Advanced Systems Lab (Fall'15) – Second Milestone

Name: *Pascal Widmer*

Legi number: *10-924-439*

Grading

Section	Points
1	
2	
3.1	
3.2	
4	
5	
Total	

1. System as One Unit

To use an M/M/1 model for our system a lot of simplifications have to be made. The model requires a single-core system (a single server) and a single queue with infinite size and FCFS service discipline. Of course, the implemented system uses parallel threads which are only independent to a certain degree. In the M/M/1 model the inter-arrival times follow an exponential distribution which requires individual requests to be independent of previous requests. The implemented system uses synchronous message passing between clients and middleware and a client never sends a request before it has received a reply. It follows that clients do not need a think-time to keep the middleware's request queue from overflowing and the response time measured by clients represents a lower bound on how much waiting can be done in the client code. This also conveniently ensures job flow balance which is needed for the model to work. The request queue implements the FCFS discipline but requests arriving from the network can nevertheless be reordered by the middleware before arriving there.

We can approximately represent the system using a M/M/1 model by first attaining the mean service rate μ. In the stability experiment done in the previous milestone no think-time was implemented. The service rate corresponded to the systems throughput because there were always jobs ready for service. For this milestone the stability test is rerun using a think time of 30ms per client. (See appendix). The service time can be calculated using the response time measured in the database threads plus the time spent in the foreground threads which interact with clients. The time spent in the request queue, i.e., the time spent waiting for a database thread to work on the request, is counted as waiting time.

The following values correspond to the average time spent in each part of the system. See the stability test log files for 95% confidence intervals:

- Database Thread: 4.597 ms
- Request queue: (4.797 – 4.597) ms = 0.200 ms
- Time spent between client and middleware:
  (4.980 – 4.797) ms = 0 .183 ms

The total service time is thus approximately 4.780 ms while the waiting time is 0.2 ms. The service time corresponds to a service rate of μ = 1/(4.780 ms/request) ~= 209 requests/second. Since the system is run using 8 parallel database threads (each middleware 4) a more realistic service rate approximation is given by 8* 209 requests/second = 1672 requests/second. However, it does not correspond to a M/M/1 model since the service rate would need to be 1/1672 seconds/request = 0.598 ms which is far off the measured 4.780 ms. It would be more accurate to model the system as 8 parallel M/M/1 models instead. But using 8 parallel M/M/1 models requires each DB thread to have a separate request queue, which is not the case either. The models do not account for the fact that work done between client and middleware, i.e., work spent on the wire and in networking threads, is actually executed in parallel to work done in the database threads. While the database threads wait for a response from the database the client requests can be handled by foreground threads. In a M/M/1 model using a single server this would not be possible and so the real service rate is higher. Neither the middleware threads nor the database operations are completely independent and as such 8 parallel M/M/1 is only accurate for a low database thread/connection count. A M/M/8 model which models a system with only one request queue would be more accurate in cases where the arrival rate is significantly lower than the service rate, but is not required here.

Figure 1 shows average response times of the 8 parallel M/M/1 queues, single M/M/1 queue and M/M/8 queue. The circle signifies measurements of the stability experiment.

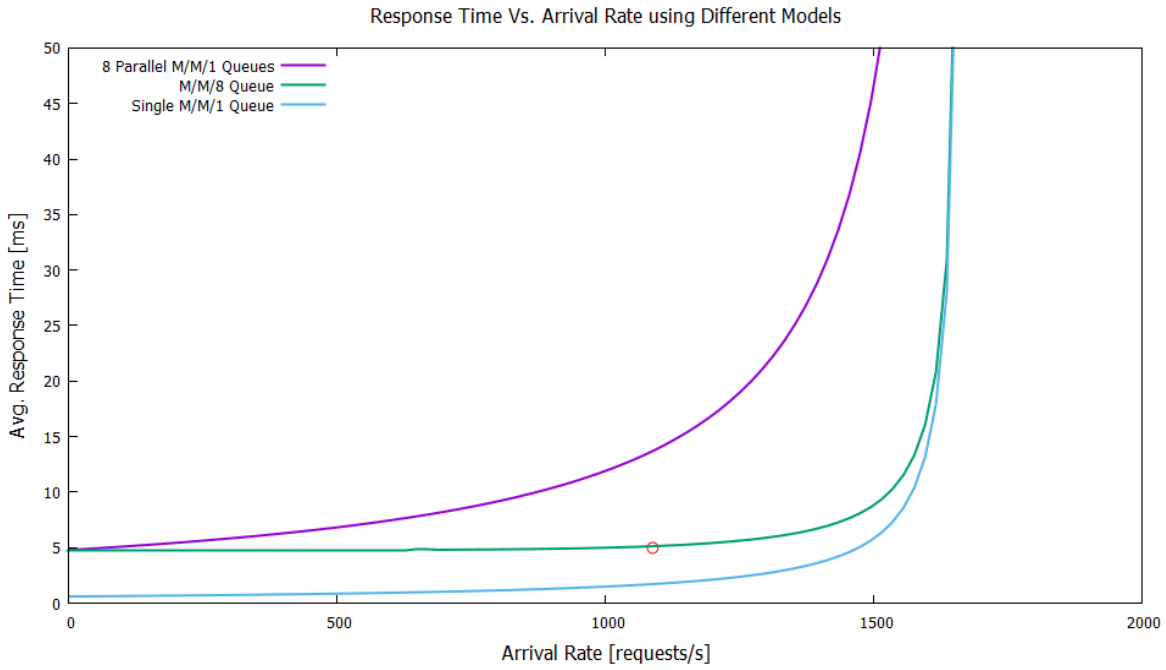

Figure 1

Not surprisingly, data from the stability experiment fits neither model. The real system is run on 2 middleware with 4 database threads each, which matches 2 parallel M/M/4 models the closest. Since clients wait for replies and 32 clients are used for the experiment the maximum number of requests waiting for service can never exceed 32 in any queue. Therefore, and contrary to the models, the average measured response time has an upper bound when λ/μ tends to 1. *Figure 2* shows the average waiting time and similar observations could be made. In this section the models had to be based on the stability data. In other sections we will see that scaling the number of clients makes more sense in regard to the analysis (and a potential deployment).

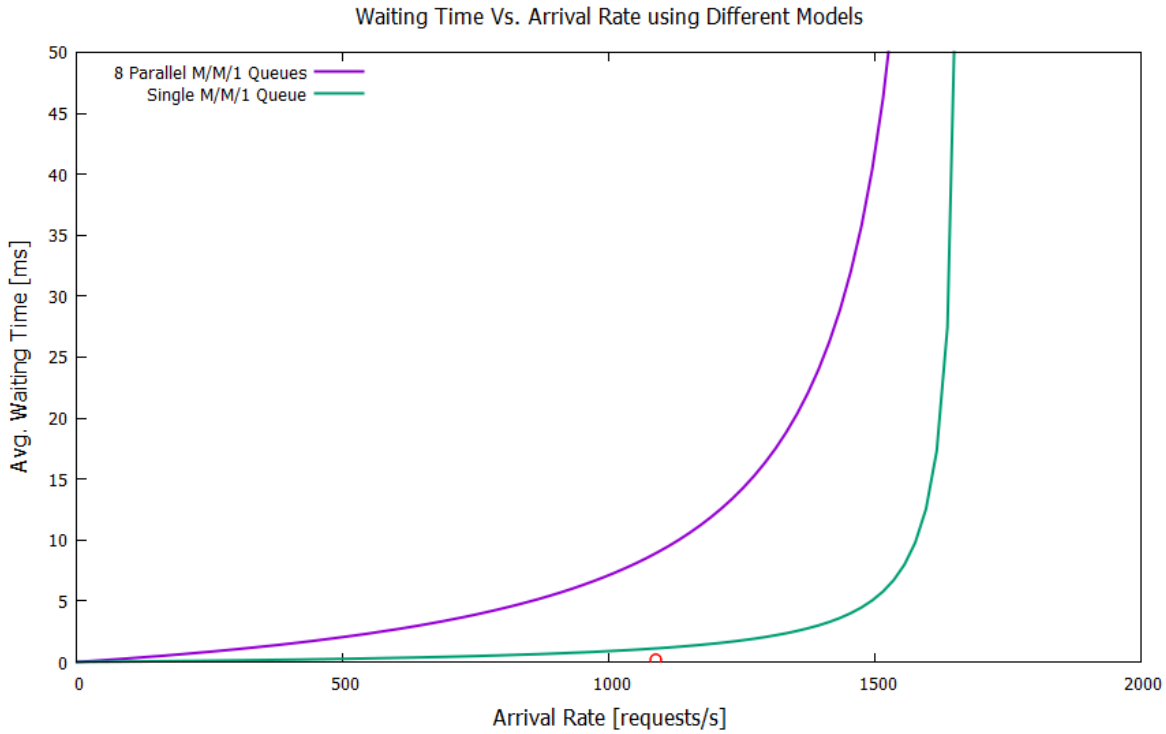

Figure 2

2. Analysis of System Based on Scalability Data

In the scalability experiments done in the previous milestone I mainly focused on the number of clients connected to the system and how it affects performance. The scalability experiments done here (See appendix) focus on a different aspect of scalability. Namely, how the system's performance changes in regard to how many middleware are run and the number of database connections each middleware uses. The following four different configurations are explored and compared:

- 1 Middleware with 4 database connections
- 1 Middleware with 8 database connections
- 2 Middleware with 4 database connections each
- 2 Middleware with 8 database connections each

Note that clients still wait for replies from the database before sending new requests. The experiment is run with 32 total clients. At least 32 clients are needed to achieve the maximum throughput (i.e., database connections always find waiting requests). The system can be represented as m parallel M/M/1 queues where m is equal to the total number of database connections. In the following the most important measurements are summarized. Service time is the time spent between client and middleware plus time spent in the database thread excluding the time spent in the queue.

Configuration	DB Connections/ m	Service Time [ms]	Service Rate DB Connection [req/s]	Total Service Rate [req/s]	Measured Throughput [req/s]
1MW/4DB	4	0.199+3.268 = 3.467	288	1154	1232
1MW/8DB	8	0.224+5.341 = 5.565	180	1438	1505
2MW/4DB	8	0.2172 + 5.286 = 5.503	182	1454	1527
2MW/8DB	16	0.230+ 10.424 = 10.654	94	1502	1552

Again, most of the discrepancy between measured throughput and calculated service rate can be explained by the parallelism of database threads and client handling threads on the middleware. The possibility of parallelism in the middleware is in turn a result of the database running on a remote instance. The table also shows that more than 4 database connections are needed to saturate the database. This is expected since the database runs on a 4 vCPU instance and the network introduces some delay. Whether 1 middleware with 8 DB connections or 2 middleware with 4 DB connections each are used has little effect on the performance. Using a think-time in clients a specific arrival rate can be achieved and the response time measurements follow approximately the m parallel M/M/1 model seen in Figure 3.

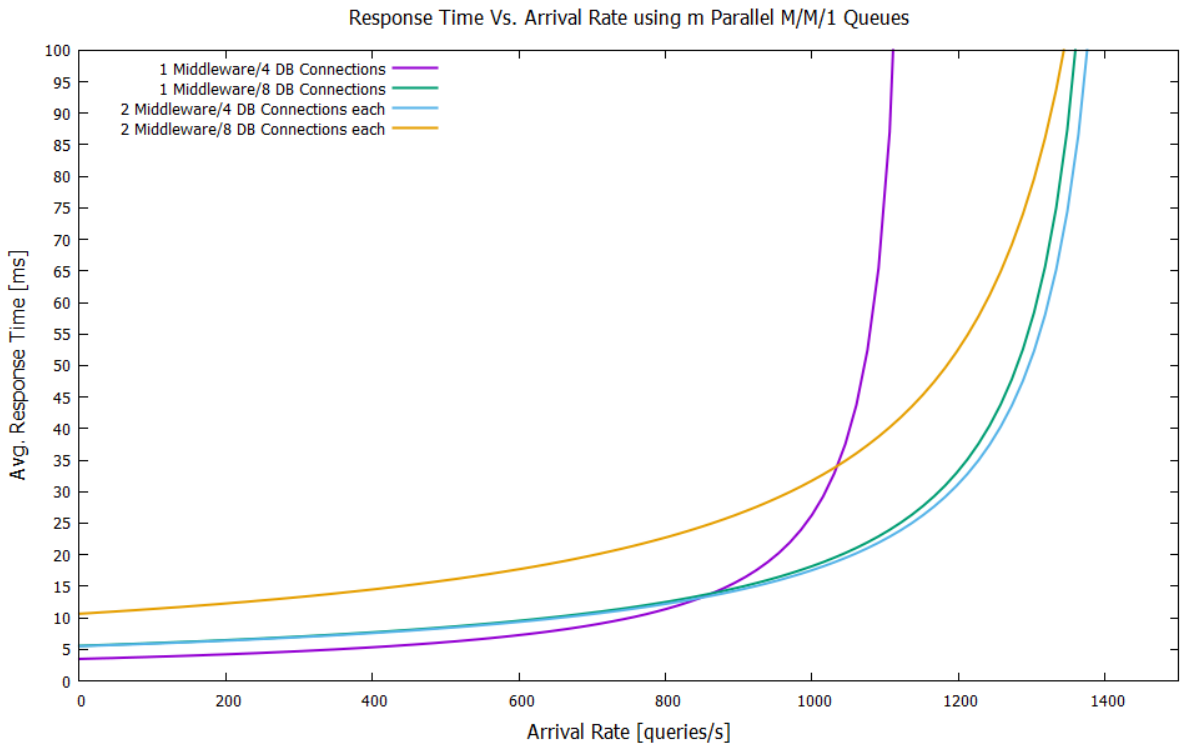

Figure 3

The response times predicted above only approximate the real response times since the system uses 1 request queue per middleware and not per database thread. When the utilization is low the real response time is also lower. In the m parallel M/M/1 model when a request arrives there is always a chance that the request reaches a queue which can not immediately be served while a queue of another database thread is idle. In a single M/M/1 or a M/M/m model this does not happen. Configurations with more database connections tend to have higher response times for lower arrival rates. This can be attributed to the fact that the model assumes a fixed service rate for a single queue. The service rate however increases for a single connection when all others are idle since there is less resource contention. The maximum throughput of each configuration corresponds to the arrival rate at the asymptote of the corresponding function. The configuration using only 4 database connections does not achieve the same throughput as 8 or more connections since 4 network connections cannot fully saturate the 4 cores of the database server.

In the following the number of clients are scaled up and a think time is used. This way response time measurements done in the client can be related to the predicted response times of the models. In the configuration used here a single middleware is run using 8 database connections. Blue circles represent measurements done with 200 clients and red circles with 100 clients respectively.

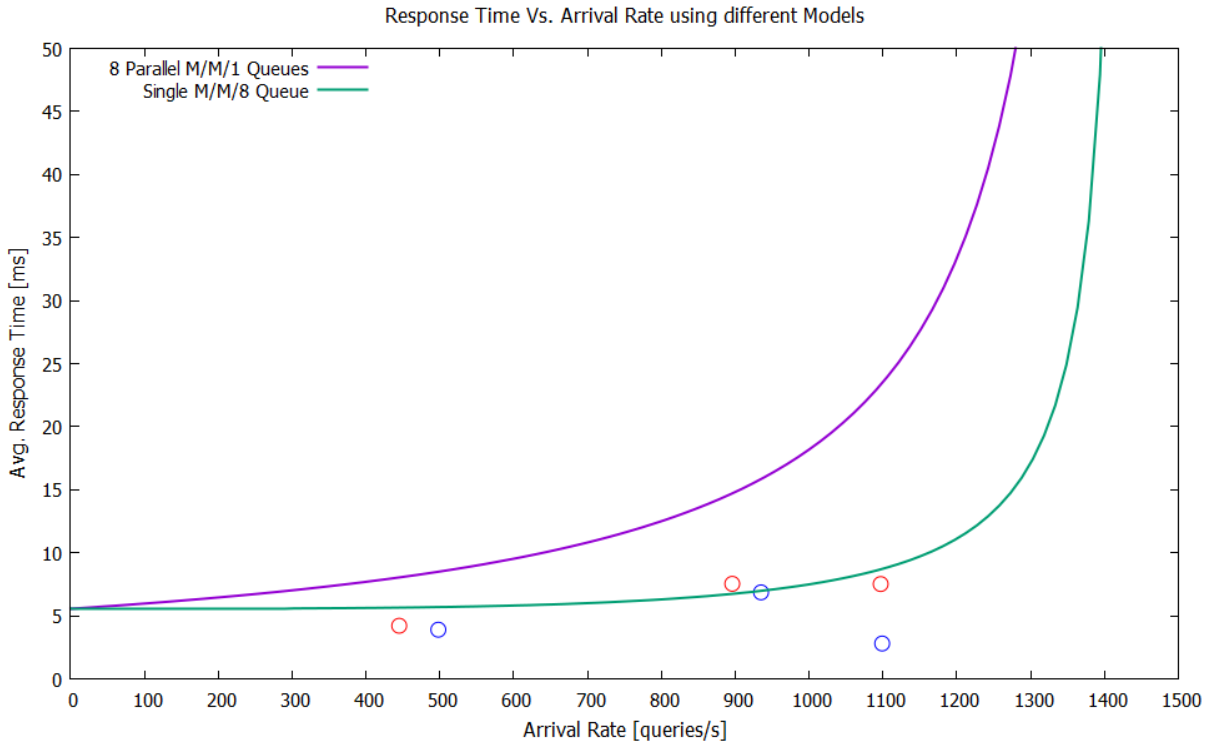

Figure 4 Red: 100 Clients, Blue: 200 Clients

We can clearly see that a M/M/8 model is a better fit for a single middleware instance/a single request queue and the measurements roughly fit the predictions. However, especially for large number of clients in combination with a high arrival rate (low think-time) the measurements are very imprecise. Because clients are implemented as single threads and run on a 2 vCPU instance we expect some imprecision. Some of the measurements are however too low by a large margin and are caused extensive use of timing methods and sleep calls.

3. Modeling Components as Independent Units

3.1. Middleware

Each client connecting to the middleware receives a dedicated communication thread which interacts with the client. There is no implemented limit on how many clients can potentially connect and no fixed- sized connection pool is used for clients. It makes intuitively sense to set m, i.e., the number of parallel servers, to the number of clients/client handling threads in the middleware and adapt the service rate of the servers correspondingly. The network device could be seen as a queue/buffer which stores requests and the client handling threads represent the servers accessing the buffer. However, this makes the model dependent on the number of clients which might not be ideal. In addition, the requests from clients can only be handled by one specific thread. Since clients never send two requests in sequence without first getting a reply the work-distributing factor of the M/M/m becomes meaningless and all threads are almost always idle. Whether threads are idle or busy also affects the service rates of other threads and the real service rate per server would in reality be a lot higher than a M/M/m model predicts. Another approach is taken here to find a meaningful number of servers and service rate.

To model the middleware as a M/M/m queue the service time of a single request is first measured. A single client sends individual Echo messages with a specific payload size to the middleware and measures response times. A 200-character payload is used to simulate Send/Pop/Peek messages. This corresponds to at least a 200-character message of either Send/Pop/Peek because their payload is either sent only one-way or no message is carried at all. Echo Requests however are always answered with Echo responses with the same content. The experiments are run on a single 2 vCPU Amazon instance running both client and middleware and on two separate 2 vCPU instances. The experiment with separated experiment allows for an accurate network delay measurement. In the middleware there are always 4 or more database threads running which simply parse and return the request retrieved from the request queue. Some quick tests determined that a single database thread can handle 24000 requests, thus running only one database thread does not limit the throughput for a single client. To achieve the maximum throughput and mirror configurations where the database is active 4 database threads are needed.

The following has been measured (95% confidence interval in log files):

Message Type	Remote Response Time [ms]	Local Response Time [ms]	Network RTT [ms]	Service Time [ms]	Service Rate [req/s]
Echo (200c)	0.245	0.082	0.163	0.021	12195

Note that here service time excludes time spent for reading from the network buffer and so the service rate is more accurately 1/Local Response Time. We can proceed to find the maximum throughput attainable where messages still have a similar response time. This can be seen as finding the arrival rate corresponding to the asymptote in the M/M/m model's response time graph. Using the measured throughput, it is possible to make an estimate on how many servers can be running in parallel without much interference. Adding more clients or more load to the middleware would now increase the response time and the requests would need to be queued.

Number of Clients /Client Threads in Middleware	Local Measured Response Time [ms]	Local Measured Throughput [req/s]	Remote Measured Throughput [req/s]
1	0.080	12369	4007
2	0.090	20440	7995
4	0.144	21238	15239
8	0.208	21915	23813
16	0.216	21744	43236
32	N/A	N/A	49762

We can clearly see that running 2 clients locally suffices to achieve maximum throughput. Since network IO contributes the most service time and clients and middleware both do the same amount of IO we can expect that relocating the clients to a remote machine would yield at least twice as much maximum throughput. Additionally, two threads in the middleware would no longer be enough to handle the load since one of them can only handle ~12500 requests/second. However, 4 clients are not enough to generate the maximum load as a result of network delay, we need more clients to achieve more throughput. 32 clients proved to be enough. A M/M/4 model with a service rate of 4*12500 is best used to represent the middleware as an independent part of the system.

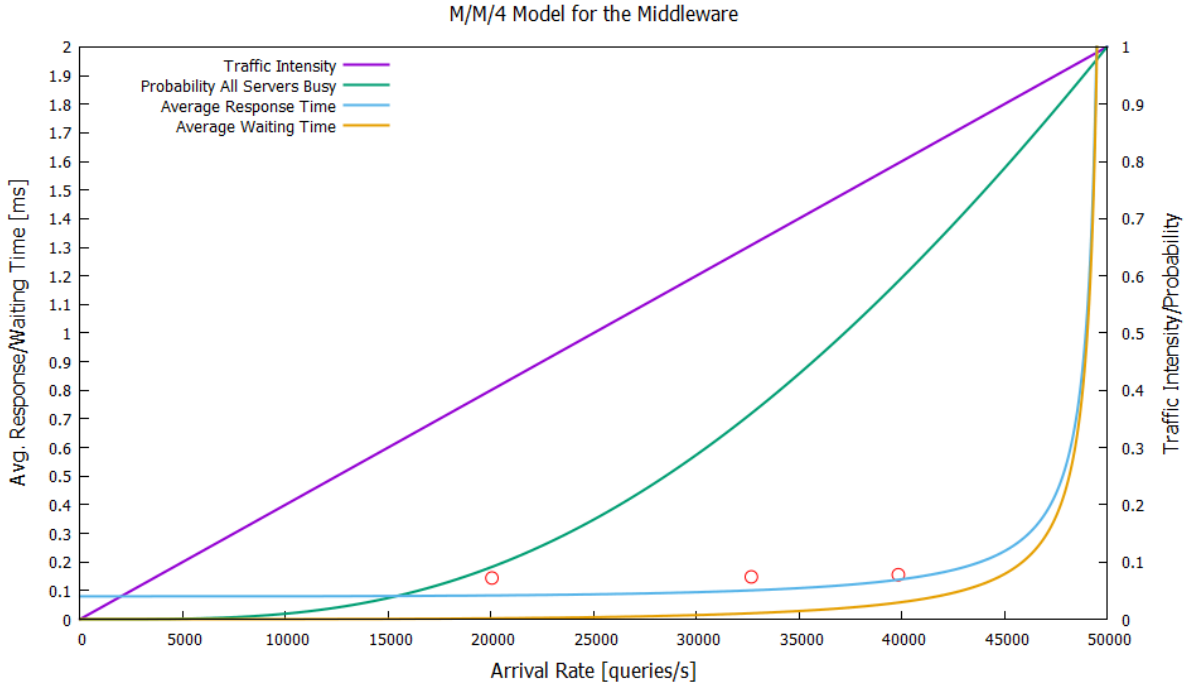

Figure 5

In general, a M/M/4 model is a good fit. An additional experiment roughly matches the model's calculated response time predictions. 32 clients are run remotely and log response times. The network delay of 0.163 ms is again subtracted but this does not account for the shared nature of the network connection nor does it account for the shared nature of the network socket. When the traffic intensity increases the model predicts the response and waiting times to grow faster than it does in experiment. This is because we use a M/M/4 model but in reality there are 32 client handling threads which offer a slower service rate instead of queuing requests, so the response time increases slower in reality. For a large number of lengthy messages, the network would have a significant impact on the response times, possibly even limiting the throughput. In the queuing network which includes models for the network this is accounted for.

3.2. Database

To build an M/M/m model of the database the service rate of the database needs to be measured first. A new experiment is run where I measure how long it takes for the database instance to respond to Send, Pop and Peek request. The experiments work by sending 10000 requests of each type one by one using only one database connection from the middleware without involving clients. Using the response time measurements, the service time and service rate can again be approximatively determined. Since the database runs on an instance with 4 vCPU a M/M/4 model is used. The PostgreSQL database starts with roughly 150 000 messages of 200 characters initially. All Send requests contain a 200-character random message string. The following has been measured (See log files for 95% confidence intervals):

Message Type	Remote Response Time [ms]	Local Response Time [ms]	Network Time [ms]	Service Rate [req/s]	% Typical Load
Send	1.259	1.006	0.253	994	40
Peek	2.188	1.828	0.360	547	20
Pop	3.809	3.674	0.135	272	40

Network delays of around 0.227 ms, similarly to the delay between clients and middleware, are measured and the response time measured between middleware and database minus the delay is used as service rate. Most of the time is lost between network interfaces and the database but not on the physical transmission. All Amazon instances managed at least 300Mbit/s down- and upstream (to the open internet) when measured. Assuming a 200 byte message is sent back and forth this results in only around 0.01 ms transmission time. In the queuing network presented later the network is modelled separately, here the network time is not counted as service time of the database. We end up with an average service time of 2.22 ms for each database thread, resulting in a total service rate of 4*446 = 1784 requests/s.

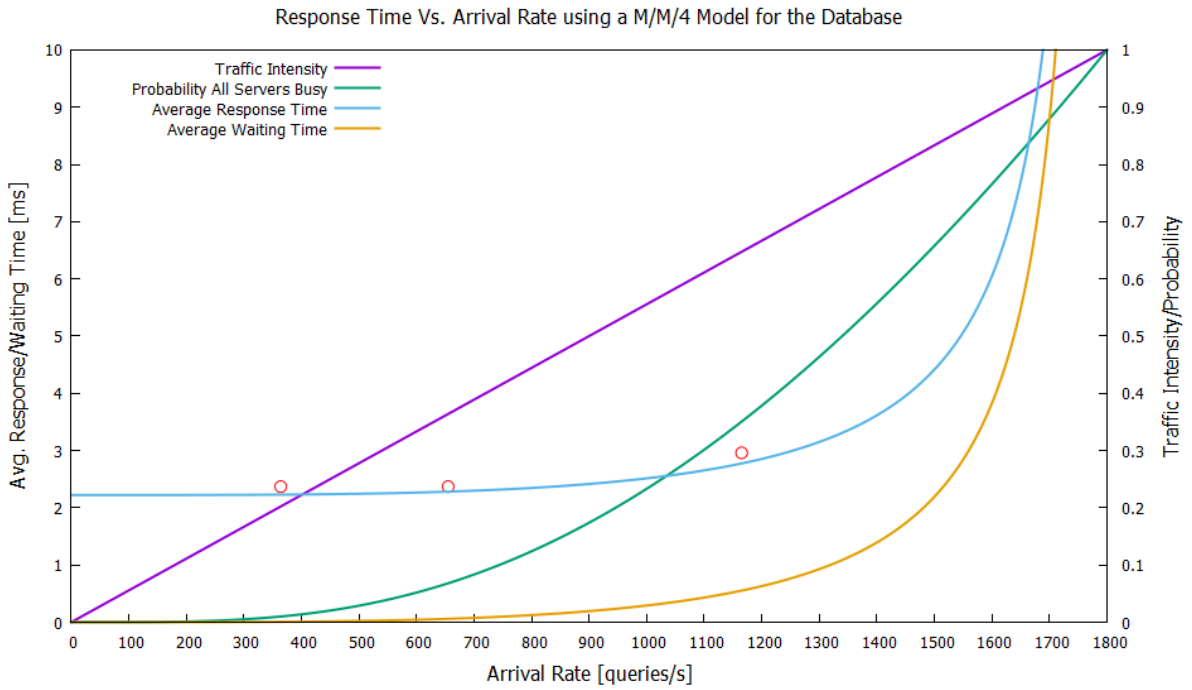

Figure 6

A separate experiment is done to compare the performance of the database under normal load with the model. The actual throughput is slightly lower because of disk I/O and also because Pop and Peek prevent each other from accessing the same queue using locks.

4. System as Network of Queues

The system is modeled as a closed queuing network where a fixed number of clients are assumed to be terminals in the network. The parameters used here match the stability experiment the closest, i.e., two middleware with 4 database threads each are run. Each system component is modelled as a M/M/1 queue. The network between clients and middleware is assumed to be representable as 32 queues, i.e., each client gets its own network connection initially. This is of course a simplification, since the network is shared among clients and the number of clients is not fixed. Similarly, the network between middleware and database is modelled as 8 queues, which equals the number of database threads in the middleware. Even though requests are queued in the middleware until the database is done with the previous request it still makes sense to model the network separately because the database instance implements 8 connections but only 4 threads are busy with actual database operations. In other words, the requests can still be partly queued in the database instead of the middleware and we get more accurate response times but less accurate throughput predictions. The system is evaluated using the mean value analysis algorithm using the parameters listed below in the order they are listed. See /scripts/MeanValueAnalysis/ for implementation details.

Parameter	Number of servers	Service time [ms]	Visit ratio
Clients	Any	N/A	N/A
Network Client – MW	32	0.183	1/32
MW: Client Service Thread	8	0.08	1/8
MW: DB Thread	8	0.04	1/8
Network MW - DB	8	0.227	1/8
Database	4	2.24	1/4

Figure 7 and 8 show the calculated throughput and response times of the analysis compared to measurements done with the stability experiment configuration.

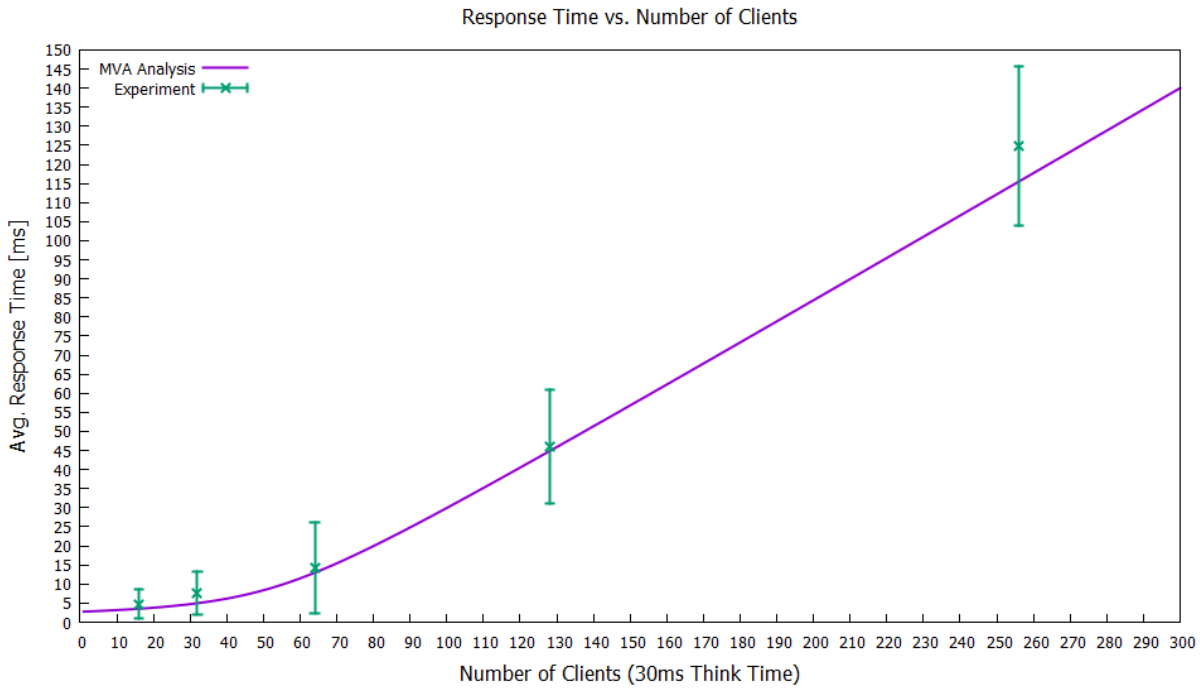

Figure 7

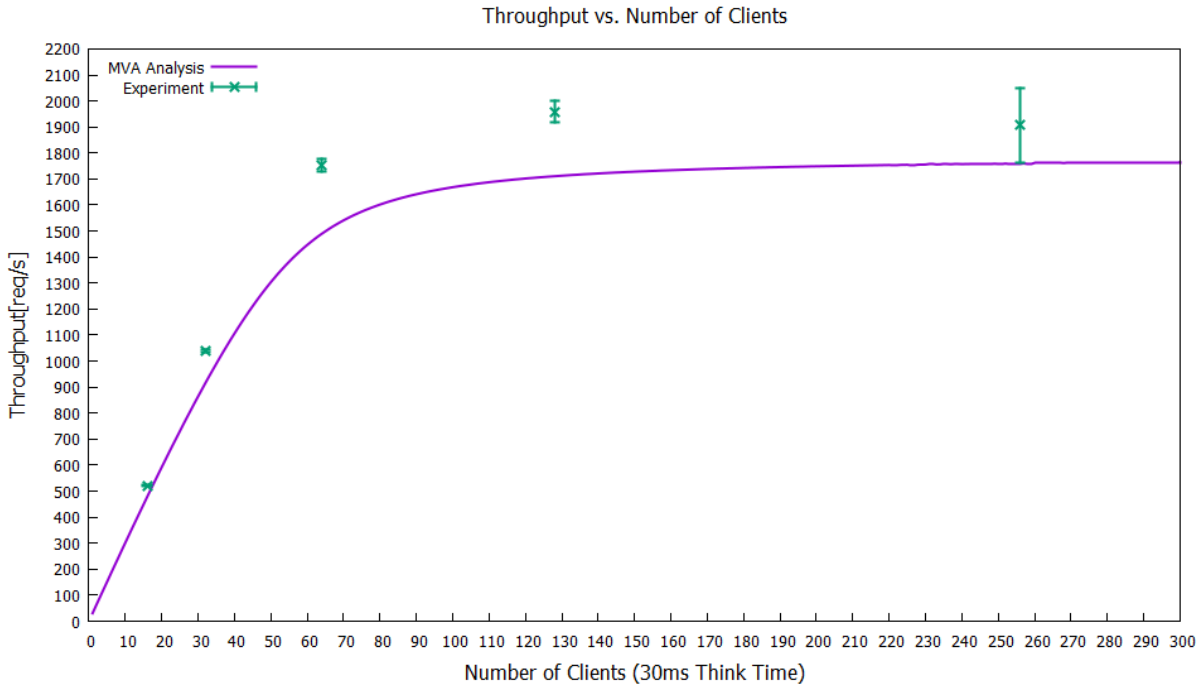

Figure 8

While the calculated response times lie within 95% confidence intervals of the experiments, the throughput has a slightly higher peak in the experiment. In the queueing network all components are modelled as M/M/1 but in the real system some parts share a common queue which leads to lower response times in some cases. PostgreSQL also spends a considerable amount of time parsing the requests which can be gathered from PostgreSQL's "explain analyze" time measurements compared to local response time measurements. This could potentially be executed in parallel to the actual disk access work and explain some of the higher throughput achieved when using 8 database connections. In the following the utilization of each system part

is compared. The database shows by far the highest utilization while other components are mostly idle.

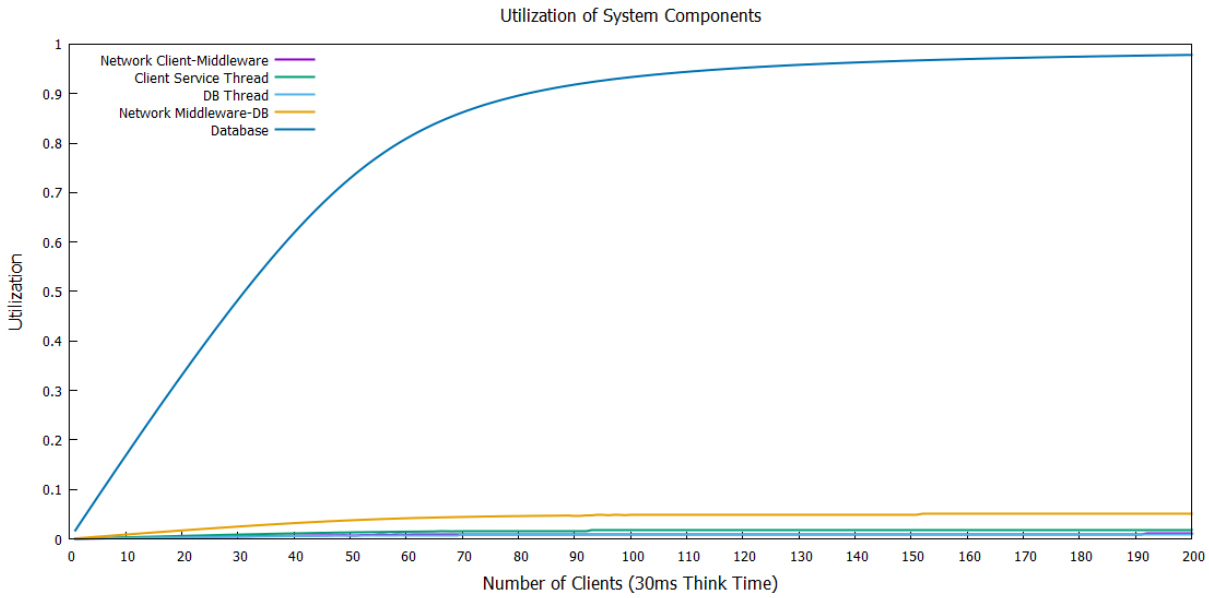

Figure 9

We can also use the model to predict how a faster database would affect the throughput as shown in Figure 10. The database speed could be increased by making the stored procedures more efficient. In fact, a 4 times faster database throughput should be achievable. Assuming the disk IO does not limit the database's performance we could also run the system on an instance with more available CPU cores and adapt the number of parallel servers in the database model accordingly.

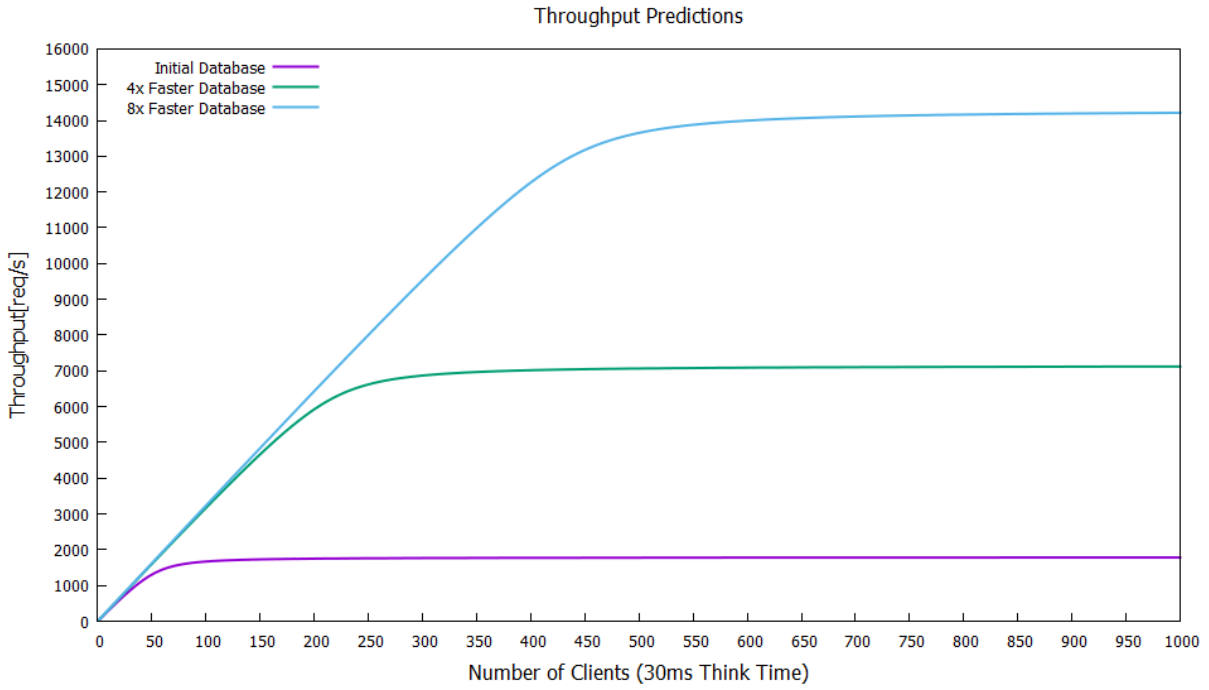

Figure 10

Figure 11 shows the predicted utilization of each part of the system using an 8 times faster database.

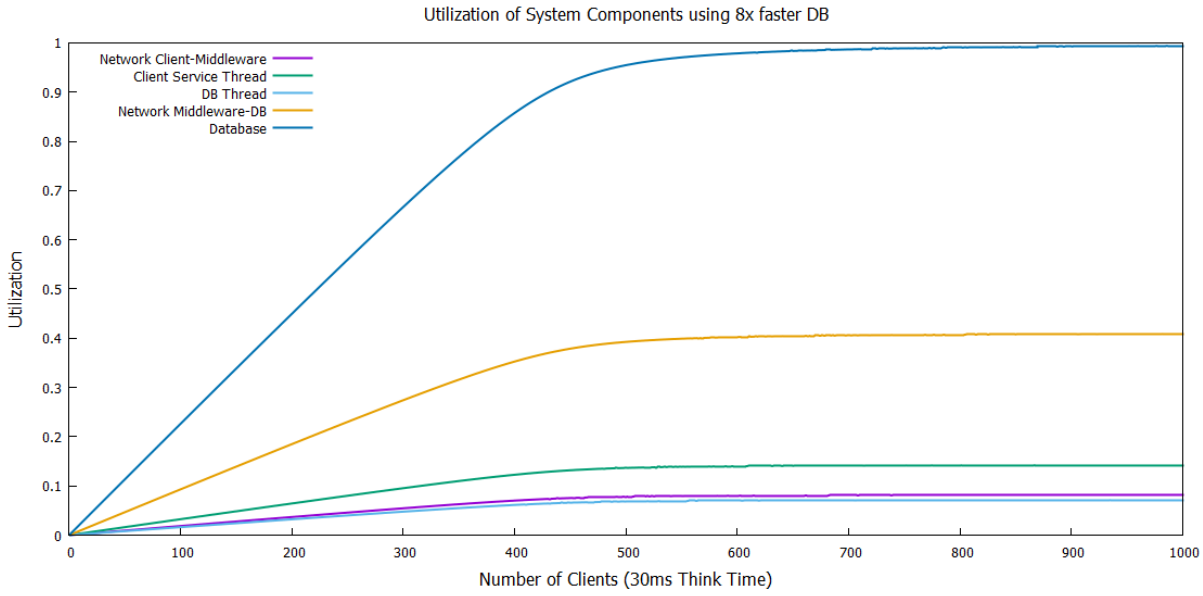

Figure 11

The network between middleware and database now has a significantly higher utilization. In reality, the utilization is not that high, i.e., it is similar to the utilization of the network between client and middleware, but we can predict that the number of connections between middleware and database needs to be scaled up when the database speed is increased to keep response times low. We can also infer that having more than 1 middleware would only become necessary for a lot faster database since none of its parts are close to their performance limit. When using larger message sizes, the network's maximum throughput could also play a role in how the system performs and the network could become a bottleneck. Modelling the network in more detail would require knowledge on how Amazon's network is shared and how the OS and PostgreSQL handle network connections.

5. Interactive Law Verification

The interactive response time law equation describes how a system's response time R, throughput X, think time Z, the number of clients N and runtime T relate to each other:

$$system\ throughput\ X = \frac{total\ number\ of\ requests}{total\ time} = \frac{N(T/(R+Z))}{T} = \frac{N}{R+Z}$$

And similarly,

$$R = \frac{N}{X} - Z$$

Using it we can check the validity of the experiments by predicting the system's throughput using the measured response times and comparing the results with the measured throughput. The following shows the stability experiment measurements. Note that in this experiment (which was required for milestone 1) the think time includes time waiting for a reply. This leads to a constant flow of 1 request/30ms.

Clients	Z[ms]	R[ms]	X predicted [req/s]	X measured [req/s]	Difference in %
32	(30-4.98) = 25.02	4.98	1067	1088	1.96

Data from the maximum throughput experiment, found in /logs/MaximumThroughput/, is shown below.

N/clients	Z[ms]	R[ms]	X predicted [req/s]	X measured [req/s]	Difference in %
4	0	3.58	1120	1121	0.1
8	0	5.70	1403	1411	0.6
16	0	10.93	1463	1482	1.3
32	0	21.65	1478	1514	2.4
64	0	44.25	1446	1530	5.8

The experiment's response and throughput measurements roughly satisfy the interactive law. In general, the measurements accuracy decreases when the number of clients increases. This is a result of having many running threads with many calls to timing methods such as sleep and logging which amplifies small inaccuracies.

Appendix: Repeated Experiments

Most experiments have been rerun for the second milestone. The following inefficiencies have been fixed:

- The java sockets now correctly implement buffered sockets and the throughput between middleware and clients is improved by a large amount.

- In addition, auto-commit previously enabled in the jdbc library has been disabled, it increases the speed by which middleware and database interact.

Note that the overall design did not change and the queuing network presented here fits both code iterations. A few methods are added to clients and middleware which allow them to measure and log service time. All experiments presented here are run on the Amazon cluster on Windows 2012 R2 instances. Each of the two middleware and clients are run on a m4.large (2 core) instance while the database runs on a m4.xlarge (4 core) instance. The instances used here overall perform worse than the ones used in the previous milestone. As before, most stored procedures execute many SQL statements in sequence. Pop/Peek and some others implement locks which prevent them from accessing the same queue concurrently. The database starts with approximately 150'000 messages distributed among 100 queues in all experiments. For these reasons the database performance might be lower than expected. However, this does not affect the validity of the queuing models but only lowers the service time of the database. Clients now also have the ability to sleep in between sending requests. Because clients always wait for the reply from the middleware this was previously not used. In this milestone it is used to check predictions regarding the load on the system and number of connected clients. Event logs (as opposed to throughput, response and service time logs) are mostly omitted because of their large size.

Stability Experiment

For the stability experiment 32 clients are run with a think time of 30ms over 30 minutes. Note that the think time here includes time waiting for a reply, which is not the case in other experiments. Each middleware communicates with 16 clients and connects to the database with 4 threads. The client sends a typical workload, defined to be 40% Pop, 40% Send, 20% Peek to ensure a roughly constant database size. Messages consist of 200 character random strings. A throughput trace is shown below and the throughput is averaged over 1 second intervals; the full logs can be found in /logs/30MinStability/.

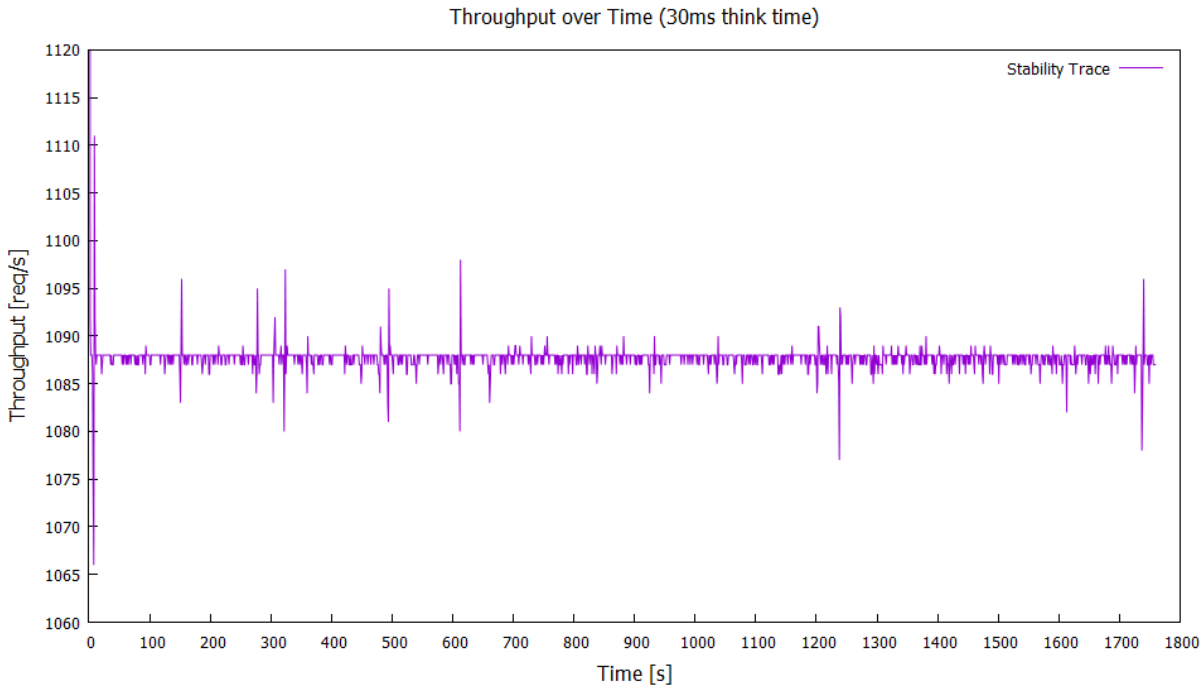

Figure 12

Scalability Experiments

For the scalability experiments 4 different configurations are examined.

- 1 middleware 4 database connections
- 1 middleware 8 database connections
- 2 middleware 4 database connections each
- 2 middleware 8 database connections each

16 clients connect to each middleware and send a mixed load of 40% Pop 40% Send and 20% Peek messages. Log files can be found in /logs/MWDBScalability/.

Maximum Throughput Experiment

1 middleware using 8 database connections is used where up to 64 clients connect. Log files can be found in /logs/MaximumThroughput/.

Service Time Measurements

Database

10000 requests of each type (Send, Peek, Pop) are sent one by one both from the middleware instance and locally on the database instance. Log files can be found in /logs/DBMeasurements/ServiceTime/

Middleware

200-character echo requests are sent from clients to middleware one by one using different number of clients over at least 1 minute each. Response and service times are measured both with remote clients and local clients. Database threads in the middleware return echo requests without accessing the database. Log files can be found in logs/MWMeasurements/Remote/ or /Local/ respectively.

Mean Value Analysis/ Response Time Law

Both are implemented in C# and can be found in code/Milestone2/MVA and code/Milestone2/Response Time Law respectively.