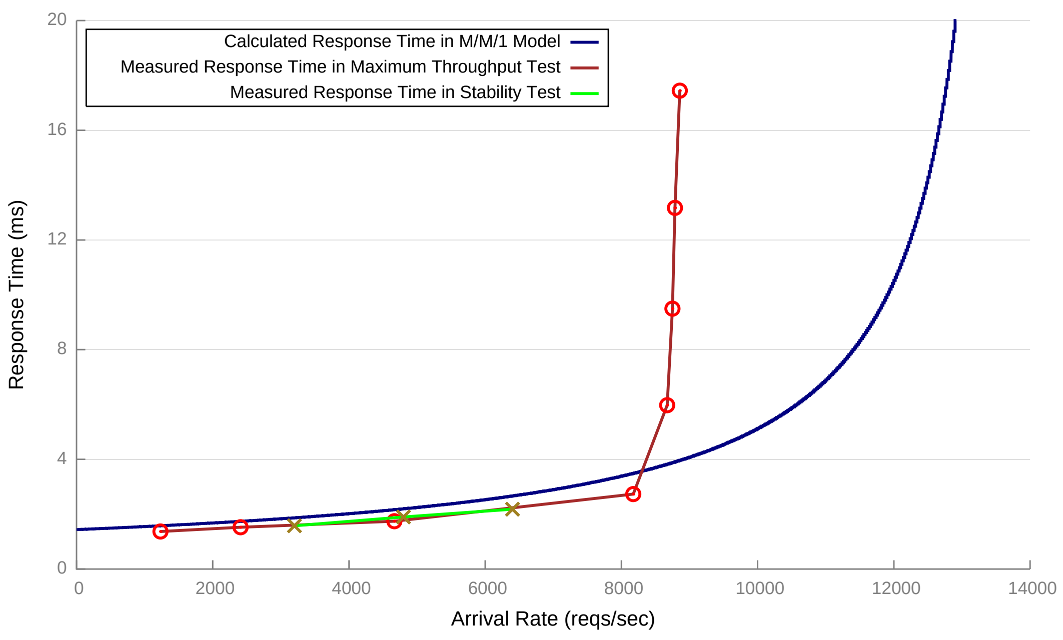

Figure 1: Response time of the M/M/1 model

2 Analysis of System Based on Scalability Data

This section applies M/M/1 queuing models to the scalability experiment with different configurations of the whole system and then compares the results of the models with the real system. The model uses a number of parallel M/M/1 queues regarding each available worker as seperate queues. As described in the first milestone, the goal of scalability experiments is to investigate how the throughput and response time of the whole system react when playing with different middleware resources (workers or nodes) under fixed client workload. For this experiment, the system was run with different numbers of middleware nodes and database workers with 72 fixed client connections. The number of backend workers per middleware varies from 1 to 16 with different numbers of middleware instance from 1 to 4. The log file can be found in *logs/scalability*. Here a coarse prediction of the response time from the model is expected.

Worker Threads	Config (instance×bworker)	Service Time (ms)	Service Rate (reqs/sec)
1	1×1	0.948	1055
2	1×2, 2×1	1.040	962
3	3×1	1.139	878
4	1×4, 2×2, 4×1	1.172	853
6	3×2	1.276	784
8	1×8, 2×4, 4×2	1.370	730
12	3×4	1.632	613
16	1×16, 2×8, 4×4	1.912	523
24	3×8	2.431	411
32	2×16, 4×8	2.515	398
48	3×16	3.125	320
64	4×16	3.048	328

Table 2: Service time and rate with different middleware configuration

Similarly, to establish such model, the service rate μ is required for each experiment. To obtain the mean service rate, I measured the service time of different configurations gathered by the same number of total worker threads. The raw data can be found in *logs/scalability/-plot_scale.xlsx*. Table 2 shows the number of worker threads in total for different configurations, the service time (ms), service rate (reqs/sec) in the scalability experiment. To plot the model, I only used configurations with the total number of worker threads 2^n (1, 2, 4, 8, 16, 32, 64) to make difference observable.

Figure 2 and fig. 3 show the response time and the utilization of M/M/1 models built with different configurations. The response time of the models is much higher than the measured time, which is probably due to the fact that the system now is modeled as a number of separate M/M/1 queues, while the real system has only one queue before the ExecutorPool for each middleware instance. That's why the complete system needs to be further adapted to M/M/m model. In general, M/M/m will be better because it leads to less waiting time where jobs waiting in a queue do not benefit if a server in another queue is free.

Basically, the models follow the general trend to the scalability experiments. For small arrival rates, the response time is the smallest for the model with the least worker threads since fewer backend workers lead to a higher service rate each. As the arrival rate increases, however, the model with the least workers has the worst response time behavior. The reason is that the model doesn't have enough available workers to handle all incoming requests with the increasing arrival rate, resulting in long waiting time in queues.

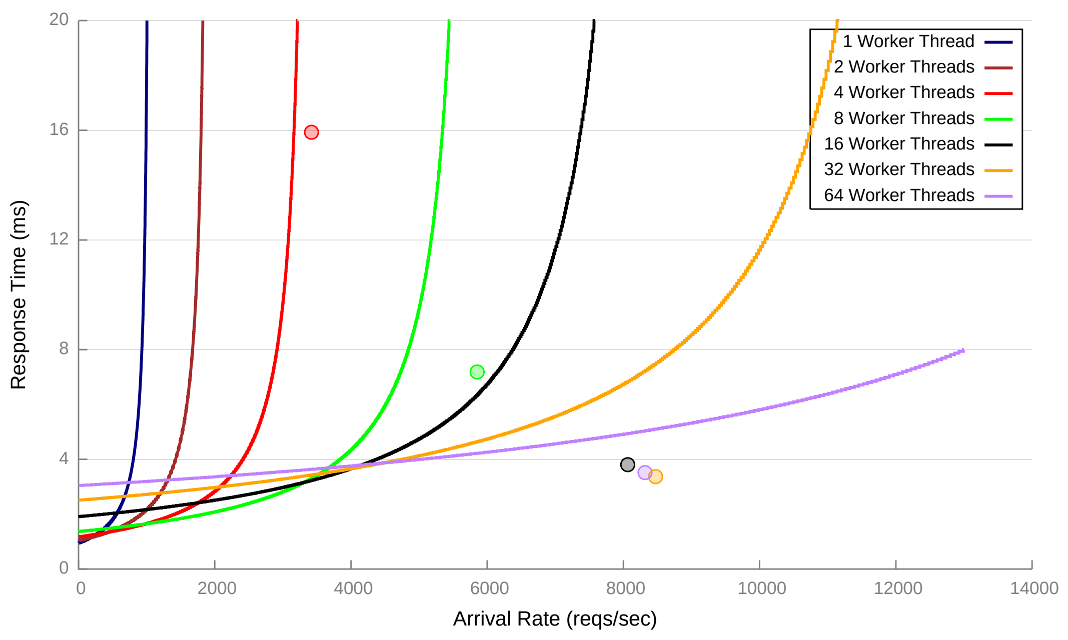

Figure 2: Response time of M/M/1 models

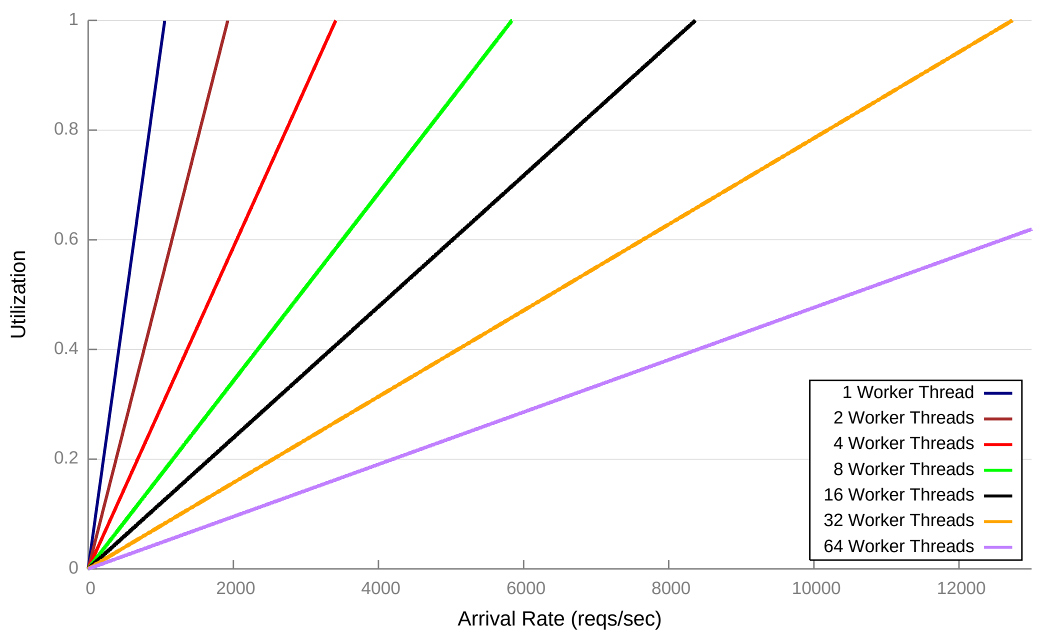

Figure 3: Utilization of M/M/1 models

3 Modeling Components as Independent Units

This section provides M/M/m queuing models of the database and middleware as independent units, then compares the models to the real system. M/M/m model can be used to model multi-processor services that keep all jobs waiting in a single queue and predict the mean response time, the utilization of the system and the queuing probability. Three parameters (the mean arrival rate λ, the mean service rate μ and the number of servers m) are required to build the model. For the following analysis, the formulas used are presented below. The utilization or traffic intensity ρ, probability of queuing ϱ and calculated response time $E(w)$ of the system is as shown in eq. (4), eq. (6) and eq. (7). The source code to establish M/M/m model can be found in *ethz.asl.statistics.MMM.java*.

$$U = TI = \rho = \frac{\lambda}{m\mu} \tag{4}$$

$$p_0 = \left[1 + \frac{(m\rho)^m}{m!\,(1-\rho)} + \sum_{n=1}^{m-1} \frac{(m\rho)^n}{n!}\right]^{-1} \tag{5}$$

$$P(\geq m\ jobs) = \varrho = \frac{(m\rho)^m}{m!\,(1-\rho)} p_0 \tag{6}$$

$$E(w) = \frac{1}{\mu}\left(1 + \frac{\varrho}{m\,(1-\rho)}\right) \tag{7}$$

3.1 Middleware

The three parameters λ, μ and m are required first before building a single M/M/m middleware model. To test the service time μ of a single middleware instance, I designed another simple benchmark that sends 8000 frontend (echo) requests to a *m3.large* EC2 instance[1] (client and middleware on the same machine).

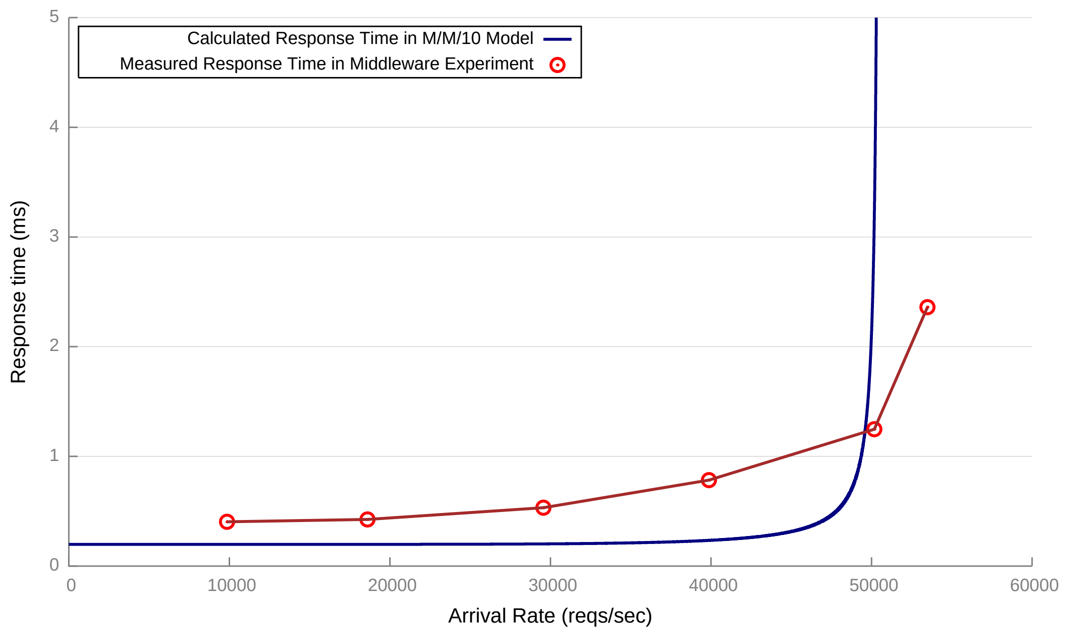

Figure 4: Response time of the M/M/10 model for the middleware

[1]See Apendix for more details

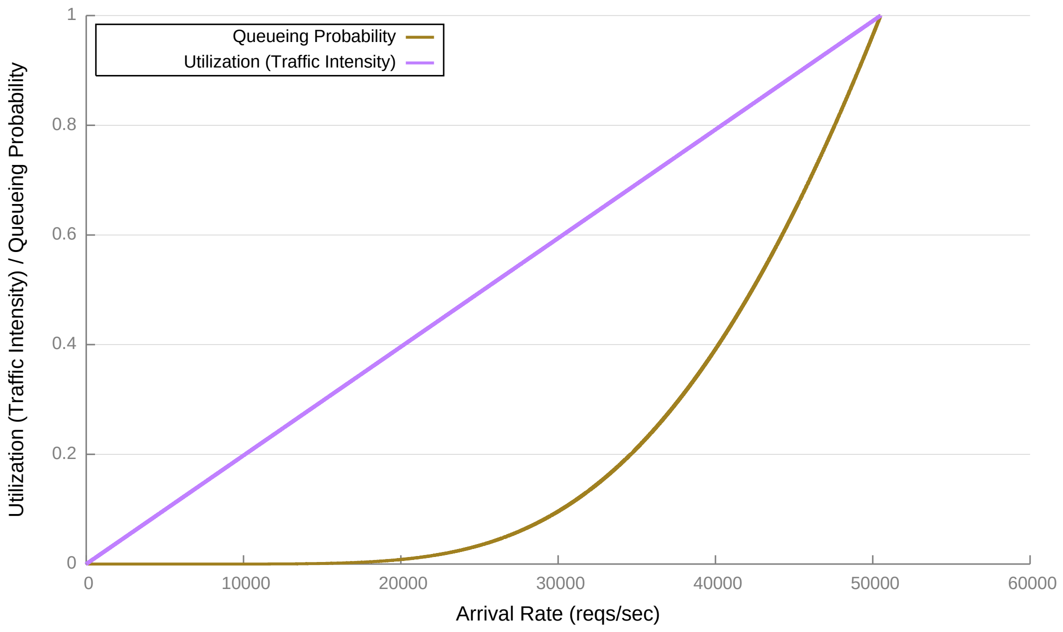

Figure 5: Utilization and queuing probability of the M/M/10 model for the middleware

The measured service time was 0.198 ms, which gives a service rate $\mu = 5050$ $reqs/sec$. The number of parallel services m is 10 corresponding to 10 parallel workers in the frontend ExecutorPool on a single machine.

Figure 4 compares the calculated response time with the mean time measured in the middleware performance experiments in the first milestone. Figure 5 shows the calculated utilization(traffic intensity) and queuing probability of the M/M/10 model. The model describes the behavior of a single middleware instance quite well, while the measured times are higher than the calculated times of the model when the arrival rate doesn't reach 50000. The reason is probably that the absolute value of response time is very small (below 1 ms) and the M/M/m model is just a rough approximation to simulate network I/O in a single thread.

Considering that, the results of the model can be used to estimate the maximum throughput while maintaining a reasonable response time. As it turns out, both the model and the experiment in the first milestone suggest that the maximum achievable throughput should be around 50000 reqs/sec with a response time of 1.2 ms. Increasing the load (the number of clients) after throughput reaching 50000 reqs/sec leads to a dramatic increase on the response time negatively.

3.2 Database

In this subsection, the three parameters λ, μ and m have to be obtained in a similar way to the middleware model. As discussed in the first milestone, the database was run on a *m3.xlarge* EC2 instance with 4 vCPUs[2] to handle 4 requests at the same time. By adjusting configuration parameters in PostgreSQL and carefully choosing indexes, the database can fit in the main memory easily so that disk accessing speed should not limit the performance for the real system greatly. Hence in the single database experiment, the number of servers m in the M/M/m model is set to 4 regardless of other influence factors like cache misses and I/O speed.

To test the service time of a single database instance, I designed another simple benchmark that sends 20000 requests (5000 of each type) one by one and measured the response time in sequence[3]. The mean response time for a single server is 0.492 ms with a standard deviation

[2]http://aws.amazon.com/ec2/instance-types/

[3]See Apendix for more details

of 0.180 ms. Since the simple experiment was run on a single machine, the service time of a single request is exactly the measured response time. The corresponding service rate $\mu = 1000/0.492 = 2033\ reqs/sec$. The total service rate for 4 servers is $m \cdot \mu = 8130\ reqs/sec$, which gives a good approximation of the measured maximum throughput in the database experiment in the first milestone.

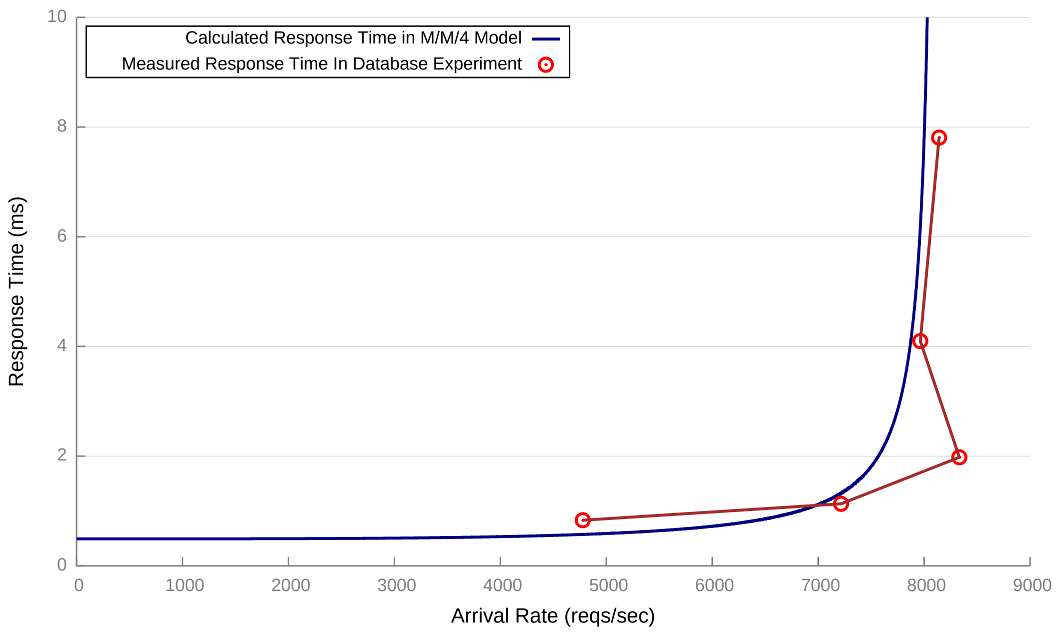

Figure 6: Response time of the M/M/4 model for the database

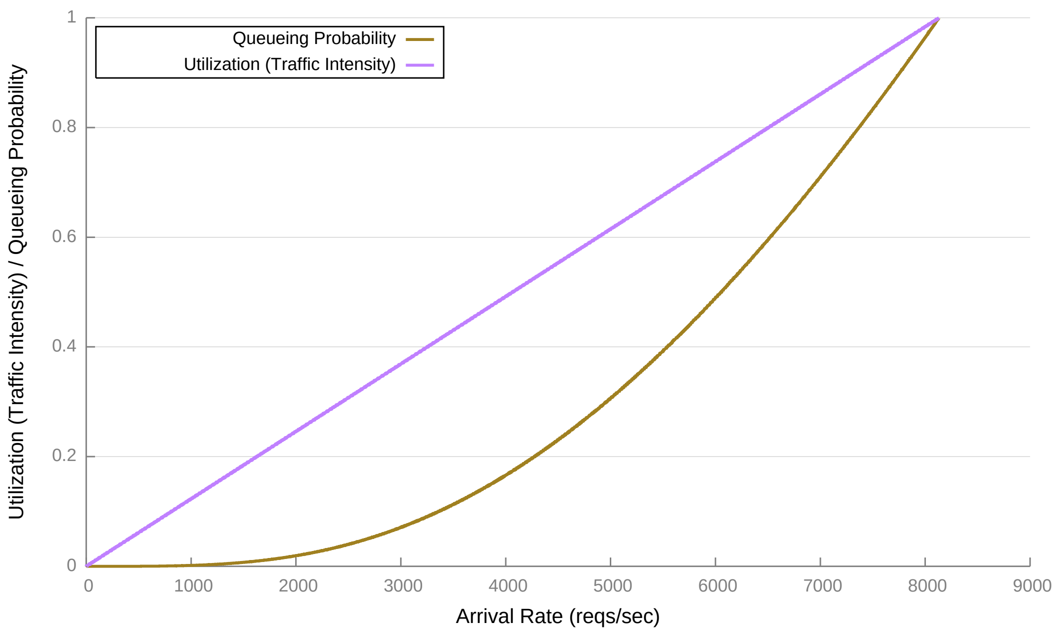

Figure 7: Utilization and queuing probability of the M/M/4 model for the database

Figure 6 compares the calculated response time with the mean response time measured of the mixed load in the database experiments. Figure 7 shows the calculated utilization(traffic intensity) and queuing probability of the M/M/4 model. The queuing probability remains reasonable and always below the line of utilization until the latter comes close to 1. Since the database experiments are done in a closed system with a thinking time, the situation that the utilization close to 1 should not occur in real.

As shown in the figure, the measured response time clearly follows the same trend of the model which suggests M/M/4 models the behavior of the database quite well. The real system can handle up to 8400 reqs/sec while the model can only handle about 7800 reqs/sec with reasonable response time. The reason is that the service time of the real database is somehow smaller than the service time of the model, resulting in higher service rate. Another interesting thing is that the increase in response time of the real system is not as fast as the model suggests. This is probably because not only the arrival rate but also some other essential factors, like the number of connections and I/O speed, not included in the model, affect the performance of the real system. Also the additional overhead of database connections leads to a drop in throughput because there will be more process locking/switching overhead as the number of connections increases to make full use of multiprocessor.

4 System as Network of Queues

All the analysis so far has only one queue. In real system, a job may receive service at one or more queues before exiting from the system. In this section, to fit the experiment setup with a various number of clients, a model based on a closed queuing network of the real system is presented as described in Chapter 32 of the text book. The model contains 4 components: network I/O queues, middleware frontend workers and backend workers, as well as database processors. Figure 8 shows the structure of the queuing network.

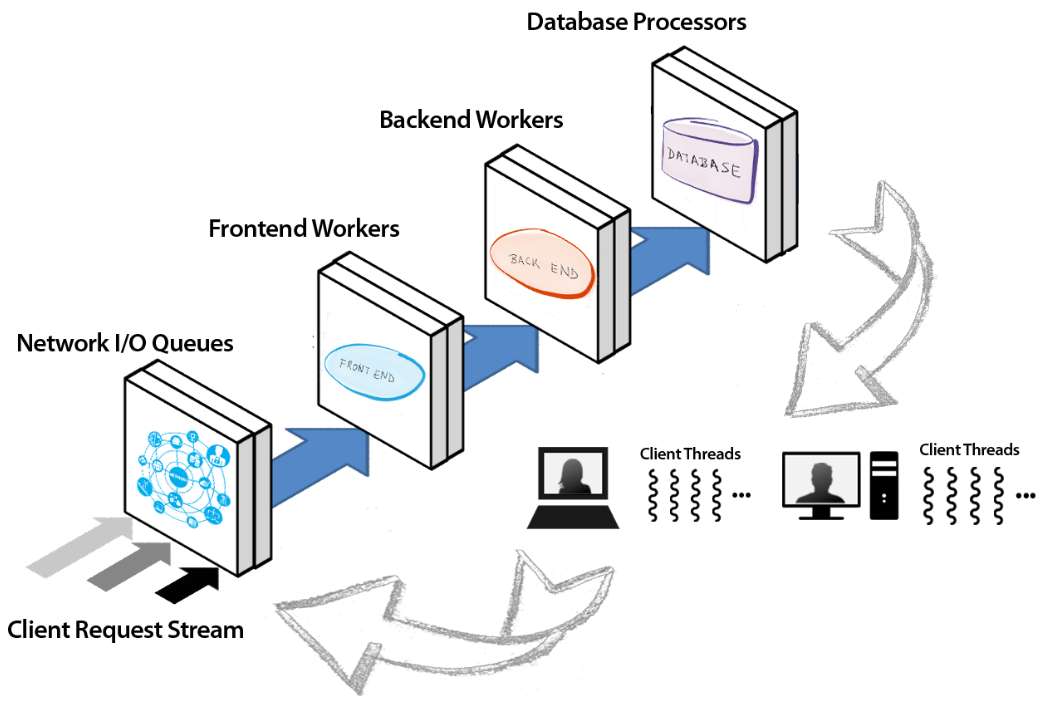

Figure 8: Structure of the closed queuing network

4.1 Comparison with Experiment Data

Table 3 shows the parameters for the closed queuing network of the system collected in the maximum throughput experiment running on two middleware instances (10 frontend workers and 10 backend workers each). The network I/O model has 20 queues based on the assumption that each frontend connection can send or receive data independently. The service time of a network queue is the time for sending a request and receiving the corresbounding response through the network. Each frontend worker and backend worker of the middleware are modeled as single queues. All service times can be obtained from the detailed middleware traces logged by middleware in the maximum throughput experiments *logs/maximum/server-max-**. The model of the database is similar to the discription in section 3. The complete queueing network is analyzed using the Mean-Value Analysis (MVA) described in Chapter 34 which solves closed queueing networks quite well. The algorithm of implementation can be found in *ethz.asl.statistics.MVA.java.*

The clients send requests with a fixed think time of 5 ms in the queuing network and the number of threads ranges from 8 to 400[4].

[4]Additional maximum throughput experiments with 240, 280, 320, 360, 400 clients were run. Details can be found in Appendix.

Parameter	Value
Network I/O queues	20
Network service time	0.198 ms
Frontend workers	20
Frontend service time	0.0117 ms
Backend workers	20
Backend service time	0.138 ms
Database processors	4
Database service time	0.492 ms
Number of clients/terminals N	Var
Think time Z	5 ms
Number of devices (exclude terminals) M	64

Table 3: Parameters for the closed queuing network of the system

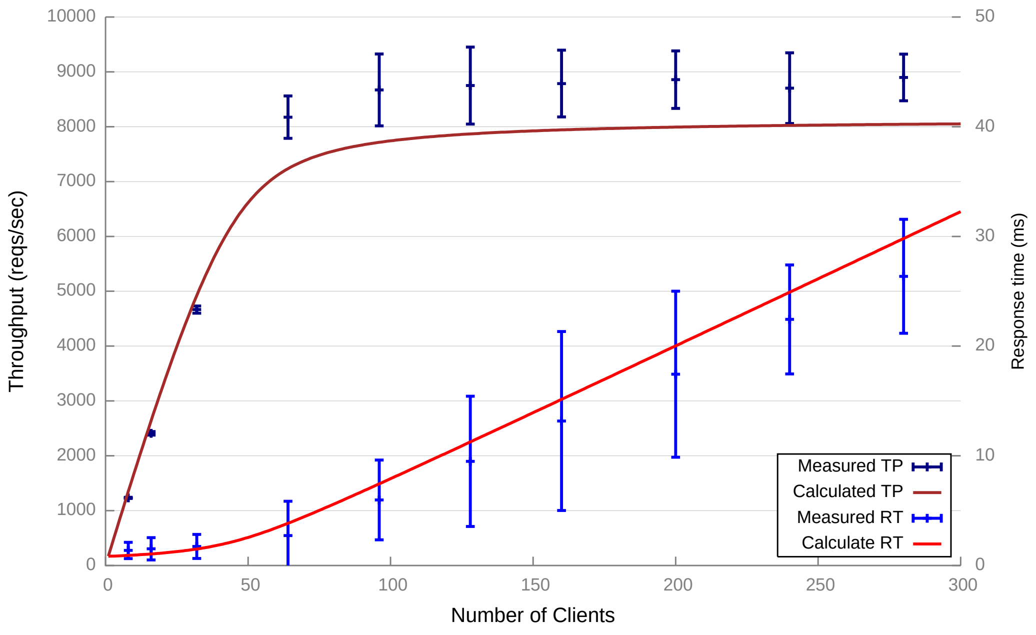

Figure 9: Comparison of measured and calculated throughput / response time

Figure 9 compares the measured throughput and response time in the maximum throughput experiments with the calculated values for a varying number of clients. The throughputs are calculated based on the interactive law. The behavior and general trend of the model is very similar to the real system. The queuing network models both the response time and throughput quite well. Only difference is that the measured values with larger number of clients are slightly lower than the calculated ones in the model. The reason is that every device here is modeled as a M/M/1 queue. In the real system, M/M/m model will be better because it leads to less waiting time (jobs waiting in a queue do not benefit if a server in another queue is free). Hence the real system is expected to achieve a smaller response time, in turn resulting in slightly higher throughput.

To conclude, the 4-component closed queuing network reflects the real system's behavior to efficiently analyze the performance of the system. The bottleneck of the system based on this model is then discussed in the following subsection.

4.2 Bottleneck Analysis

To find the bottleneck device, we need to find the device type with the highest average jobs and utilization rate. Both can be obtained from the MVA algorithm. Figure 10 and fig. 11 show the utilization and average number of jobs of different component types.

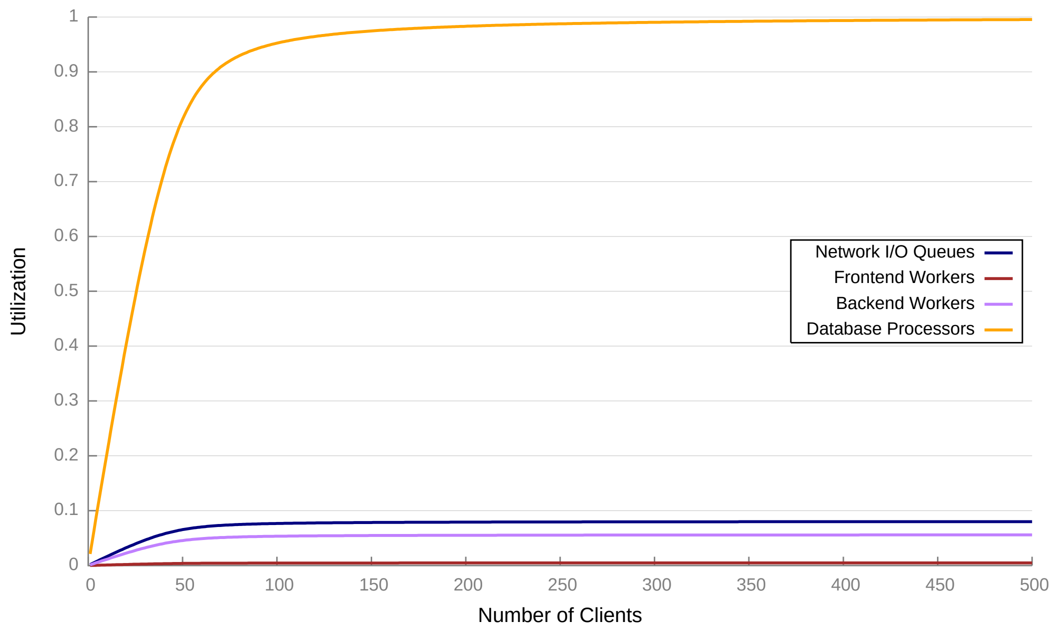

Figure 10: Utilization of the queuing network of the system

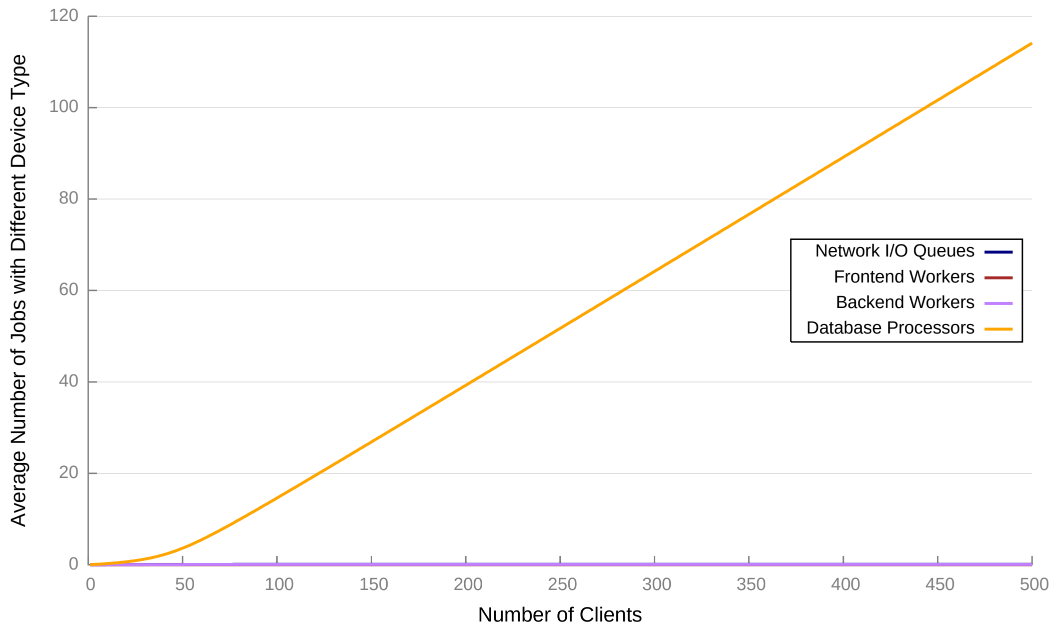

Figure 11: Average number of jobs with different device type

As both figures suggest, database is the bottleneck of the system, which draws the same conclusion as the analysis of the first milestone. The maximum utilization is reached very close to 1 with about 200 clients. Now let's focus more on the database components with a relatively high value in service time, utilization rate and average jobs. As the number of clients increases, more jobs are accumulated in the database components. First of all, there's only one single database instance to handle all the queries from two middleware instances and many client threads, resulting in a high processing overhead. Moreover, though using indexes keeps a high reading/querying performance, it's the database that performs the most time-comsuming part of the system like writing/modifying data to the hard drive. These factors greatly limit database's performance. As suggested in the conclusion part of the first milestone, it would be preferable if more database instances are added or the processing time of the database is speeded up, which will surely reduce the pressure of middleware instances and make full use of the latter.

The utilization rates of the three components of the system excluding database processors are all less than 0.1, proving the good design of the middleware instance. The non-blocking I/O network and compact format of messages allow the system to handle a large amount of requests from clients easily with a considerable transmission time, resulting in a low network overhead. Frontend workers only have to parse request bytes, create corresponding database task and package response back to the client. Backend workers generate the sql statement and create a frontend task to send back the response. Since all the tasks above are light-weighted and can be done efficiently due to the middleware structure, frontend and backend queues achieve a even lower utilization than of the network components. However, the low utilization of these three components reflects that the system didn't make full good use of each available one. Hence, it's reasonable for us to find ways to reduce redundancy.

To confirm the above supposition, then I predicted the behavior of the system with different choices of optimization as suggested above.

4.2.1 Database Optimization

Figure 12 and fig. 13 present the calculated response time and throughput of the queueing model with speeded up database processing time shown in table 4. Figure 14 takes a a closer look at the utilization of each component part.

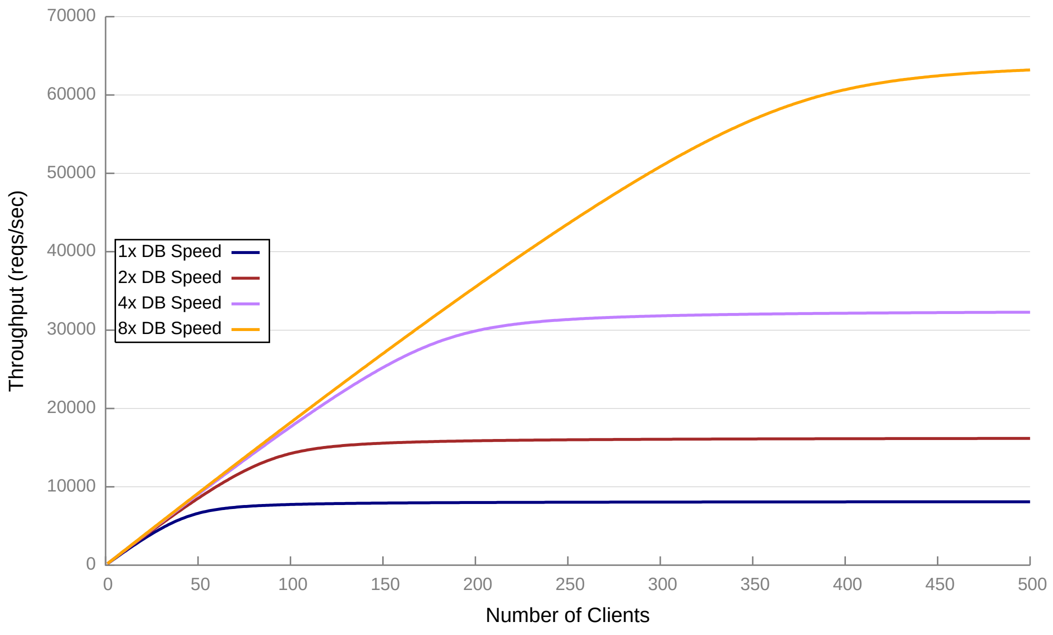

Figure 12: Response time of the model with speeded-up database instance

It shows clearly that the throughput increases and response time decreases proportionally to speed of the database since database processing accounts for the most important part of request handling. A speed-up in database processors makes it possible to get reasonable utilization of the other three components because there will be more client requests coming into the system,

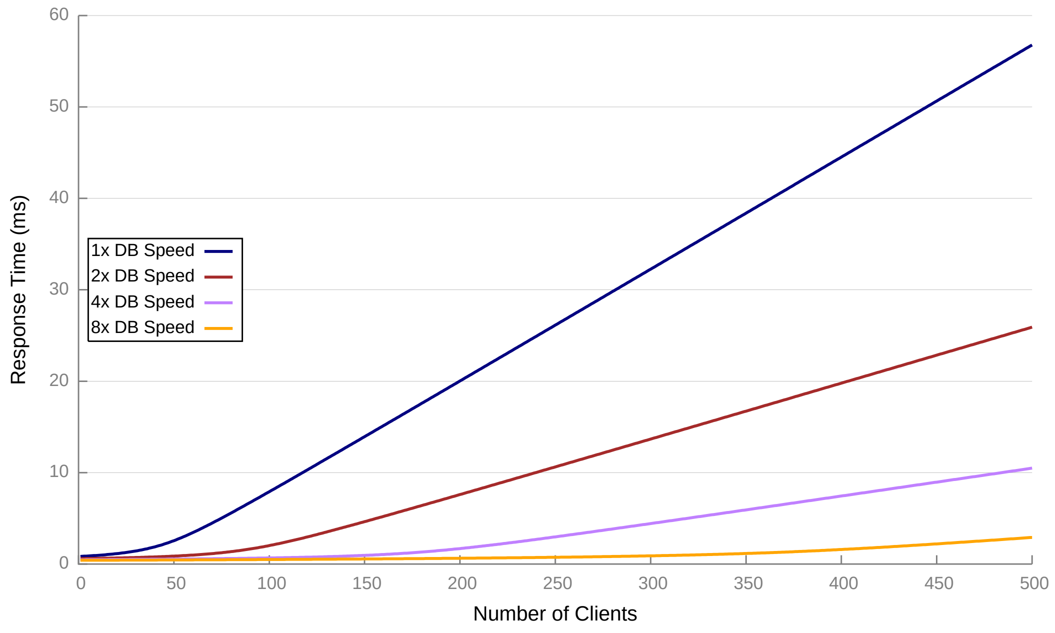

Figure 13: Throughput of the model with speeded-up database instance

Configuration	DB Service Time
1× Speed (Baseline)	0.492 *ms*
2× Speed	0.246 ms
4× Speed	0.123 ms
8× Speed	0.062 ms

Table 4: Speed-up database configuration

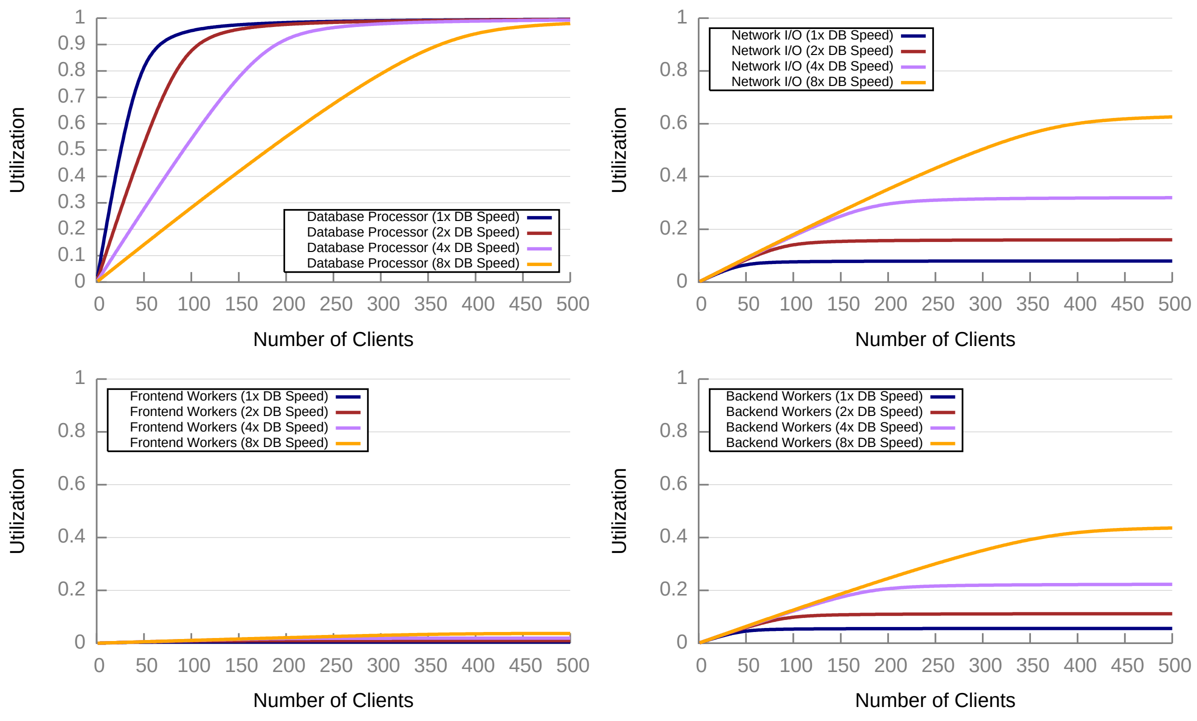

Figure 14: Utilization of the four components with speeded-up databases

0.6 for network I/O and 0.4 for backend workers at 8× DB speed. Hence increasing database's performance will surely lead to a higher use of other device types. Adding more database instances has the same positive effect as speeding up a single database. Figure 15 and fig. 16 show the calculated response time and throughput as the number of database instances increases.

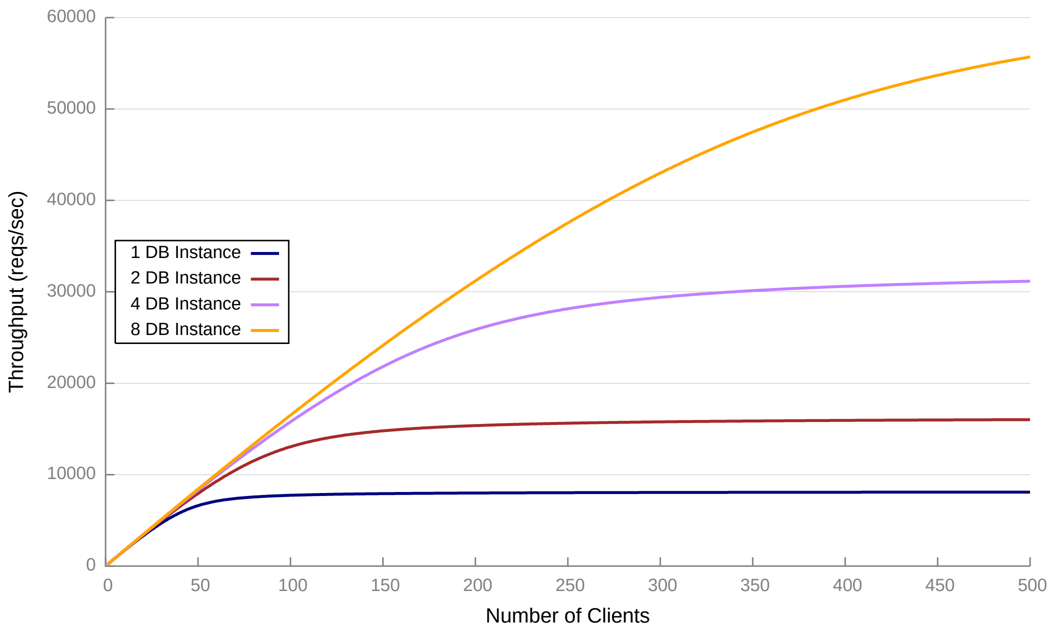

Figure 15: Response time of the model with more database instances

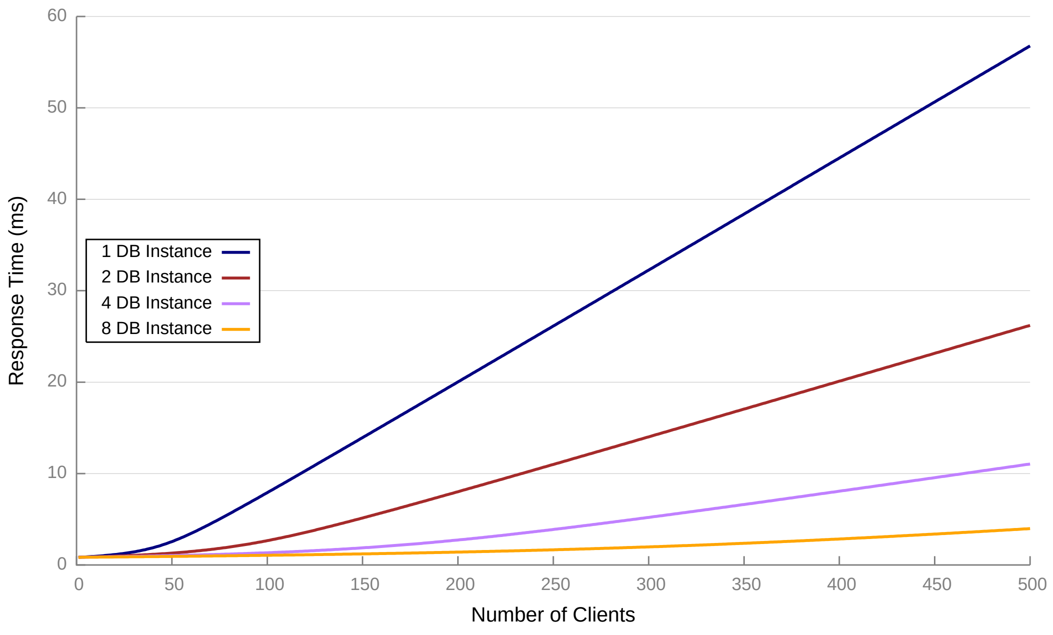

Figure 16: Throughput of the model with more database instances

4.2.2 Middleware Optimization

A good design of system not only requires to achieve optimum performance, but also to keep it compact and light-weighted. As shown in fig. 14, though there exists a reasonable increase in the utilization of backend workers and network I/O queues, the value of frontend workers is still very low. It seems that the only way to increase the utilization would be to reduce the number of front workers. Figure 17 and fig. 18 show the throughput, response time and utilization calculated by the model with different number of frontend workers.

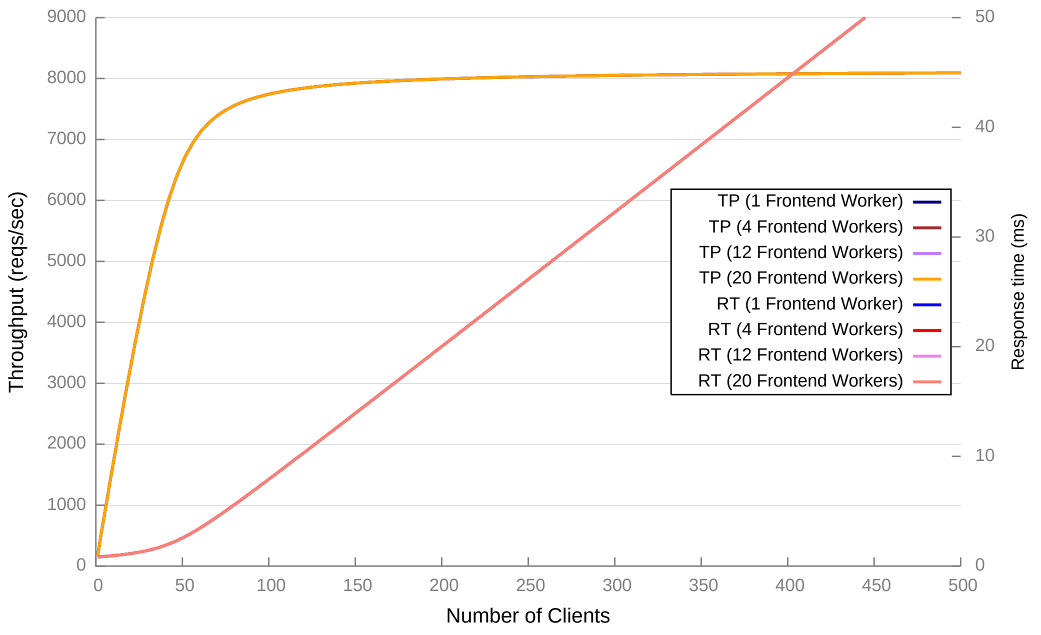

Figure 17: Throughput and response time with different number of fworkers

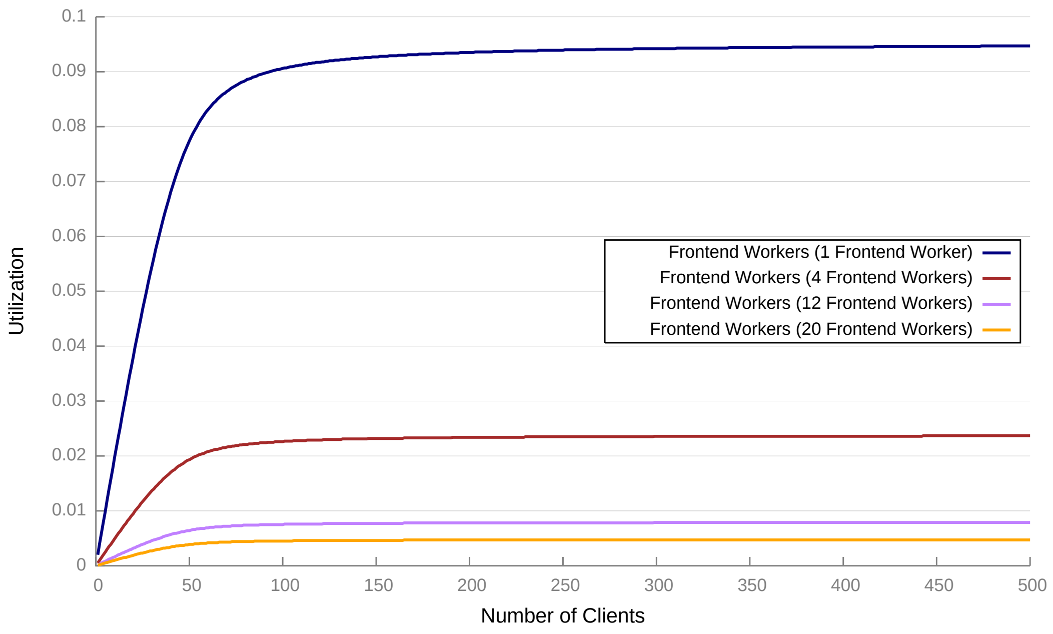

Figure 18: Utilization with different number of fworkers

The throughput and response time are almost the same as with different number of fworkers varies from 1 to 20 since frontend components accounts for little on the overall system performance. The maximum utilization of a single worker is just about 0.1, still far behind the value of a database processor. On this point, the number of fworkers could be drastically reduced. It's not necessary to maintain the frontend ExecutorPool with a number of separate frontend task queues.

5 Interactive Law Verification

In this section, I applied the interactive law to the measured throughput and response time.

$$X = \frac{Jobs}{Time} = \frac{N\frac{T}{R+Z}}{T} = \frac{N}{R+Z} \tag{8}$$

In the above equation, X is the throughput, N is the number of users, R is the response time and $R+Z$ is the total cycle time. It's assumed that the measured average throughput and the calculated one should match.

Similarly to the throughput, the response time can be computed using another interactive law as below.

$$R = (N/X) - Z \tag{9}$$

Table 5 shows the measured and calculated values for database benchmark experiment (mixed workload) without any thinking time (Z=0). There's only a small diviation between the measured and the calculated ones, which can be explained by the fact that the measured response time and throughput for the calculation and comparison are averaged over a large set of measurements. Therefore it can be concluded that the experiment data is valid according to the law.

Then I applied the same analysis to the maximum throughput experiments (mixed workload) and 2^k experiments with a thinking time of five ms. Table 6 and table 7 show the results for the comparison between the measured and calculated throughput and response time. The results are quite similar to the database analysis with interactive law. The highest difference(%) is about 8 % for response time with 8 client threads in maximum throughput experiment and 7 % for response time with 24 clients in 2^k experiment, which is due to the fact that the absolute value of the response time for a small workload is very low. The small absolute values can also explain why the difference(%) of all response time is higher than the corresponding throughput.

N	R_m(ms)	R_{cal}(ms)	X_m(Reqs/sec)	X_{cal}(Reqs/sec)	$Difference_R$	%	$Difference_X$	%
4	0.83	0.84	4778	4809.43	0.005	0.65	31.43	0.65
8	1.13	1.11	7216.42	7071.51	0.023	2.05	144.91	2.05
16	1.98	1.92	8333.17	8088.57	0.058	3.02	244.60	3.02
32	4.10	4.02	7963.83	7812.31	0.078	1.94	151.53	1.94
64	7.81	7.86	8141.41	8197.67	0.054	0.69	56.26	0.69

Table 5: Measured and calculated throughput and response time for database experiments

N	R_m(ms)	R_{cal}(ms)	X_m(Reqs/sec)	X_{cal}(Reqs/sec)	$Difference_R$	%	$Difference_X$	%
8	1.37	1.49	1232.76	1255.59	0.12	7.92	22.83	1.82
16	1.53	1.64	2408.99	2451.32	0.11	6.99	42.33	1.73
32	1.74	1.86	4666.45	4746.86	0.12	6.25	80.41	1.69
64	2.73	2.83	8174.19	8277.72	0.10	3.46	103.52	1.25
96	5.98	6.07	8670.85	8745.80	0.09	1.56	74.95	0.86
128	9.49	9.63	8749.50	8830.81	0.13	1.40	81.31	0.92
160	13.17	13.21	8786.66	8805.82	0.04	0.30	19.16	0.22
200	17.44	17.58	8857.89	8912.30	0.14	0.78	54.41	0.61

Table 6: Measured and calculated throughput and response time for maximum throughput experiments

N	R_m(ms)	R_{cal}(ms)	X_m(Reqs/sec)	X_{cal}(Reqs/sec)	$Difference_R$	%	$Difference_X$	%
96	9.61	9.71	6525.19	6572.37	0.11	1.09	47.18	0.72
96	10.17	10.28	6281.62	6326.57	0.11	1.06	44.95	0.71
24	1.70	1.82	3520.43	3584.34	0.12	6.69	63.91	1.78
24	1.93	2.04	3408.58	3461.31	0.11	5.25	52.72	1.52
96	6.09	6.25	8536.66	8653.72	0.15	2.44	117.06	1.35
96	6.71	6.88	8078.09	8196.02	0.17	2.48	117.93	1.44
24	1.79	1.92	3469.70	3532.79	0.12	6.44	63.09	1.79
24	1.76	1.88	3486.27	3547.78	0.12	6.33	61.50	1.73

Table 7: Measured and calculated throughput and response time for 2^k experiments

Appendix: Repeated Experiments

Single Database Experiment

To test the service time of a single database instance, I designed a simple benchmark that sends 20000 requests (5000 of each type) one by one and logged the response time in sequence. The mean response time for a single server is 0.492 ms with a standard deviation of 0.180 ms. The code can be found in *ethz.asl.benchmark.simple*. The logs are included in *logs/simple*.

Single Middleware Experiment

To test the service time of a single middleware instance, I designed another simple benchmark that sends 8000 simple frontend requests (without database access) to a *m3.large* EC2 instance. The measured service time was 0.198 ms with a standard deviation of 0.236 ms. The code can be found in *ethz.asl.benchmark.simple*. The logs are included in *logs/simple*.

Additional Experiment on Maximum Throughput Benchmarks

To verify the queueing model of the system with large number of clients, additional maximum throughput experiments with 240, 280, 320, 360, 400 clients were run. The setup was similar to the first milestone. The log files can be found in *logs/maximum_update*.