Advanced System Lab – 2014

Project report – milestone 2

Amrollah Seifoddini Banadkooki

Changes to the system and experiments

In previous version of my queueing design, there were some design and implementation mistakes which prevent correct data collection. They are listed in the following with the changes that has made to fix them.

1. There were two queues holding client sockets and client IDs separately. As you know, in multi-worker cases it will lead to not matching client_id and socket in some workers. So, I replaced it with one queue holding both socket and ID as a java class. This class also holds the arrival_timestamp of a client request. In log_parser I use this data to calculate waiting time in middleware queue.
2. There was a mistake in one of the log lines which caused not matching measurements for response time and throughput according to interactive law. The place that I was getting the timestamp in client for "get message" operation, was after sending the "get pending queues" query. So, the time for that expensive request was not included in the get_message response time. As a result of that, my measured response time was lower the actual one. To fix that, I could consider "get pending messages" as a separate request, but I decided to remove that query in my experiments since it is a really expensive request compared to send or get_message. In other words, including that in the normal requests, leads to high variance in measured response time and measured model parameters.
3. There was other requests including create_queue, delete_queue, read_message in my client experiments and they were used with a certain probability. Since these queries are different in terms of response time, having them in experiments leads to high variance of data. So, I decreased their probability to zero, meaning that they will never happen in experiments. So, we only have "send" and "get" message operations in the clients and system. This also keeps the database size roughly constant.
4. In my connection pool design, every prepared_query had a dedicated connection during the lifetime of middleware. Although it was better in performance, but it leads to having many connections which some of them might not be in use at all. So, I fixed that by getting the connection from connection pool for every query -including prepare queries-. So, before executing the query, they prepare the statement in the connection which is passed to them. JDBC cache helps to remember the statements which are prepared for connections.
5. For improving the DB speed, I replace prepared statement by preparedCall of the pre-stored functions in database. I have functions in DB for all the queries that clients might send.
6. For further simplifying experiments, I changed the load generator part of client, so every client machine starts with a list of client_ids and a middleware IP address that will service all clients of this machine. Client_ids are counted from offset (which is passed to the client creator) until offset + client_count. So, in case of several client machines, one should set the offset accordingly. In every client machine, clients only send and receive messages to each other, so there is no need to get or update the list of all clients from the DB. First client send and receive messages to/from last client in the very same machine, second client communicate with the client before last, and so on. Another simplification is done for queue_id in sending or receiving part. I assume there is a dedicated queue for every client with the same ID as that client. So, queue_id is simply set to client_id in send/get message functions. Before running the experiments, I make sure that there are enough client and queue entities in the database. This way, clients don't need any other

information from database during the experiments and also because of this ping-pong approach, size of database does not change.

7. I added some logging for measuring the waiting time for getting and preparing connection from DB connection pool. The measured values for every middleware are stored in corresponding csv file.
8. In the client, after writing in the socket, there was not flush() to send the request right away. So, it was done by operating system and in some experiments with only one client, it caused longer response time because of this delay in socket pushing. So, I fixed this by flushing the socket after every command.
9. In experiments, I don't count get_message request which return no message, or send_message request which return Error.
10. get_message function in previous version of middleware, was consisting of two query parts. First query, only read the message from DB and if it was successful, the second query delete that message from DB. This was not necessary. So, I replaced it with one DELETE query which also returns the deleted message row. This change reduced the response time of get_message command. Note that the extra queries for error handling are still in place. For example, if the DELETE query does not return a message, a query will be send to check if the requested queue exist or not. The same error handling approach is done in send_message as well.
11. I changed the DB connection isolation level from serializable to repeatable read. I still can guarantee ACID of my database, because there is not repeated read in my transactions which lead to phantom read problem.
12. Logging is added for socket read and write time both in client and server. Log parser changed to include them and csv data files are generated for them as well.
13. All the new logs are compressed and put in file named milestone2_logs.rar in log folder of repo.

1. System as one unit

- M/M/1 model of entire system based on trace

For modeling the whole system as one unit, I consider middleware, database as the system. Client load generator is creating request and practically not a part of system. Also because of its negligible think time, it does not make a considerable difference to include it in the system or not. The requests are sent to request queue and then system picks requests from the queue, process them and return the response to client as output.

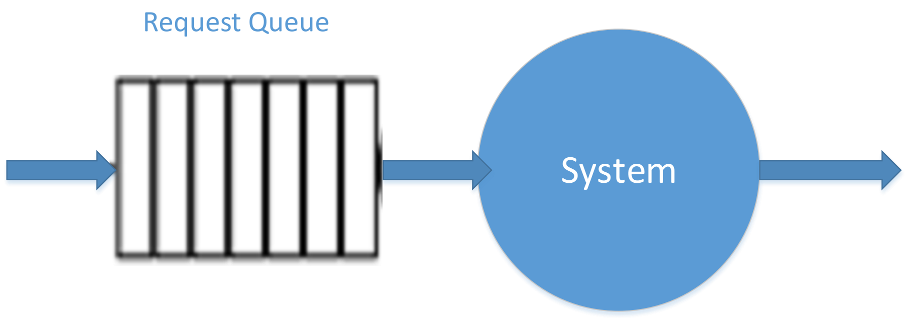

Figure 1: M/M/1 model of whole system

- Characteristics and behavior of the model built

For calculating the service time of system, I did a simple experiment with one client, one worker in one middleware. The size of connection pool in middleware is set to 10 for all the experiments except if it's explicitly mentioned otherwise. During the experiment, system takes log from the point it picks the request from queue until the point where respond is sent back to client and client has been put back at the end of queue. This time interval is called service time for the system. It shows how much time the system spends for responding successfully to a single request. In my experiments this time was between 4.9 and 5.4 ms in different repeats but with high variance. So, I average it to **5.04 ms**. This number includes the time a request spends in middleware, database and links between them. It does not include client think time, but it is only 0.2 ms in my experiments. Based on my detailed logging points in different parts of the system, I could further divide this whole time to component service times. These individual service times are shown in the table below.

Table 1: service time of components

Middleware (think time)	**DB**	**Connection pool**	**Network between server and DB**
0.0019	0.0011	0.00079	0.00125

We see in the table that service time for middleware is the biggest service time (and a possible bottleneck). Service time for DB is only the time for execution of query. Also, the service time for connection pool is only time for getting a connection. All other times including picking client from queue, processing command, preparing the connection and prepared queries, putting client back into queue are considered as middleware service time. The network time between server and DB is simply computed based on average ping time. Although this ping time is really high, but it is possible that server and DB are in different locations and network hops between them take that long.

Based on this service time, service rate is 1/0.00504 = **198.5**. In other words, the maximum number of requests a worker can handle stably is 198. Note that when the arrival rate goes near this service rate – utilization gets close to one- system becomes unstable and queueing model formulas does not work anymore for predicting response time. For computing the response time of this model, we need arrival rate too which is equal to throughput of our closed system assuming that all requests are successful. I designed the experiments this way and also checked it manually to make sure this assumption holds.

I use this formula for computing the response time: here μ is service rate and ρ is traffic intensity (or utilization) which is defined as λ/ μ; where λ is arrival rate.

$$E[w] = \frac{1/\mu}{1-\rho}$$

- Comparison with the experimental data

For generating different load with low arrival rate, I put the client to sleep for 5 or 15 ms after each request to generate 90 and 45 req/sec arrival rates overall. If I use the full power of clients, the arrival rate is around 175 req/sec which is too close to worker maximum capacity and that response time formula does not work anymore. Table below compares the model results with the measured response time of some experiments. All these experiments are done with only one worker because our service rate is valid for only one worker in M/M/1 model, but number of clients can vary to produce different loads.

Table 2: Comparison of M/M/1 and measured data

Service rate	**λ**	**ρ**	***Model response time***	**Measured response time**
198.4127	45.9	0.231336	0.006557	0.006
198.4127	90	0.4536	0.009224	0.006
198.4127	173	0.876355	0.040762	0.0056
198.4127	197	0.99288	0.707865	0.804
198.4127	198	0.99792	2.423077	0.81

This table shows that for low arrival rate (low utilization), the model can predict the response time relatively good, but as the utilization goes upper, the difference between model predication and measured one gets bigger.

- Analysis of the model and the real behavior of the system

The model is designed to have exponentially response time with increasing arrival rate. That is the pattern we see also in the experimental results where it gets close to service rate. This is because we have a high queueing effect in those cases and that leads to higher response time. But the length of queue in our closed system cannot be bigger than client size, because clients wait for a respond before sending another one. In some experiments, we have only one client and hence there is no queue wait time. In other cases (specifically last two rows of table) it is less than 1 second, but his model cannot predict that correctly and response time goes way off as soon as we get close to 80-90 percent of service rate. Another point is that in first two rows, experiments were done with manual pauses between requests (but these pause times are not counted in response time). So, client could send different load amount with the same response time. But model could not comprehend that and returns a higher response time based on the arrival rate. The observed response time should be simply service time + network time between system and client. The service time is 0.0051 and the network time is 0.0006 based on ping time statistics, therefore, the expected response time without queueing effect should 0.00057. Since the arrival rate is much less than service rate, there is not considerable queueing effect. So, the measured values are close enough to expectation based on system implementation.

2. Analysis of system based on scalability data

- Queuing models for different configurations

As explained in my report for milestone 1, I divide scalability experiments into three category, speed up, scale up and scale out. For speed up, the number of clients remains fixed but worker thread count in middleware increases. In scale up, the number of clients increased proportionally to number of worker threads. For scale out, middleware machines are added proportionally to client number increase.
In these models, a worker means the whole system including middleware and database and network links (unless otherwise stated).

Speedup and Scale up:

The M/M/m model for speed up and scale up is very simple. There is a queue which feeds several workers and they provide clients with service.

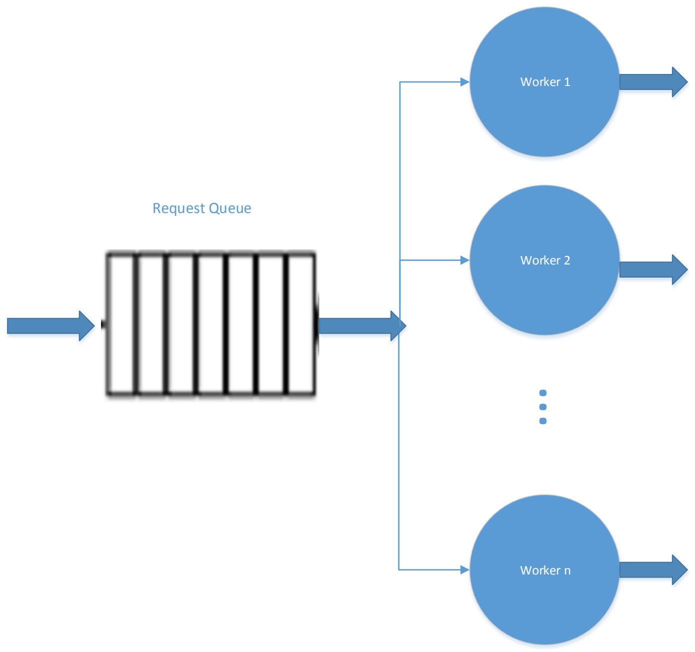

Figure 2: Simple model for speed up and scale up

Service time for every worker here is the same as M/M/1 model of whole system which is 0.00504 seconds. That means every worker’s maximum capacity is around 200 request/sec. So, by adding n worker we should theoretically reach the n*200 request/sec and 1/n of previous response time in scale up/speed up. Here are the results for speed up experiments and predicate values of model. For computing the M/M/m model response time I used the formula in page 529 and 530 of text book. The python code for computing that is available in the script folder of my codes.

Table 3: Speed up results

Client number	Worker count	Response time	Throughput
40	1	0.241	162
40	2	0.116	341
40	4	0.082	487
40	8	0.071	550

Table 4: Predicated values of M/M/m model for speed up

Worker count	Arrival rate	M/M/m response time
1	162	0.027
2	341	0.021
4	487	0.0062
8	550	0.0050

From the table, we can see that predicated values do not match to the measured ones. The reason is that utilization is between 80 and 90 percent for these experiments, so it is not surprising to see queueing formulas not matching with real data. One of the main reasons of scalability experiments for me was to find the maximum throughput. Therefore, aiming for this full utilization is inevitable. Also, I could make M/M/1 model for this part, instead of M/M/m. surely, it will have be more simple but less accurate. For doing that, one should multiply the service rate by number of workers to get the cumulative service rate. Then the rest is just like a normal M/M/1. I did that for speed up experiment and for example M/M/1 result for 4 workers is 0.0032 which is again suffering from high utilization. M/M/1 values for 2 workers and 1 worker are 0.017 and 0.027 respectively. We can see that in measured data that adding n workers does not always have N times effect in the throughput or response time. That means, by adding 8 workers we won’t be able to achieve 1600 throughput. The reason is that service time of middleware is around 0.002 (refer ro table 1) which means the maximum throughput for a middleware machine would be 500 request/sec. So, after reaching that value, machine is at full utilization, and adding more workers will not increase the throughput anymore.

Table 5: scale up results

Client number	Worker count	Response time	Throughput
40	1	0.241	162
80	2	0.237	337
160	4	0.314	508

Table 6: Predicated values of M/M/m model for scale up

Worker count	Arrival rate	M/M/m response time
1	162	0.027
2	337	0.020
4	508	0.0065

Again, we see that high utilization (arrival rate so close to service rate) caused not matching response time for the M/M/m model.

The other effect comes from OS scheduler which allocate CPU time slots to all threads and processes running in the machine. This thread switching is not a free task. So, by adding a worker, we are decreasing the amount of CPU share that every worker can get in the machine. This means a higher service time per worker and hence, a lower service rate for them. I try to model this effect in the model below in figure 4. It has an extra queue for threads waiting for a CPU slot. If the number of threads get high, this wait time in considerable. In my experiments, it is observed that service time of every worker increased by 1 ms by adding a new worker to the machine assuming very high utilization. This time is mainly caused by latency of database response, but since in this model we are considering database as part of the worker service, so, we can assume it as the scheduler service time. For example, effective service time for 4 workers will be 7.5 ms and for 8 workers it is 14ms which is affecting the throughput too. In other words, because of considerable think time of middleware, it is a bottleneck in the system. That is, with only one worker, we can get to maximum of 200 throughput because system behaves like one unit with overall service time of 5ms but if we add several workers we are removing this bottleneck and sending a huge load to database. This increase stops around 500 as throughput because middleware think time is 2ms and service rate is 500. So, with scale up we cannot reach beyond 1/0.0019 = 550. The data in table above also confirms that. We see that rate of increase in throughput decreases as the worker count grows.

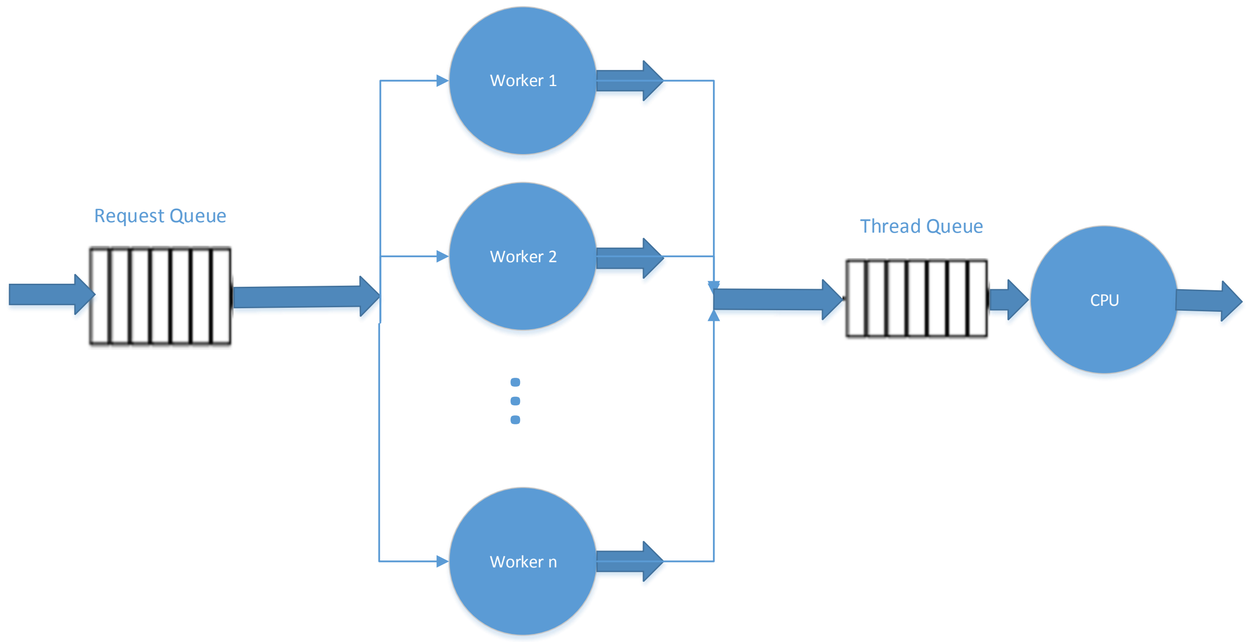

Figure 3: Better model for scale up

Scale out:

For scaling out, we add server instances as well as clients. I tried 1, 2 and 3 servers with 40, 80 and 120 clients respectively. Worker count is fixed to 4 for all the servers. The results are shown in table below.

Table 7: scale out results

Client number	Server count	Response time	Throughput
40	1	0.082	487
80	2	0.0966	824
120	3	0.1289	937

Table 8: Predicated values of M/M/m model for scale out

Server count	Arrival rate	M/M/m response time
1	487	0.0062
2	824	0.005
3	937	0.005

From the table 7 we can see that by increasing the server from 1 to 2, throughput almost doubles. Yet after adding the third server throughput only improved by %8. The reason is saturation of database service rate. If we look at table 1, database think time is 0.0011 and that tells us its maximum service rate is 930. We theoretically cannot exceed that 930 throughput. So, after 3 server, no matter how many servers or workers we add the throughput will not increase, but the load balancing between servers might improve. So, for modeling the scale out, I consider all workers of all servers as service providers which will pick requests from queue and handle them. The service rate of every worker is 198. Assuming that clients are uniformly distributed among servers, I average the response time of clients and add the throughput of servers to get the final values for them.

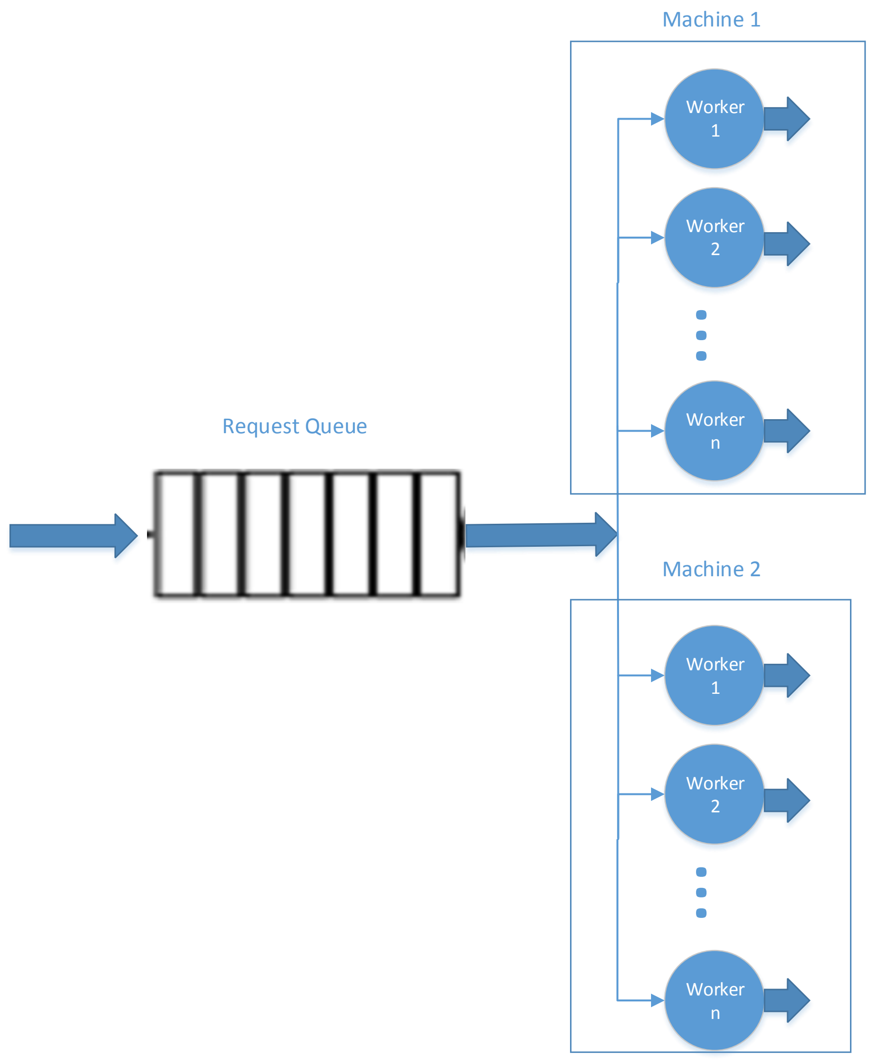

Figure 4: M/M/m model of scale out

3. Modeling of the components as independent units

My system consists of middleware servers and one database layer. Every middleware also holds several workers to handle requests.

- M/M/m models of database and middleware

Middleware:

In the M/M/m model of middleware, requests are served from the request queue by several workers. Then each worker process the command and issue relevant database query. These queries are put in the query queue which will be served by database connections (from connection pool). Every middleware can have m worker and m DB connection. Service time of a worker is 0.0019 (according to table 1) and service time of connection acquiring from pool is 0.00079. So, the corresponding service rates are 526 and 1265. Obviously, the worker time is bottleneck here and if we have m >= n, there will be almost no wait time in connection pool. If m<n then the service time of connection pool is service time of database (0.0011) because worker have to wait for connection to do its job and get available. The overall response time of middleware is sum of response time of these two parts.

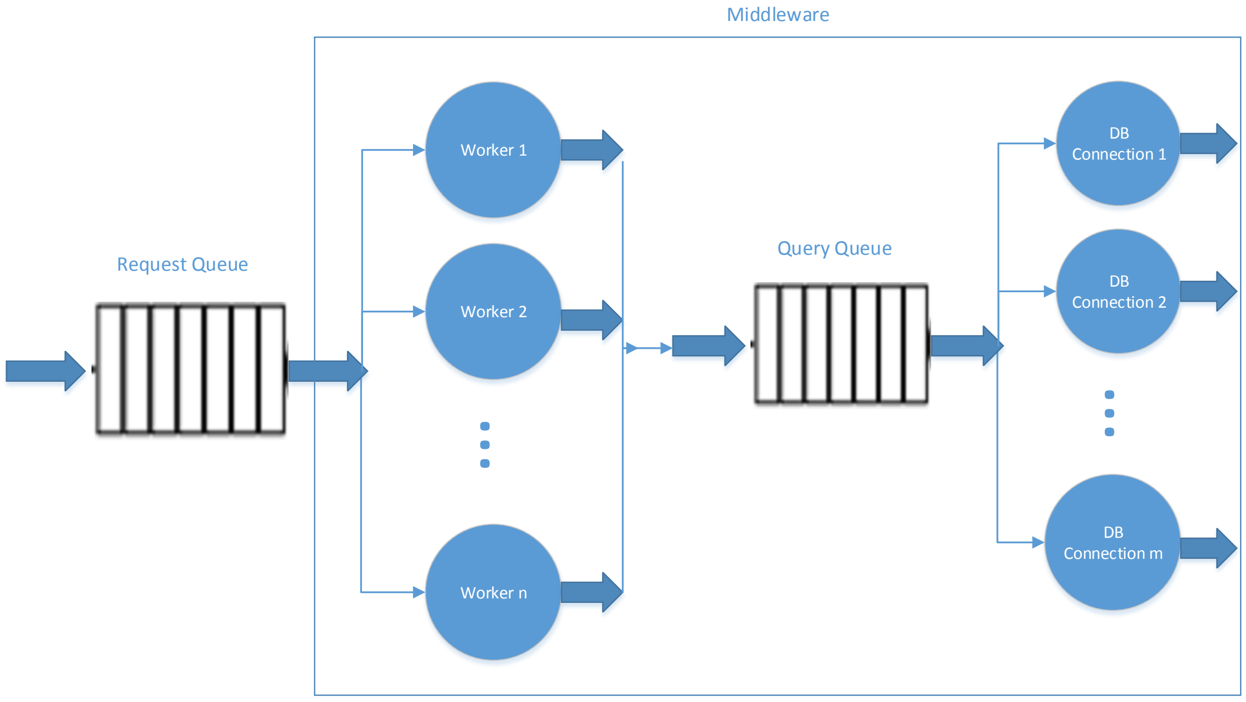

Figure 5: M/M/m model of middleware

Based on the log data, I have response time of each of these components. So, I am going to show some of the experiments values as samples to verify the model predication.

Table 9: Comparison of middleware model to data

Worker count	Connection count	Arrival rate	Measured response time	Model response time
2	10	341	0.00198	0.0029
1	10	45	0.00188	0.0027
8	4	484	0.0102	0.0031
4	10	530	0.003	0.0026

We see that in first two cases, the model could predict the response time approximately, but in the third row, there is not enough connections for workers, so wait time of workers increases and it leads to higher response time. Model could not predict this value good, because the utilization of connection pool is 100% here, so the formula does not work well.

Database:

For the database, M/M/m model consists of a queue of connections which is served by RDBMS processes in the operating system. Every connection is holding a get/send qurry. Usually number of these processes is equal to number of cores in the machine. In my case, the db machine has 2 CPU cores, so there are two db processes handling queries. The average service time of every process is 0.0011 (according to table 1). This means DB can handle around 950 requests per second.

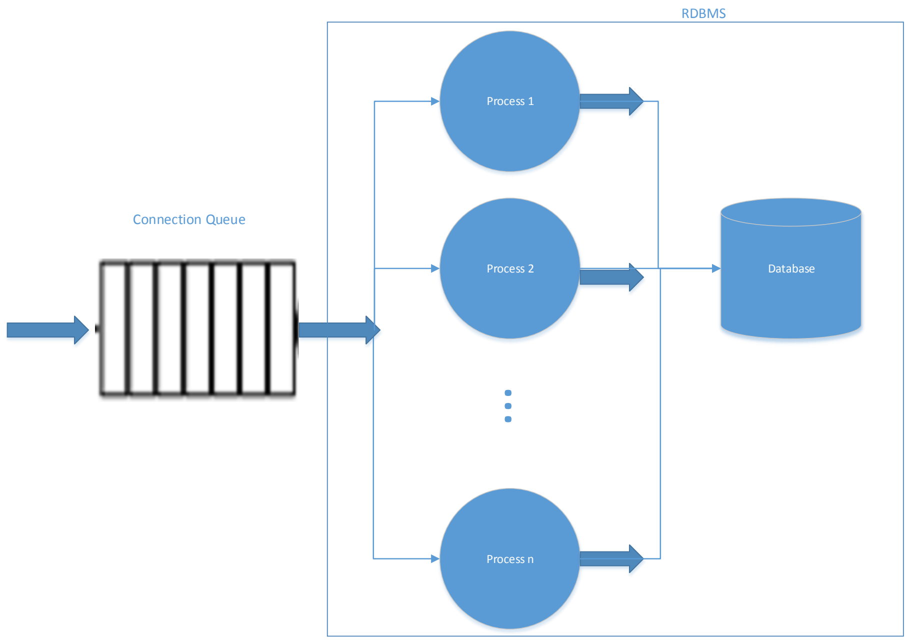

Figure 6: M/M/m model of database

But we still can make a better model of database. Since execution time of get_message queries are slightly bigger than send_message queries, we can assume two different queues for these two queries. In the figure8 it is shown that every queue is served by the same set of DB processes that are available in system. Note that these processes are technically the same, but service time of them is different while handling each queue. So, I draw them twice and as different service providers. Based on the log I know that DB time for get_message is double the DB time of send_message. So, service time for get_message is 0.0015 and service time for send_message is 0.0075. But as in my experiments, ping pong approach is used, number of send and get requests are exactly the same. This means that probability of routing query to either of those queues in RDBMS is 0.5. So, it is fine to assume them as a single queue with the average service time which is 0.0011. Based on the scalability experiment we know that maximum arrival rate was 937 and that confirms this service time (0.0011). And for all values blow 500, the DB time remain around 0.0013 which is consistent with the data logs. Note that in data logs, we also have the network time between server and DB. After subtracting that, the DB time is around 0.18 in normal cases with several workers. There are some variance in the DB time but it could be because of high network time which might be bursty.

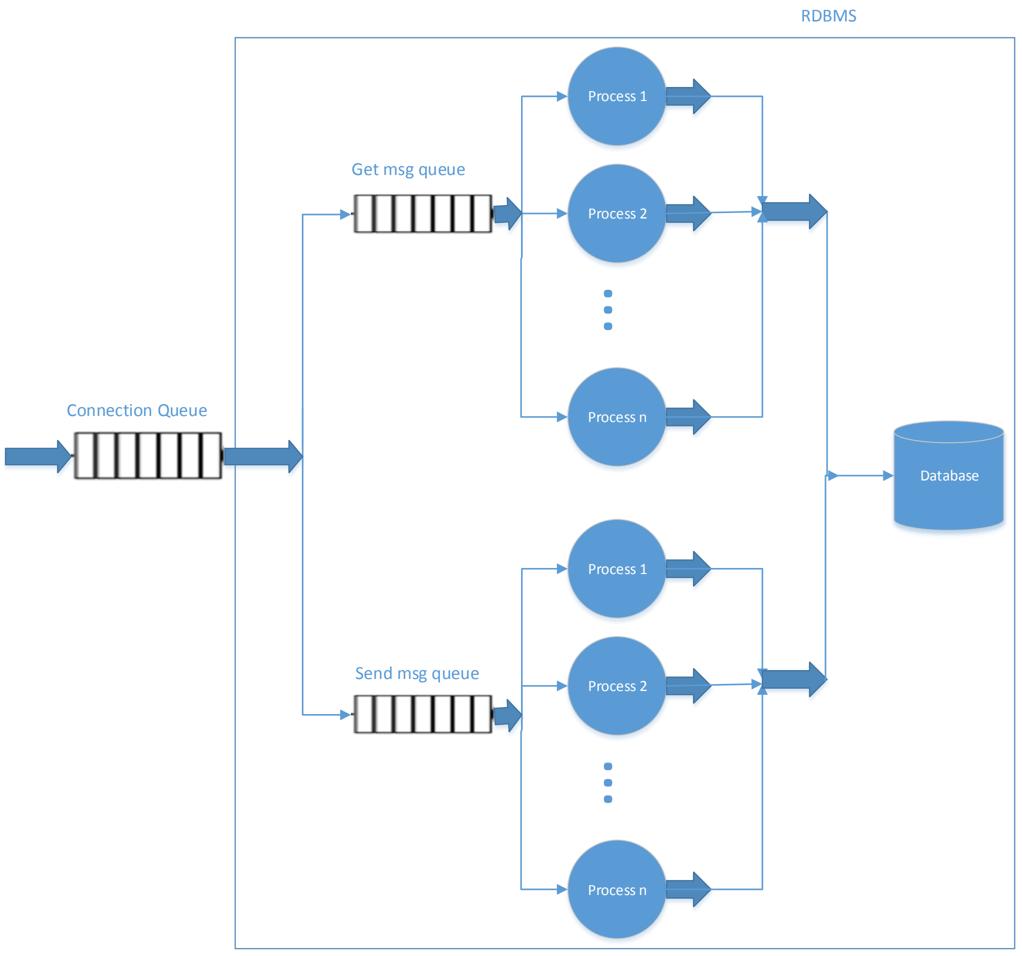

Figure 7: Better model of database

4. Model of systems as a network of queues

- Model

For modeling the whole system as a network of queues, I also considered network latency between components as a service. The service time of this service is computed from ping time. So, I simply add the ping time to calculated response time from middleware or database to get the final response time value. Client is not considered as a part of system here, because we assume having enough load (their think time is negligible).

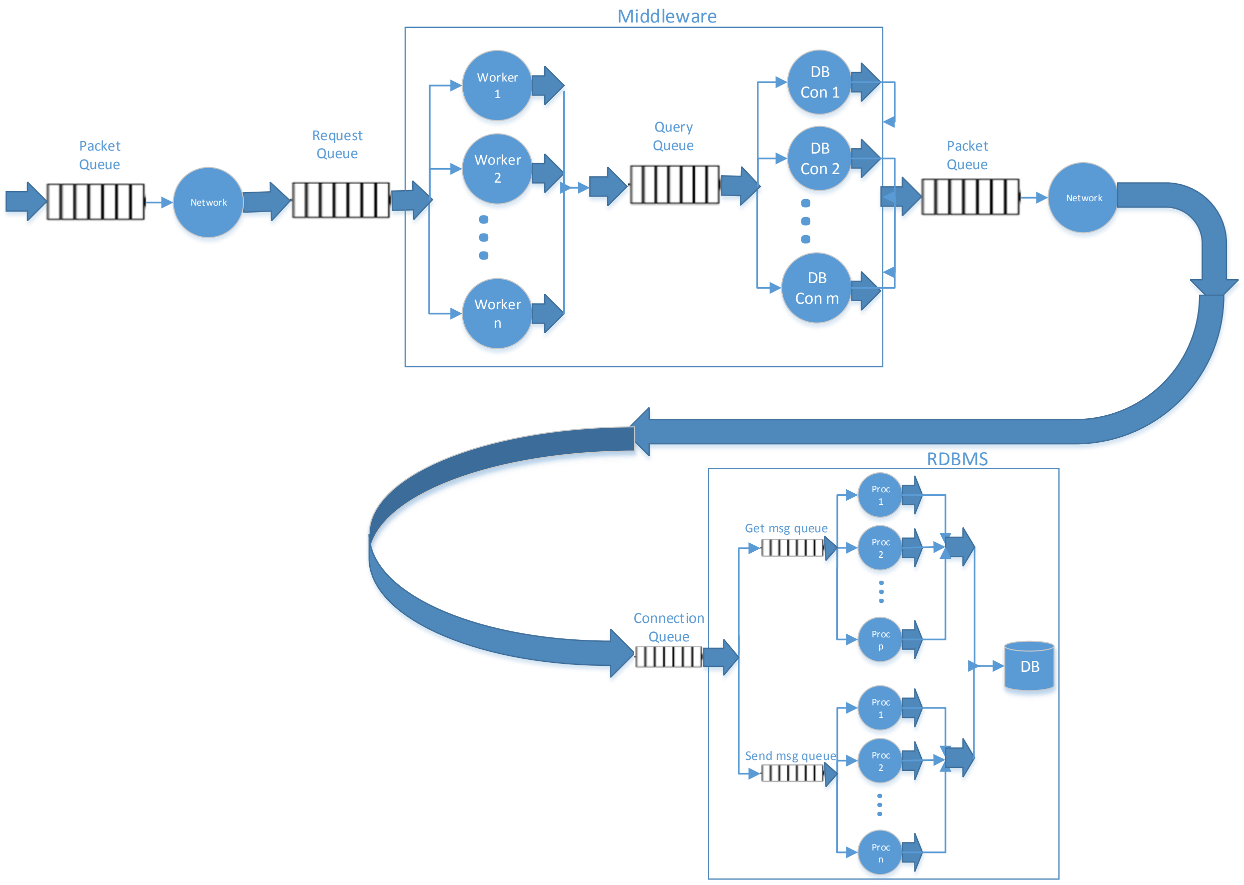

Figure 8: Network of queue model for whole system

- Characteristics and behavior of the model built

The request are send to packet queue, network process them and send them to middleware request queue. Every worker in the whole middleware tier (all workers of all servers) pick requests from the queue and process them. Then send the required query into query queue where DB connections serve them and put them in network packet queue. Network send them to connection queue of database machine where they are routed in appropriate queue based on type of query. Then DB processes pick the query, execute it and return the result. The service time of components in middleware and database are discussed in section 3 of this report. The only extra thing here is packet queues and network latency and combining all those small service time in one big picture. Network latency for client-middleware is 0.0006 second and for middleware-DB is 0.00125 second. Size of connection pool is set to 10 and there are 2 db process in DB machine. Service time of workers are 0.0019 and DB service time is 0.0011.

Table 10: Comparison of model with the experimental data

Worker	Arrival rate	Measured response time	Model response time
1	45	0.0066	0.00612
1	90	0.0058	0.0064
1	122	0.008	0.007
1	173	0.0056	0.0075
2	341	0.11	0.010

- Analysis of the model and the behavior of the system

It can be seen from table 10 that for low utilizations, model can predict response time well, but for high traffic intensities the model response time does not match with data. It is understandable because of sensitivity of models to load. The analysis of different parts of system has been done in other sections,

- Bottleneck analysis and identification and analysis

Big messages in our system are around 2200 bytes in size. Therefore, a small instance of EC2 in amazon which has 250 mbps bandwidth can send and receive about 8,000 messages per second. That means, if we ignore burst of AWS, network cannot be a bottleneck in our system. Also the think time of clients is 0.0002 which means they can generate 5000 request per second.
We identify bottleneck by computing demand based on this formula.

$$D_i = V_i S_i$$

In my system, every request visits the middleware, connection pool and DB only once. So, considering that visit ratio is equal, the component with highest service time is bottleneck. By looking at table 1, and also confirming that by scalability data, we see that at first, middleware is bottleneck, and at the next level, database is the bottleneck. If we look more detail, in database, get_message request is bottleneck. This is with assumption that we have enough connection pool size for workers. We saw that every worker can reach 200 service rate, and a middleware machine can reach 500 service rate, and database machine can reach 930 service rate in our deployment configuration. I use medium AWS instances for middleware and s3.large instance for DB. Clients are on small AWS instances. So, for scalability, one should first increase the worker count up to saturation, then add another middleware and get the highest performance with one database. Also by optimizing get_message queries we can reach higher performance in DB.

5. Apply interactive laws to experimental data from milestone 1

My old data from milestone 1 was not valid according to interactive law, because my log point in the client was not correct. So, actual response time was bigger than measured value. For this milestone, I fixed the problems and also simplified all my experiments to only contain simple send/get message requests. After re-running all the experiments, now the measured data completely match to the interactive response time law. Think time (Z) for my clients is 0.2 ms which is very low and negligible.

- Check validity of experiments using interactive law

In table below response time from interactive law is compared with the measured one for some of most important experiments of milestones 1. For computing the interactive law response time I used this formula. N is client number and X is throughput.

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

Table 11: checking validity of interactive law

Client count (N)	Throughput (X)	Interactive law response time	Measured response time	Experiment topic
1	173.8	0.00555	0.00558	Service time calculation
160	198	0.80788	0.81	Different client count
40	341.5	0.11693	0.1168	Scale up test with 2 workers
80	538.4	0.1483	0.1483	Message size 2000, with 4 workers
40	498	0.08028	0.0802	4 Connection pool size with 4 workers
80	824	0.0968	0.0966	Scale out with 2 middleware serves
120	937	0.128	0.1289	Scale out with 3 middleware serves
80	337	0.2368	0.236	Stability test with 2 workers

- Analyze results and explain them in detail

We can see that in all the experiments, response time based on interactive law is almost the same as measured response time from clients. So, the data are verified by this law. The basic principle which is applicable to all the experiments and is visible in table data as well, is that when the throughput gets higher, it is direct result of either a lower response time or an increase in the clients (or both of them). For example in row 2, the higher number of clients caused higher throughput and in row 3, the lower response time justified that high throughput. Note that it is possible to have a small number of clients but then for achieving a high throughput, response time must be very low to compensate for the lack of enough clients. Intuitively, we can say that when response time is low, clients can send more requests per second (if they use their full potential), hence, the requests number in the server increases and if server utilization is not close to 1 (which should not be, because otherwise response time would be high), server will respond to those request in a short time. So, it leads to higher throughput.

6. 2_k analysis

For this analysis I chose worker_count, client_number, message_size, factors to inspect. The reason for not picking connection pool size as a factor is that, logically it is irrelevant to client size and message size. Then, I ran an experiment for every combination of those which 2^3=8 experiments. In case of uncertain data, I repeated the experiment. For worker_count I chose 2 and 4 as options. For client_numberI tried with 40 and 80 clients (all of them are in one machine). For message_size I have 200 and 2000 characters as the message size levels. I used throughput as the indicator of performance, but the same analysis can be done with response time as well. By solving the corresponding non-linear regression with a sign table shown below I got the importance of each factor and their interactions.

	msg size	#worker		#clients				throughput
q0	qA	qB	qAB	qC	qAC	qBC	qABC	y
1	-1	-1	1	-1	1	1	-1	340
1	1	-1	-1	-1	-1	1	1	322
1	-1	1	-1	-1	1	-1	1	500
1	1	1	1	-1	-1	-1	-1	470
1	-1	-1	1	1	-1	-1	1	337
1	1	-1	-1	1	1	-1	-1	325
1	-1	1	-1	1	-1	1	-1	528
1	1	1	1	1	1	1	1	530
419	-7.25	88	0.25	11	4.75	11	3.25	

In this table, A is symbol for message size factor, B is for worker_count and C is for client_count. Their interactions are qAB, qBC and so on. The last row is the calculated weight of each column header parameter, so we can write the throughput formula as this:

$$Throughput = 419 + 7.25 * qA + 88 * qB + 0.25 * qAB + 11 * qC + 4.75 * qAC + 11 * qBC + 3.25 * qABC$$

This means that:

- mean of throughput in these experiments is 419
- Worker count in the middleware is the most important positive factor for boosting performance.
- Message size is negatively related to the performance. Bigger message size leads to lower performance, but the effect is not that high compared to other two parameters.
- Client count is positively related to performance, but with a small effect.
- Worker count and message size does not have any interaction with each other. It means they are almost independent.
- Message size and client number show a small interaction, but I believe it's only because of some bad in this experiment. Some of the experiments were done is different time of day.
- Worker count and client count show a considerable positive interaction but it's not bigger than effect of client count alone. So, based on only this interaction we cannot say it's better to increase worker_count along with client_count.
- The interaction between all three factors is positive but very small. So, we can ignore it.
- The effect of worker count is so important that also resulted in positive interaction effects with client count and in qABC.