Advanced Systems Lab (Fall'15) – First Milestone

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

Grading

Section	**Points**
1.1	
1.2	
1.3	
2.1	
2.2	
2.3	
3.1	
3.2	
3.3	
3.4	
3.5	
3.6	
Total	

1 System Description

1.1 Database

PostgreSQL is open-source and free, yet a very powerful relational database management system, adorned with many great and open-source third-party tools for designing, managing and using the management system. The message passing platform I developed utilizes Postgresql 9.3.10 database to store messages and queues persistently.

1.1.1 Schema and Indexes

The system uses two tables to store messages and queues with unique identifier. There is a foreign key qid in message table to keep track of the corresponding queue. The schema provides all characteristics required from the database side. Since detailed information of clients is not required in the project's description, so I didn't register a table for client, which is however easy to extend.s

Table 1 and table 2 shows the schema of two tables, message and queue.

Field	Type	Discription
id	SERIAL	Unique identifier of queue (PK)
time	TIMESTAMP	Time of queue creation
creator	INTEGER	ID of creator

Table 1: Details of Queue Table

Field	Type	Discription
id	SERIAL	Unique identifier of message (PK)
time	TIMESTAMP	Time of message creation
sender	INTEGER	ID of sender
receiver	INTEGER	ID of receiver (*Null* for broadcast message)
message	TEXT	Content of message
qid	INTEGER	ID of the queue it belongs to (FK)

Table 2: Details of Message Table

When it comes to querying from database, indexes are extremely essential in order to maintain good performance. As data grows and even explodes, the performance is tend to becoming slower and slower. The interested user can scan the index relatively quickly, rather than having to read the entire hard disk where all data stores, which would be of advantage to optimise the "ORDER BY" statement.

When we query for the topmost message from a queue, instead of sorting all messages from disk according to the time of creation, multi column index_qid_time [message(qid, time)] allows Postgresql to access messages from a specific queue quickly and get the most recent message from the multi column index. Reading peek message from a given sender and querying queues for a receiver are very similar to querying for the topmost message from a queue. Index_sender_time [message(sender, time)] and index_receiver_qid [message(receiver, qid)] are used to speed up the above two kinds of queries respectively.

1.1.2 Stored Procedures

PostgreSQL offers functions, which are used to move some of the application logic into stored procedures. Generally speaking these interfaces will make system faster, which make database access more closer to the data. I provided 7 stored functions described as below (2 for queue table, 5 for message table), which covers all basic functionality of the messaging system, making it convenient to call directly in Java code.

- create_queue *func* (Clients can create queues)
- delete_queue *func* (Clients can delete queues)
- send_message_broadcast *func* (Clients can send broadcast message to all receivers)
- send_message_to_receiver *func* (Clients can send a message to a queue indicating a particular receiver)
- delete_message_from_queue *func* (Clients can read a queue by removing topmost message)
- peek_message_from_queue *func* (Clients can read a queue by looking at its contents)
- peek_message_from_sender *func* (Clients can query for messages from a particular sender)
- query_queues_for_receiver *func* (Clients can query for queues containing messages for them)

1.1.3 Design decisions

A problem that might arose when dealing with multi threads to access database is that two db clients want to delete messages from the table simultaneously. Deleting a message can be executed as two separate sql statements. The first one read the peek message and the second does the real delete. When two db clients read the same peek message, only one of them, however, could delete it. To solve this problem, mutual exclusion on the database designed needs to be carefully designed. A simple way to fix this bug is to lock critical records. If one tries to delete a message it selects with locks by a "Select ... For Update" statement to make sure the selected record is now exclusively processed by the current transaction. Other clients which try to get a lock on the very same record should wait until the first transaction is completed.

1.1.4 Performance characteristics

In this section, different workloads listed as below were used to measure the performance characteristics of the database. I sent as many requests as possible to the database via different numbers of connections. The database was initialized by 6000 broadcast messages each time before performing the next experiment.

- Raw insert message operation.
- Raw query for queues operation.
- Raw peek message operation.
- Mixed workload: 25% peek, 25% delete, 25% query, 25% insert. This mixed load roughly represents a common behavior in a regular message system.

Since records in the database are limited after initialization, delete operation might not be executed after the table is empty. So I didn't test raw delete operation performance in this section.

For the database, an AWS m3.xlarge instance with 4 cores and 15 GB of RAM was used with high network performance. The experiments run for 5 to 10 minutes each. The raw logs can be found in *logs/perf_db*.

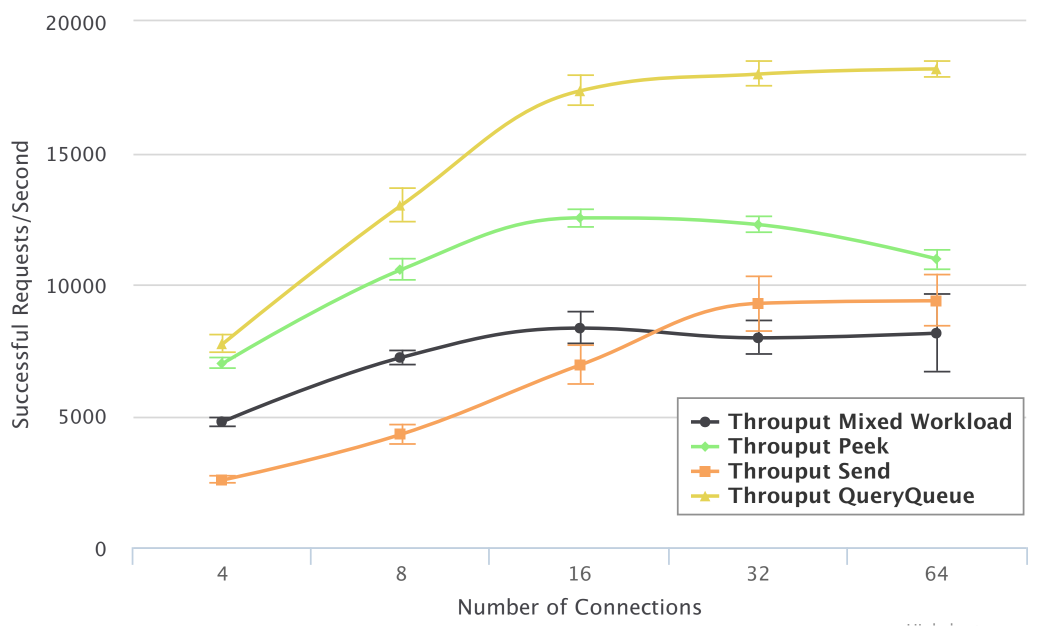

Figure 1: Database throughput with different number of connections

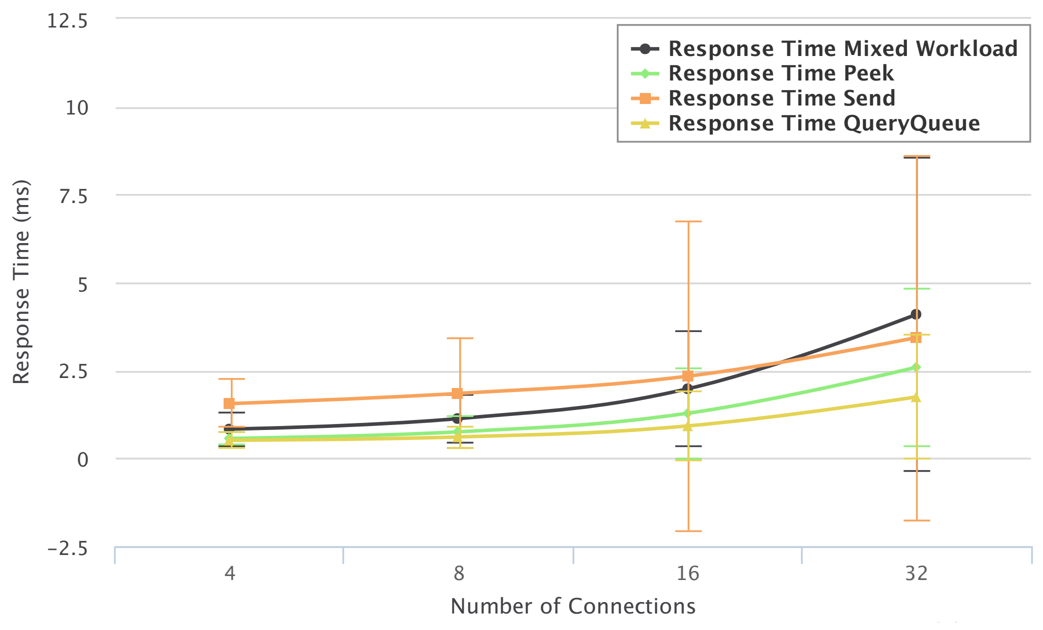

Figure 2: Database response time with different number of connections

Figure 1 and fig. 2 show the throughput and response time for the above different workloads with different number of client connections. It shows that increasing the number of db connections has a negative effect on the response time, which can be explained by the fact that the machine has only four cores available. When using less than eight connections, there is no need to compete for processor cores. As the number of connections increases, they have to compete for the available resources to complete the database access. In addition, the growth rate of the response time of mixed workload is larger than all other pure single operations. This is possibly for the fact that four database ops interact and compete with each other. The standard deviation of the response times of all operations is also relatively high for all measurements compared to throughput. Because throughput is collected per second, however since the absolute value of the response time is very small and surely any difference can cause a high deviation for the experiment runs.

It obviously shows that the maximum throughput of the database varies from different workload, 18000 for pure query queues op, 12500 for peak op, 9400 for send op and 8500 for mixed workload per second. As expected, for insert or delete operations has to be flushed to shared memory or the hard disk, so the database supports more reads than writes. The throughput reaches its maximum at 16 database connections. After that, adding more connections has limited effect on or even decreases the throughput, following the same pattern as the behavior of response time.

Another interesting thing is that the throughput of the mixed work load is smaller than all other single operation workload. There might be two reasons for this situation. First, since delete operation can be regard as two ops: read peek and real delete, which makes it more expansive than send operations. As a result, the size of database grows continuesly, making query operation slower since it takes more time to look up each index block, which in turn suppresses database performance.

1.2 Middleware

1.2.1 Design overview

The middleware side can be seperated into two parts: frontend and backend. Figure 3 depicts the overview of the whole messaging system.

Table 3 shows the difference between JAVA NIO and traditional IO. Java NIO's non-blocking mode enables a thread to request reading data from a channel, and only get what is currently available, or nothing at all, if no data is currently available. Rather than remain blocked until data becomes available for reading, the thread can go on with something else. The same is true for non-blocking writing. The thread can then go on and do something else in the mean time.

Traditional IO	JAVA NIO
Stream oriented	Buffer oriented
Blocking IO	Non blocking IO
	Selectors

Table 3: Difference Between JAVA NIO and IO

For Frontend, I decided to use connection-oriented transaction. Fixed number of frontend workers lie in the ExecutorPool waiting to handle tasks to do the network transmission. These tasks include parsing request from client thread and sending corresponding response to the client. Each client thread owns a socket connection through its lifetime, uses this connection to send all its requests. From middleware side, non-blocking network I/O(JAVA NIO package) was chosen to support a request-oriented architecture to handle connections. After process committed, requests will be pushed into backend task queue waiting for backend workers to make real database access. The source code can be found in package *ethz.asl.mw.frontend.*

On the other hand, Backend ExecutorPool only handles tasks that access database directly, performs db access and puts back the returning response to frontend task queue. The source code can be found in package *ethz.asl.mw.backend.*

1.2.2 Interfacing with clients

Unlike traditional one thread per connection model, Java NIO structure allows threads to spend their idle time on performing IO on other channels when not blocked in IO calls. That is, a single worker thread can now manage multiple channels(client threads) of input and output to achieve maximum performance.

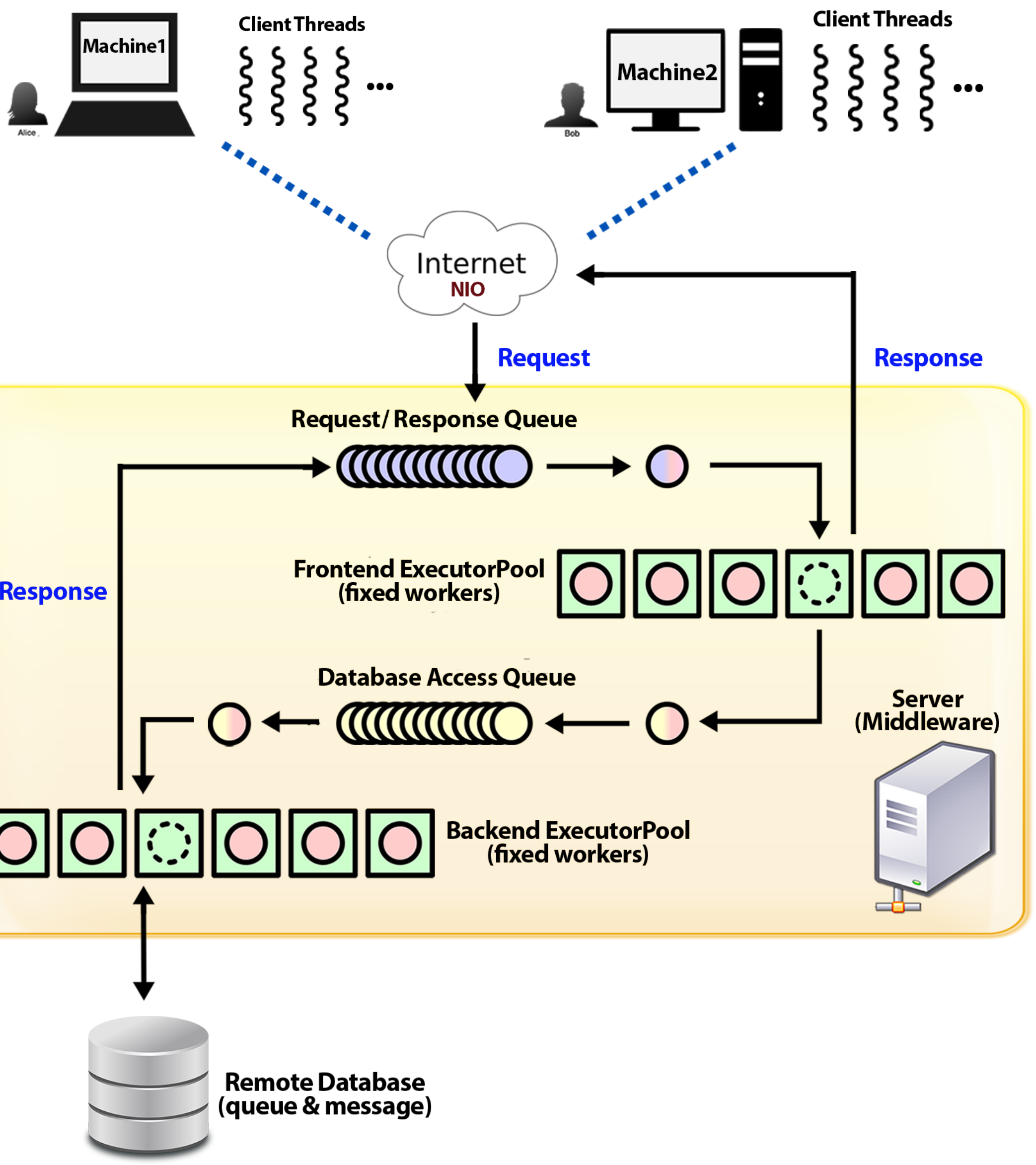

Figure 3: Overview of The Messaging System

After reading incoming data through NIO network, middleware instance is able to create separate frontend tasks to parse requests or responses and send these tasks to the Frontend ExecutorPool by a light-weighted bytes transfer strategy. I created a MessageType Interface to declare different types of requests and responses, shown in listing 1.

Listing 1: Interface of Different Types of Messages

```
public interface MessageType {
        //Message Type
        public static final byte SEND_BROADCAST_REQUEST = 1;
        public static final byte SEND_OK_RESPONSE = 2;
        public static final byte SEND_WRONG_RESPONSE = 3;
        public static final byte SEND_TO_RECEIVER_REQUEST = 4;
        public static final byte PEEK_FROM_QUEUE_REQUEST = 5;
        public static final byte PEEK_FROM_SENDER_REQUEST = 6;
        public static final byte TEXT_RESPONSE = 7;
```

```java
        public static final byte EMPTY_RESPONSE = 8;
        public static final byte DELETE_FROM_QUEUE_REQUEST = 9;
        //Queue Type
        public static final byte CREATE_QUEUE_REQUEST = 10;
        public static final byte CREATE_QUEUE_RESPONSE = 11;
        public static final byte DELETE_QUEUE_REQUEST = 12;
        public static final byte DELETE_QUEUE_RESPONSE = 13;
        public static final byte QUERY_QUEUES_REQUEST = 14;
        public static final byte QUERY_QUEUES_RESPONSE = 15;
        //Test Type
        public static final byte FRONTEND_REQUEST = 16;
        public static final byte FRONTEND_RESPONSE = 17;
        //Invalid Type
        public static final byte INVALID = 0;
}
```

Each request/response packet starts with a byte ID to represent its type which determines the structure of the payload. Then follows the length of the request/response, allowing the middleware to determine the time to finish reading. Finally, it comes to the real content middleware needs for further process, where all fields are represented by JAVA basic types. For instance, considering PeekFromSenderRequest, sender ID and receiver ID are required to be contained in the payload packet. JAVA DataOutputStream and DataInputStream are used to encode and decode the fields in the payload structure for network transmission. After the request generated is completely processed, an response type will be returned to the frontend queue, waiting for a frontier worker to transform it to byte sequence and then transmit back to the corresponding socket channel.

1.2.3 Queuing and Connection pool to database

The database connection pool is implemented also by JAVA ThreadPoolExecutor to queue backend tasks. There're several tasks for different types of request corresponding to the seven stored functions from database side: CreateQueueTask, DeleteQueueTask, PeekFromQueueTask, PeekFromSenderTask, SendTask, DeleteFromQueueTask and QueryQueuesForReceiverTask. Each worker thread creates a database connection once at the time of initialization. This connection is used during its lifetime for all tasks to be executed on this database worker.

1.2.4 Performance characteristics

To measure the performance characteristics of middleware only, we need to exclude the effect of database access. As a result, a special frontend request was introduced to test characteristics of a single middleware instance. The middleware instance only reads the string contained in the payload of request and sends it back directly to the corresponding client with the same text without accessing database.

A middleware instance with 10 frontier worker threads(Amazon m3.large instance) were used was tested for middleware performance using a varying number of clients. Each client sends as many frontend requests as possible. The logs can be found in *logs/perf_mw*.

Figure 4 shows that middleware instance with 10 frontier worker threads is already enough to support 128 clients sending over 50000 request per second with a quite reasonable increase of response time, which is due to the efficiency of communication architecture based on Java NIO so that frontier workers do not have to wait for requests from single client connection. Whenever a request from any client socket is ready to be read, middleware can asynchronously handle it in the frontier queue order. Combined with the above database experiment, we do have reason to believe that database access might be the bottleneck of the messaging system.

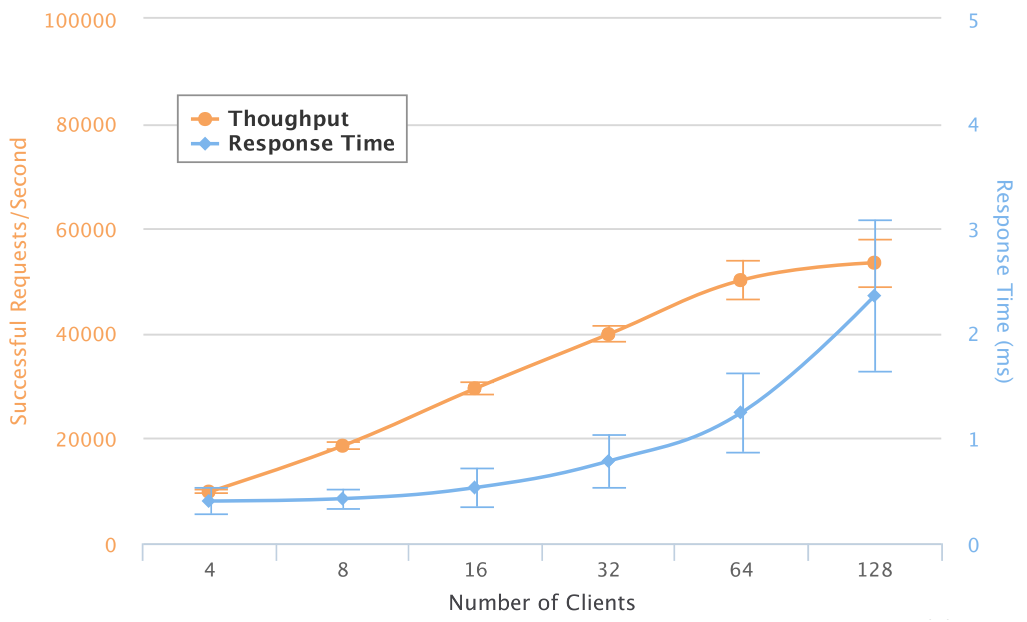

Figure 4: Middleware throughput and response time with different number of clients

1.3 Clients

1.3.1 Design and interface

The client[1] accepts one socket connection and send requests to middleware via this socket channel during its lifetime. Multiple parallel client threads can be started on the same client instance using different command line argument. A message package including all types of requests and responses is shared by client and middleware to handle requests and responses. Since Client.send(Request) requires a Response to be returned before this function ends, the way client make requests can be seen as a blocking network transmission, which is quite different from middleware mentioned above.

To simply activate a client thread without any benchmarking and logging, List 2 shows a simple client interface to send request.

Listing 2: Client Interface

```
//Start Middleware Server
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.socket().bind(new InetSocketAddress(PORT));
new Thread(new Middleware(ssc, new ClientConnFactory())).start();

//Start Client Thread
Client client = new Client(mw_ip, PORT);
client.send(new FrontendRequest("helloworld"));
client.send(new SendRequest(1, 1, qid, "hallo"));
client.send(new PeekFromQueueRequest(1, qid));
client.send(new DeleteFromQueueRequest(1, qid));
client.send(new QueryQueuesForReceiverRequest(1));
```

[1]See *ethz.asl.client.Client.java*

1.3.2 Instrumentation

For the experiments, clients is run in separate threads on a small number of client instance to simulate real user's behavior. Each client thread has its own log file to record the response time of each request sent per second, seperated by a timestamp[2] .

For debugging, Java's logging class was used to track the runtime behavior of each and every steps in a request-response cycle. Each class has its own logger recording whether or not this specific component of the system runs normally under our expectation. If there's any exception or severe feedback in an experiment, the logs for this experiment has to be recollected immediately.

For benchmarking, I designed two different benchmarking strategies for open system and close system both, shown in Table 5. Close system benchmarking is to generate as many requests as possible with a think time after each complete request[3]. The idea of open system benchmarking is inspired by the slides, which is to create a fixed rate workload by Timer.scheduleAtFixedRate()[4] without waiting for the previous request to finish.

1.3.3 Workloads and deployment

Each client thread is capable of sending all types of requests. However, to simplify the logging and benchmarking behavior, I fixed one client thread to send only one type of requests. So workload and benchmarking can be generated by choosing different number of client thread types via command line arguments. To run client threads, m3.large AWS EC2 instances are used with many client threads on each machine. To avoid overloading, if the number of clients increases rapidly, instead of increasing number of client threads on one single machine, total number of client instance should increase in turn. The workload used in the whole evaluation section consisted of four types of operations distributed equally as described in table 4. Actually this workload can represent a type of common use case of a message passing system in real world. At least the number of delete operation is equal to send operation, keeping the size of database as stable as possible.

Operation	Peek	QueryQueue	Send	Delete
	25%	25%	25%	25%

Table 4: Mixed Workload

In addition, it's meaningless if we just want to start client instance and record nothing. We also want to record system behavior(response time and thoughput) at the same time. So deploying client instance is also together with doing experiment on different types of benchmarking strategy. The source code can be found in *ethz.asl.benchmark.**.

1.3.4 Sanity checks

Each client logs the response time of each messaging function. Server also records the micro time of request queue/duration, db queue/duration and response queue/duration with ID of different types of operation[5]. So it's intuitive to check whether the workload from the server side is similar or at least follows the workload from client side. To check the sanity of the generated load, the number of arriving requests(from client to server) and the number of returning responses(from server to client) should be equal.

[2]See *ethz.asl.log.BenchLog.java*

[3]See *ethz.asl.benchmark.MaxRequestGenerator.java*

[4]See *ethz.asl.benchmark.FixRequestGenerator.java*

[5]See *ethz.asl.log.MiddlewareLog.java*

Moreover, as shown in list 1, each client checks whether a valid response is returned for the corresponding request with different ID of response type. Every Server instance and client instance each owns a private performance log started with "server-**.log" or "client-**.log". After experiment, I run *run_log_retrieve.sh* to get all logs from every instance taking part. Only if I find no error in the error logs, then the data collected is thought to be valid.

2 Experimental Setup

2.1 System Configurations

The experiment is performed on the Amazon EC2 with constant load. For the setup configuration, the project used the following specifications. The system consists of a single database instance, a small number of middleware instances(1 to 4) and a number of client instances with parallel client threads. Each Client instance connects to only one middleware instance assuming balanced workload, that is to say each middleware has the same number of client threads in total. All middleware instances operate on the same database instance. The concrete number of middlewares and clients depends on different types of experiment. Detailed configuration of chosen clusters is shown in table 5. Response time and throughput are two key metrics of the system.

Instance	Type	Configuration
Database	m3.xlarge	Ubuntu Server 14.04 LTS — 4 vCPU — 15 GB RAM
Middleware	m3.large	Ubuntu Server 14.04 LTS — 2 vCPU — 7.5 GB RAM
Client	m3.large	Ubuntu Server 14.04 LTS — 2 vCPU — 7.5 GB RAM

Table 5: Configuration of Instance Types on AWS

2.2 Configuration and Deployment mechanisms

Table 6 and table 7 shows some basic configuration parameters and the functionality for each ant task in build.xml.

benchmark	Name of Benchmark
machine	ID of Client/Server Instance
fworkers	Number of Worker Threads in Frontend ExecutorPool
bworkers	Number of Worker Threads in Backend ExecutorPool
parallel	Number of Parallel Client Threads in a Single Client Instance
task	Middware/Database Operation
host	IP of Middleware Instance
think	Think Time (ms)
textlength	Length of Message

Table 6: Definition of Configuration Parameters

2.3 Logging and Benchmarking mechanisms

Logging can be classified into two categories. First one is performance logging. Each Java class in this messaging system uses java.util.logging.* to log system events like errors, warnings, and infos[6]. Second is response time logging[7] to record response time of each successful request for

[6]Example *logs/maximum/perf_logs*

[7]See *ethz.asl.log.BenchLog.java*

Task	Functionality
ant run-middleware	Start middleware server
ant setup-db	Create tables, stored procedures and indexes
ant teardown-db	Drop tables, stored procedures and indexes
ant run-db-initialization	Create initial queues and messages for database
ant run-db-bench-multiple	Database performance test (mixed workload)
ant run-db-bench-single	Database performance test (single operation)
ant run-mw-bench-single	Middleware performance test
ant run-system-bench	All system evaluation tests excluding stability test
ant run-stability-bench	System stability test

Table 7: Ant Tasks

further statistical computation.

Listing 3 shows how to deploy the benchmarking mechanisms of the message passing system on several different machines, which can be found in */scripts*. Each experiment includes the following four steps: sync files, setup database, run middleware, run client. Each step is represented by a function in these scripts.

Listing 3: Deploy System on AWS.

```
//Performance of Database
./run_db_bench.sh key/asl.pem <db-ip> <mw-ips>

//Performance of Middleware
./run_mw_bench_single.sh key/asl.pem <db-ip> <mw-ip> <client-ips>

//Evaluation
./run_mw_stability_bench.sh key/asl.pem <db-ip> <mw-ips>
   <client-ips>
./run_mw_system_max.sh key/asl.pem <db-ip> <mw-ips> <client-ips>
./run_mw_system_scale.sh key/asl.pem <db-ip> <mw-ips>
   <client-ips>
./run_mw_system_rt.sh key/asl.pem <db-ip> <mw-ips> <client-ips>
./run_mw_system_2k.sh key/asl.pem <db-ip> <mw-ips> <client-ips>

//Retrieve Logging from Remote Server
./run_log_retrieve.sh key/asl.pem <mw-ips> <client-ips>
```

Response time was recorded during running time and the throughput was summerized every second. Every experiment was repeated until 95% confidence interval is found. Java's statistic jar package *org.apache.commons.math3* is used to calculate mean value, standard deviation and confidence interval[8].

[8]See *ethz.asl.statistics.**

3 Evaluation

3.1 System Stability

The stability test was performed in 30 minutes using two client instances with 32, 48 or 64 clients(16, 24 or 32 on each machine), two middleware instances(10 frontend workers and 10 backend workers each) and a non-empty database instance. The workload was distributed equally among the middleware instances by connecting to different client instance and taking responsibility only to the threads on individual client machine. Database was initialized by creating 20 queues and inserting 30(NumOfUser) * 20(NumOfQueue) * 10 = 6000 messages.

As the above experiments indicate, the time spent in database access is usually less than 5 ms, which makes each request able to be handled in 10 ms as expected. Every client generates 100 requests per second at a fixed rate, resulting in a load of 3200, 4800 or 6400 requests per second. Since all of them are slightly lower than the maximum throughput in the database load test in section1.1.4, it's expected that these requests can be successfully processed in any individual time interval. The log file can be found in *logs/stability*. For every evaluation sections, I sorted out an excel file *logs/*/plot_*.xlsx* of all statistics to easily check out.

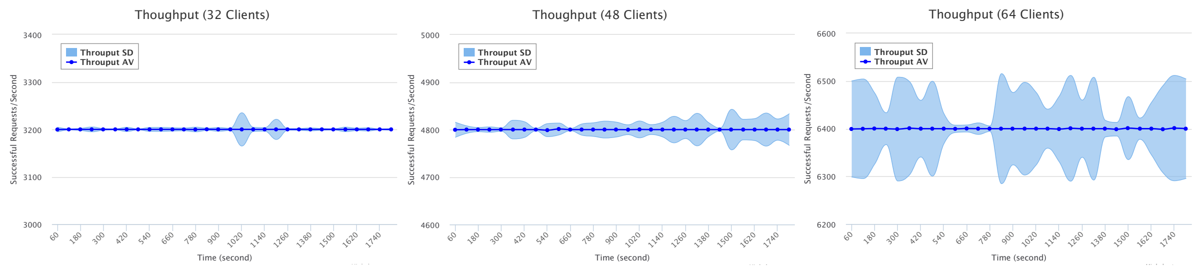

Figure 5: System throughput with mixed workload of 3200, 4800, 6400 messages/second

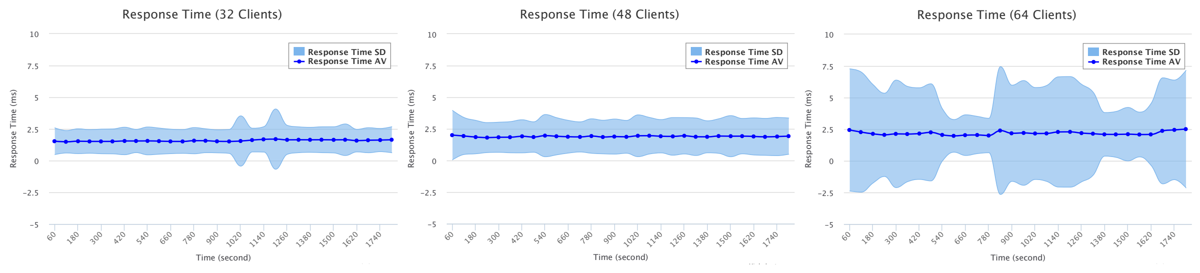

Figure 6: System response time with mixed workload of 3200, 4800, 6400 messages/second

Figure 5 shows the throughput of all different workloads measured by the clients. The data is average over 1 minute interval. It clearly shows that the system's throughput remains stable in all three different workload tests during the whole run. Through there are some small peeks and drops, the average value of successful requests is almost the same as input request without losing any messages, which indicates that the system performs stably and reliably.

Figure 6 shows the response time of all different workloads measured by the clients averaged every minute. Similarly, the average response time remains stable during the whole run, but fluctuates more than the deviation of throughput, following the same rule as the database performance test above.

Another interesting thing is that as the number of clients increases, standard deviation of both response time and throughput becomes larger, which indicates there exists more competition in the messaging because of the increasing queue time. However, this instability is still within a certain range, which is somehow acceptable. The system is ready for further evaluation.

3.2 System Throughput

To measure the maximum throughput of the system, we need to find the appropriate configuration and workload. From the experiments of single database/middleware performance test discussed above, it's inferred that the latency of database access is probably the bottleneck of the system. Hypothetically, the number of client connection which generates the actual message load has great impact on the throughput. As a result, it's reasonable to test the system with different numbers of client threads in order to find the maximum throughput.

According to the database experiments the maximum throughput can be achieved around 16 database connections. For this experiment, I used 2 middleware instances configured as the above run. The throughput is expected to reach over 8000 requests per second for a mixed workload, which is the maximum in the database experiment. The workload follows the same distribution as for the stability and database experiment. Clients send as many requests as possible with a fixed think time.

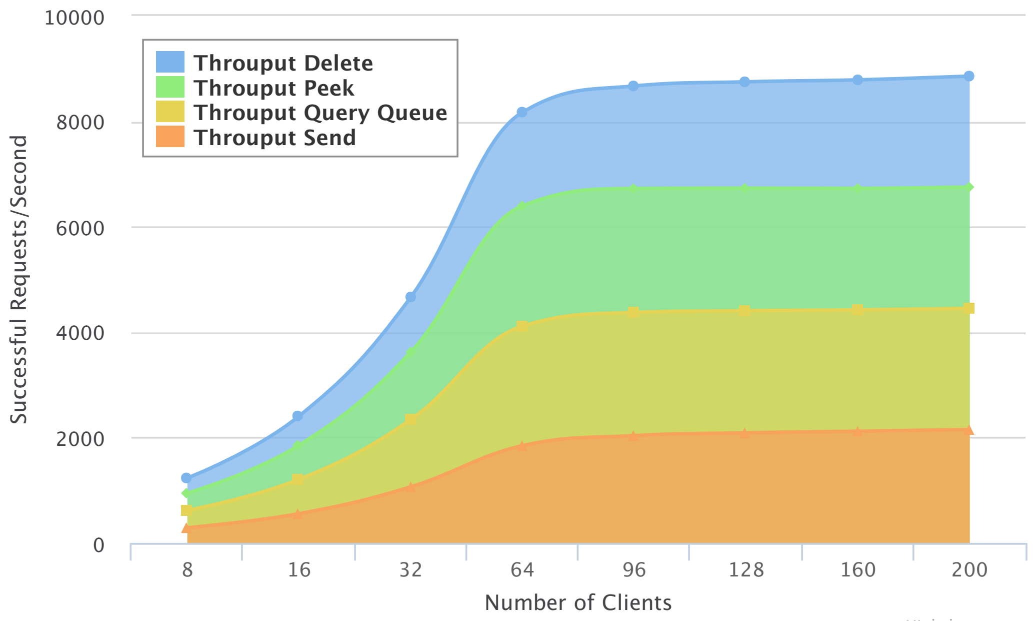

Figure 7: Average throughput of 4 individual operations with different number of clients

Figure 7 and fig. 8 show the average throughput and response time of 4 individual operations for different numbers of client connections. As expected the throughput peaks above eight thousands request per second . For throughput, there exists a dramatic growth before the client number reaches 64, which indicates that 64 clients is already enough to generate enough workload. If the number of clients grows more than 96, the increasing rate becomes mainly negligible, regarded as a waste of resources. In addition, as the number of clients increases, the growth rate of both response time and throughput is somewhat alike for all four operations. To achieve the maximal throughput, at least 64 clients should be used to generate as many requests as possible with at least 16 database workers totally, which is indicated by experiment in section 1.1.4.

Figure 9 shows the micro benchmark (including the time spent in request queue/task, database queue/access, response queue/task when dealing with requests) of single components and units. The proportion of time spent in request and response queue/task is negligible due to the efficient structure of non-blocking IO. It shows clearly that the database queue and duration times are two major part of the time spent in the middleware. Through number of clients increases, the time to do the real database access remains stable after the warmup time. Once database connection pool is saturated and too many clients are sending requests, the time spent

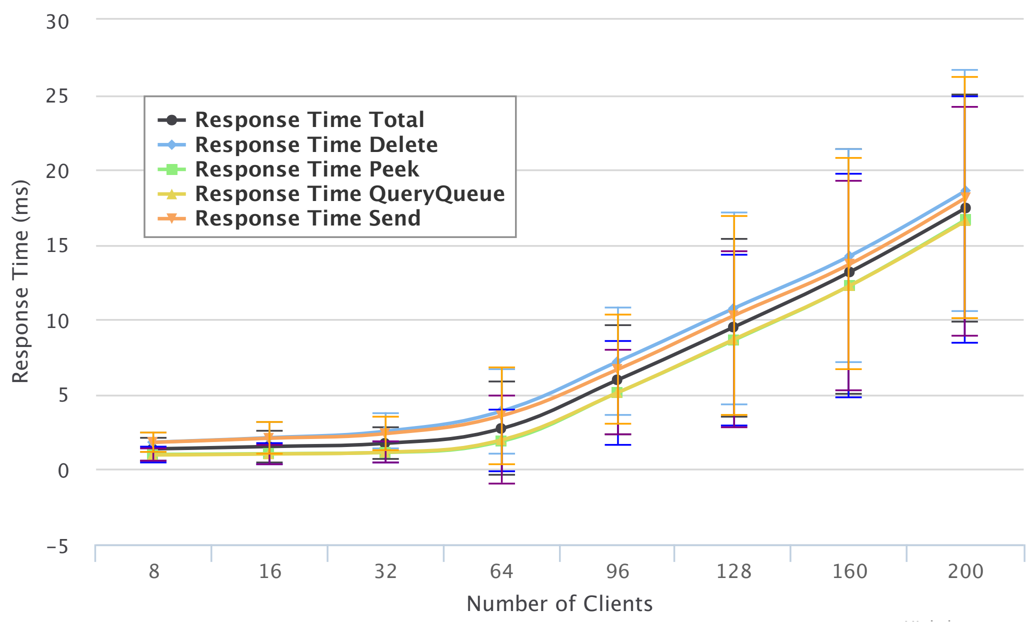

Figure 8: Average response time of 4 individual operations with different number of clients

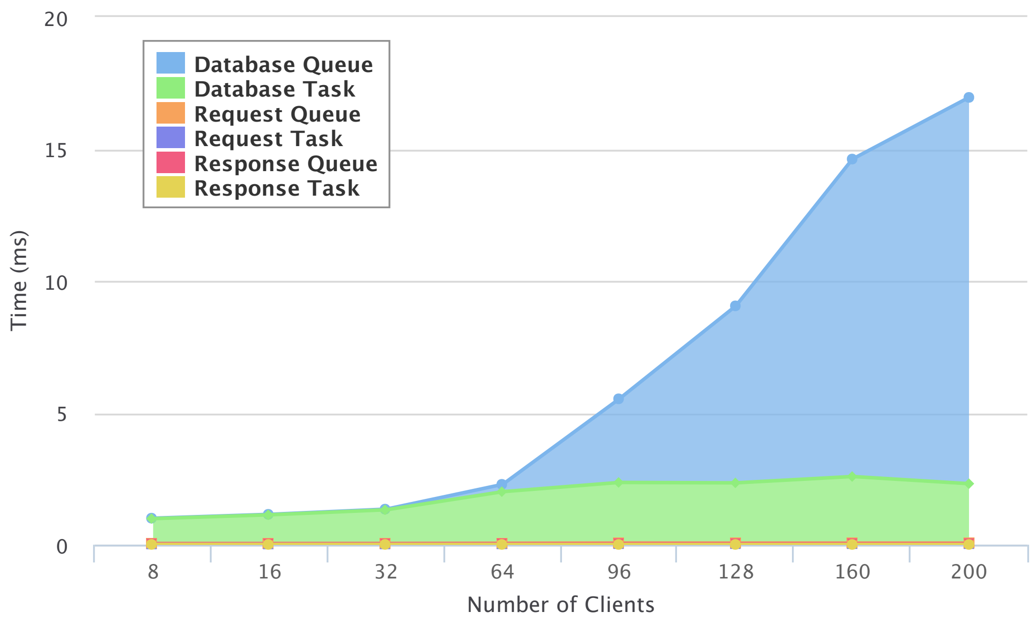

Figure 9: Micro benchmarks of single components and units

in database queue increases drastically, which can clearly explain why the throughput of the system doesn't increase so much as the number of clients reaches 64.

3.3 System Scalability

To determine the scalability of the system is to learn 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, which are able to generate enough load as shown in the above maximum throughput experiment. 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*.

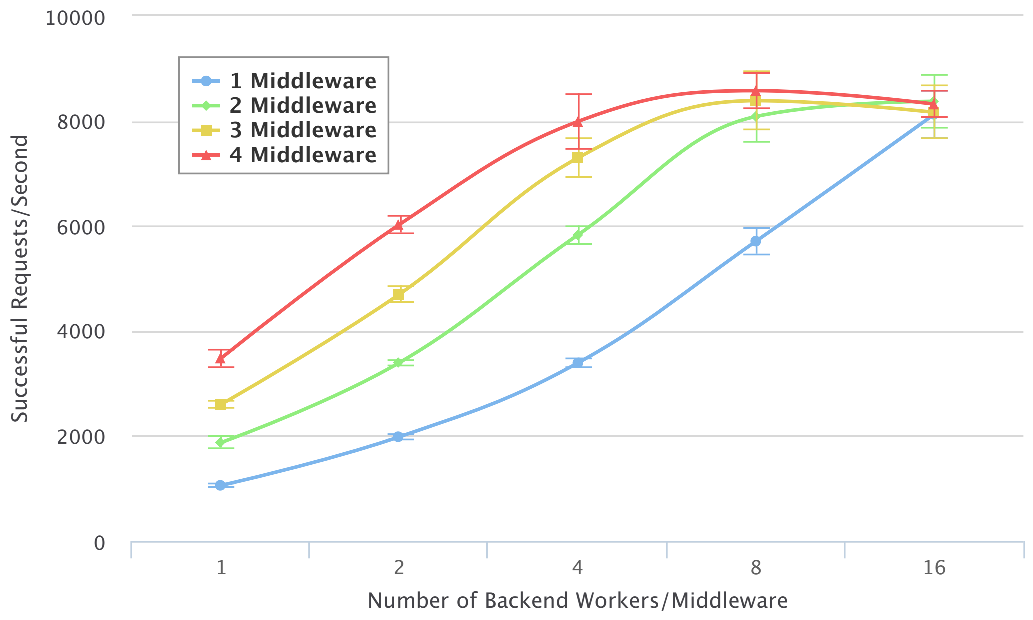

Figure 10: Throughput with different number of backend workers and middleware nodes

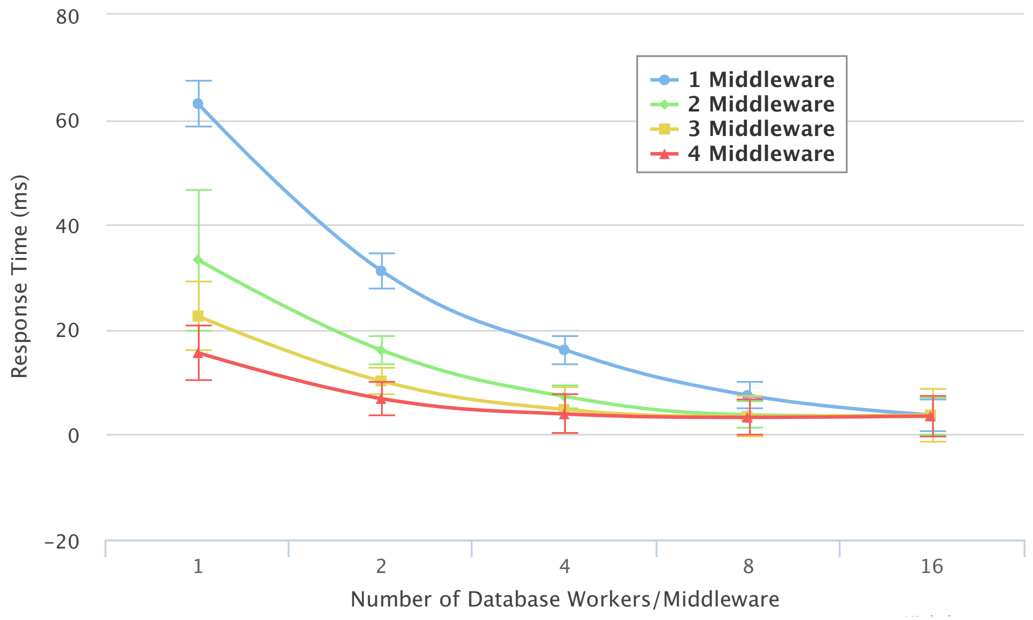

Figure 11: Response time with different number of backend workers and middleware nodes

Figure 10 shows the results of the experiments. The results prove our hypothesis that at least 16 database connections (middleware * backend workers = 1* 16 or 2* 8 or 4 * 4 = 16) are required to saturate the database. So as the number of middleware increases, the minimal number of backend workers to reach maximum throughput decreases accordingly. It's not the number of bworkers or middleware itself but together that really affects the actual throughput and response time. Again, "The more the better" fails for if there're extra bworkers over 16, they in fact can not be fully used and spend most of time waiting for a database request.

When taking a glance at the response time in fig. 11, all four trends achieve the optimal value and keep stable after the database connection reaches 16. When checking micro time logs, fig. 12 shows obviously that the time spent in database queue decreases dramatically while time in database access increases little as the number of bworkers increases with 2 middleware nodes.

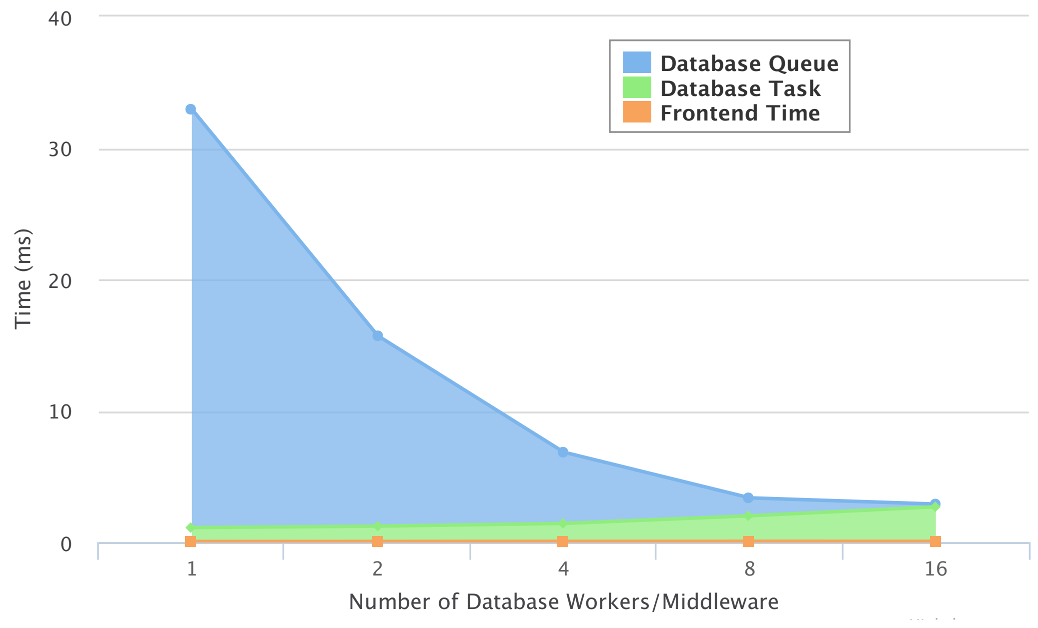

Figure 12: Micro time analysis with different number of bworkers (2 middleware nodes)

3.4 Response Time Variations

As it's already discussed how throughput and response times of the messaging system change with different number of clients in section 3.2, the goal here is to test how response time changes with different sizes of message, various number of middleware nodes or different think time. The configuration of the three experiments is shown in table 8.

Variable	Invariant
Think Time (from 0 to 16 ms)	64 Clients, 2 Middlewares, Message Size 100 chars
Message Size (from 10 to 1000 chars)	64 Clients, 2 Middlewares, Think Time 4 ms
Number of Middlewares (from 1 to 4)	96 Clients, Message Size 100 chars, Think Time 4 ms

Table 8: Configuration of three types of response time experiment

Figure 13 shows how the messaging system reacts with different think time. Both response time and throughput decrease as think time increases. Response time changes little when think time increases from 0 to 4 for backend executor pool is saturated because of the massive incoming requests with small think time. As the sending interval of clients becomes larger, database workers are not busy any more, in turn optimising the time spent in database queue and resulting in less response time. As for throughput, similarly the system becomes unsaturated for long waiting if think time is larger than 4 ms, so that the throughput drops rapidly from 8400 to 4000. Therefore it's reasonable to use 4 ms for the following runs to generate maximal throughput and keep response time as small as possible at the same time.

Figure 14 shows how different message sizes influence response time and throughput. For response time, the average value slightly increases as message size becomes larger, but the change is not obvious. The reason for this kind of stability is that message size does not really have an effect on the whole time spent on processing requests because higher network byte transmission time and frontier time only accounts to a small fraction of the total response time.

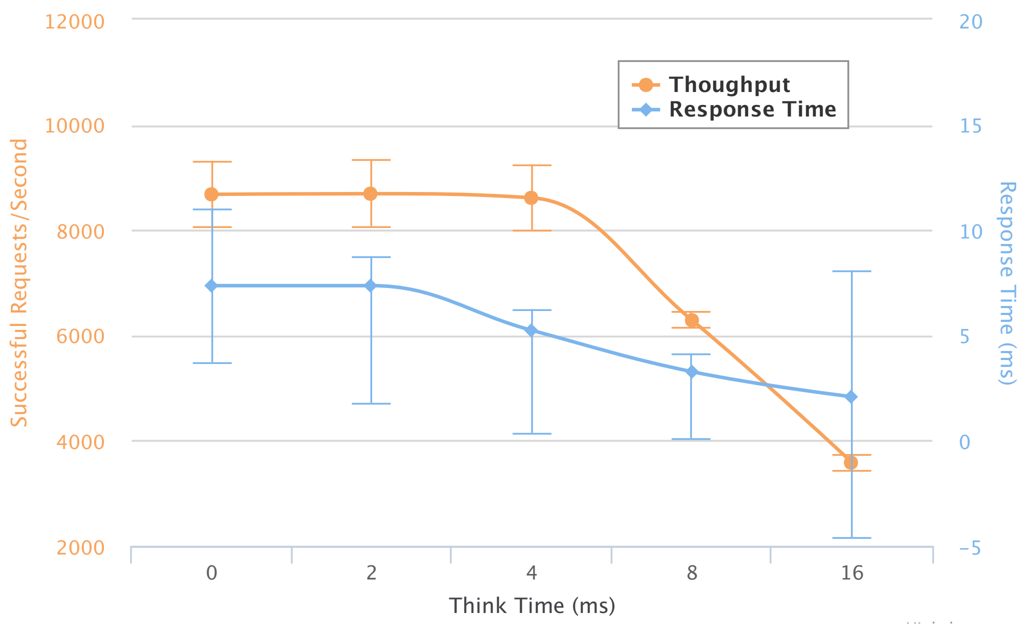

Figure 13: Response time and throughput with different think time

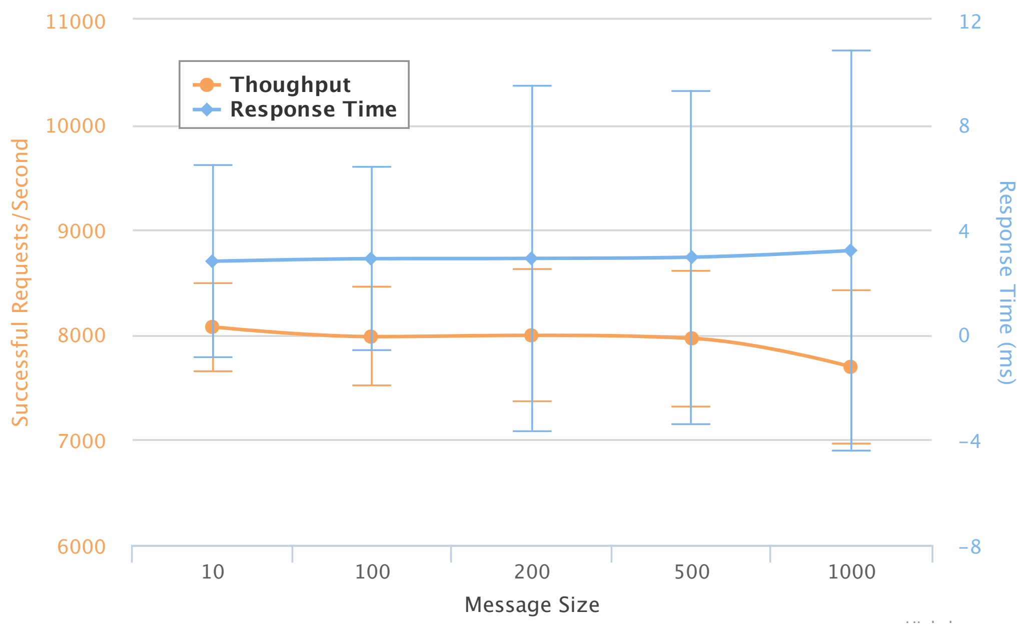

Figure 14: Response time and throughput with different message sizes

The only thing that might affect system's performance is the time spent on database queue and task. When taking into a look at micro benchmark, query and peek operation is rarely affected by the growth of message size. The efficiency of send and delete operation grows and varies more as message size increases where text length really matters. The throughput of the system only observes a negligible decrease because of the small rise in response time.

Figure 15 shows the result of the experiment with different number of middleware nodes. As seen in Section 3.3, it's not middleware itself but together with the number of bworkers that really influence the performance of the system. For response time, the average value drops obviously from 1 middleware to 2 as the database connections are over saturated. After the total number of database connections reaches 16, both response time and throughput remain stable.

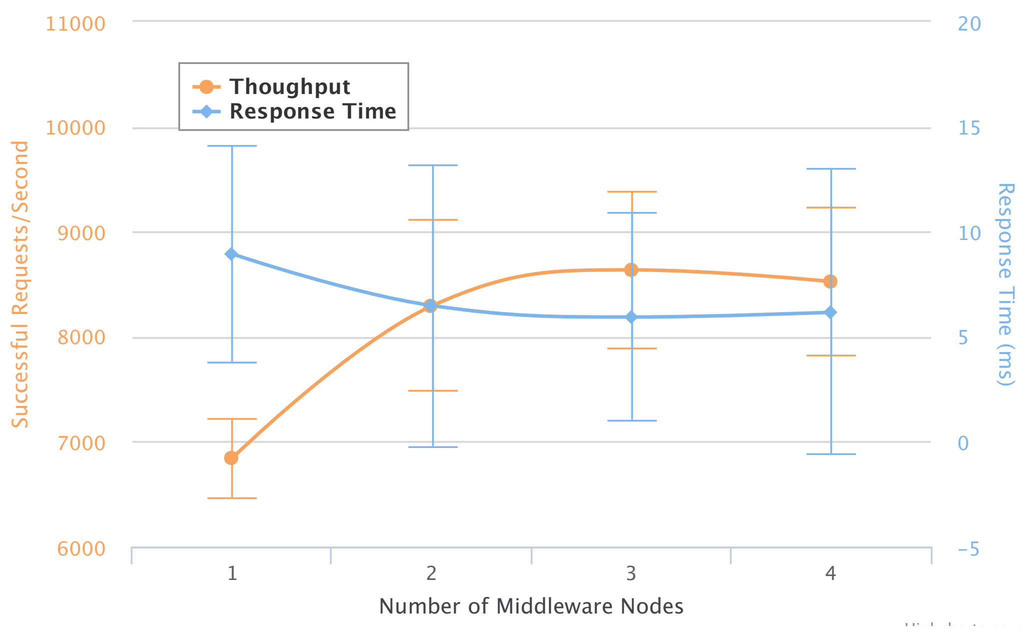

Figure 15: Response time and throughput with different number of middleware nodes

Moreover, the deviation of response time changes little here compared with last experiment for the change in the number of middleware nodes has the same effect on all four single operations.

3.5 2^k Experiment

From the above section, it seems clear that the number of clients and middleware nodes, the size of message are three major factors to affect the system's performance. 2^k experiment aims at exploring non-obvious interactions of these three individual parameters. Table 9 shows feature level signs of throughput and response time for the 2^k factorial designs. Under one's expectation, the response time is supposed to increase with larger message sizes and fewer middleware instances, in contrast to the direction of throughput.

RT	Factor	Level 1	Level -1
A	Client Threads	96	24
B	Middleware Nodes	1	3
C	Message Sizes	1000	10

TP	Factor	Level 1	Level -1
A	Client Threads	96	24
B	Middleware Nodes	3	1
C	Message Sizes	10	1000

Table 9: Feature level signs of throughput and response time for the 2^k factorial designs

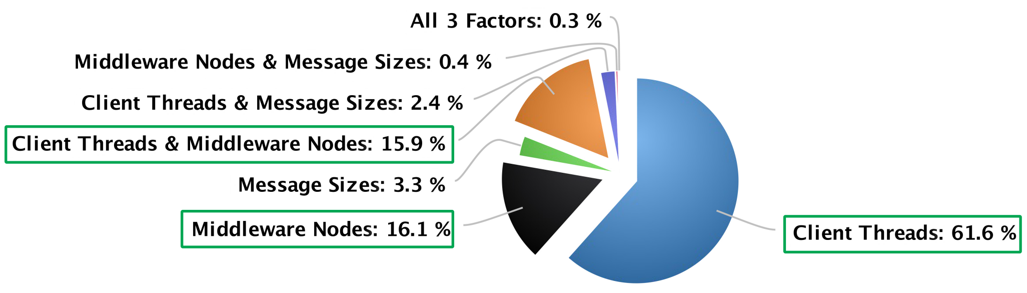

Figure 16: 2^k Factor Weight Percentage

Sign	I	A	B	C	AB	AC	BC	ABC	TP	RT
	1	1	1	1	1	1	1	1	8536.6567	10.1741
	1	1	1	-1	1	-1	-1	-1	8078.0933	9.6066
	1	1	-1	1	-1	1	-1	-1	6525.1867	6.713
	1	1	-1	-1	-1	-1	1	1	6281.6167	6.0935
	1	-1	1	1	-1	-1	1	-1	3469.7	1.9338
	1	-1	1	-1	-1	1	-1	1	3486.2733	1.6958
	1	-1	-1	1	1	-1	-1	1	3520.4267	1.7648
	1	-1	-1	-1	1	1	1	-1	3408.5833	1.7935
WeightTP	5413.32	1942.07	479.36	99.68	472.62	75.86	10.82	42.93		
%		62.18%	15.35%	3.19%	15.13%	2.43%	0.35%	1.37%		
WeightRT	4.97	3.17	0.88	0.17	0.86	0.12	0.03	-0.04		
%		61.06%	16.94%	3.36%	16.59%	2.35%	0.52%	-0.77%		
AVG %		61.62%	16.14%	3.27%	15.86%	2.39%	0.43%	0.30%		

Table 10: 2^k factorial test measurement and result

Table 10 shows that the sign table for 2^k factorial test measurement. Figure 16 clearly proves that the number of client connection has the biggest impact (60%) on the both throughput and response time. The number of client instances is the only source to generate enough workload of the whole system, which results in its major role in performance. The effect of middleware nodes individually can not be ignored which accounts for a 16% of the whole weight. Still as discussed in the last section, message size has a minor influence (3%) on system performance. The reason's already discussed in section 3.4. In addition, the number of clients together with middleware nodes can interact with each other to influence the system.

The number of middleware nodes matters because of the configuration of 2^k experiment. The more middleware instances, the more workers since all middleware used in this run has similar configuration(10 front workers and 10 backend workers). The middleware benchmark in section 1.2.4 shows that 8 front workers can already handle 20000 requests, far more than the system's maximum throughput measured in section 3.2. However, one middleware with 10 bworkers compared with three middleware with 30 bworkers has less ability to efficiently deal with all database connections. As shown in fig. 1, 10 database connections can only handle around 7500 requests, resulting in unsaturation of the system. However, over 8500 requests per second can be processed when the number of connections reaches 16. So in this section, it's the number of database connection rather than middleware nodes that actually leads to the 16% weight on factor B. However, as indicated in section 3.3, if the two configuration both can saturate the system (mw nodes * bworkers >16), the weight of middleware effect is supposed to drop down to a low level.

There exists an obvious interaction between the number of middleware nodes and clients before database connections are saturated. If the two factors change in the same direction, both throughput and response time will increase accordingly, vice versa.

3.6 Conclusion

To conclude, the experiments conducted in this section show that the performance of the system is under my expectation. The factors that might influence system performance, including the number of clients, the number of middleware nodes or workers per node, message size and think time, were carefully analysed with log data and intuitive charts. Java's non-blocking IO allows each middleware instance to handle a large number of requests instead of stucking on one single client connection. The database performs well because of the efficient stored procedure and index design so that the response time doesn't linearly increase as table size of message became larger. There exists many factor that may affect system's performance. The database, however, was still the bottleneck and limited the performance of the whole system even if the optimal values to saturate the whole system of all other factors are chosen.

A number of open problems must be solved to allow the development of a truly general purpose messaging system. These problems suggest a variety of directions that need to be pursued to make such a system feasible. One such direction would be to allow single client thread has the ability to send multiple requests, which is quite easy to modify. The current framework requires that clients should only send one single type of request to easily generate logs and keep track of each operation seperately. However, since in real messaging system clients may want to chat with each other, they should not be limited to send only one type request all the time. Moreover, it would be preferable if there're more database instances for the database itself is the bottleneck of the messaging system. Adding more will surely reduce the pressure of middleware instance and make full use of it. Another possibility would be to play with configuration parameters of Postgresql to better performance. I just changed the value of max_connections, shared_buffers, checkpoint_segments to keep time spent in database access as small as possible. However, careful tuning other parameters, like work_mem, fsync and effective_cache_size would probably increase the system's performance. Finally, though not required in the project description, authenticating a client and storing its identification are essential features realized by almost all modern systems, which can be seen as one direction to expand the functionality of the system.