ASL 2014 – Milestone 1 report

Gustavo Segovia

13-922-463 - gsegovia@student.ethz.ch

Table of Contents

1. Introduction

This document describes what was implemented and done for the ASL course project. It is divided into 16 sections. Basic definitions, design and verifications that are common for all modules of the system are in sections 2 to 7. Section 8 to 13 will contain the design, instrumentation, deployment, experiments and basic initial measurements of each module of the system and the system as a whole. Finally, section 14 and 15 will analyze the results of each experiment and conclude. All figures and graphs in this report can be found in **figures/** for detailed inspection.

2. Implemented Functionality

The Project and Course Description (1) document defines features that the system must implement however leaves some details open to interpretation. Generally, the approach was to try to find the simplest interpretation that fully implements what is described in the document and is functional. The next subsections will describe how the features were implemented without getting into the design details.

2.1. Class attributes

The diagram below represents the main object types of the system and their attributes.

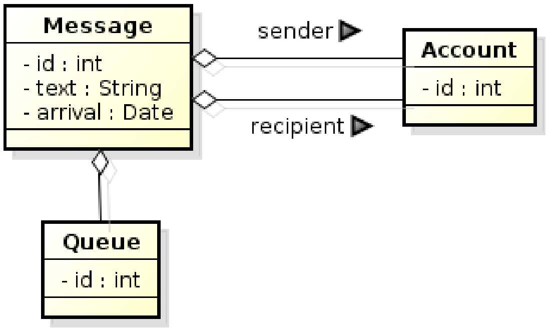


FIGURE 2-1

2.2. Protocol

When a client wishes to make requests to the system, it must first send a "login" request where it identifies itself with its account id. The system checks the id against the allowed account ids of the system and returns either an "ok" or an "account not found" response. If the login is ok, the client is allowed to make any other system request. The client must always wait for the system to return a response for a request to make a new request thus creating a closed system. The client terminates communication by sending a "logout" request.

2.3. Queue Creation

This feature is implemented by a request called "create queue". When a client requests the creation of a queue, the system generates a new numeric id, persists the queue and returns the id. There are no other parameters. The only possible error response is:

- "Failure to create queue" when there is an error creating the queue.

2.4. Queue Removal

This feature is implemented by a request called “delete queue. The client must inform the id of the queue it wishes to remove. When a queue is removed, all messages that were contained in that queue are also removed. The possible error responses are:

- “Queue does not exist” when the specified queue does not exist in the system’s records.
- “Failure to delete queue” when there is an error deleting the queue.

2.5. Send Message

This feature is implemented by a request called “send message”. With this request, a client can place a message on any queue. It can also optionally specify the desired reader by informing its account id, this becomes the message’s “recipient id”. If a reader is specified, only a client with the reader’s account id will be allowed to read the message. The corresponding “recipient id” account does not need to exist at the time the message is sent. The actual contents of the message is text. If the message is successfully placed in a queue, an “ok” response is given. The possible error responses are:

- “Queue does not exist” when the specified queue does not exist in the system’s records.
- “Failure to write” when there is an error adding the message to the queue.

2.6. Read Message

This feature is implemented by a request called “read message”. This request encapsulates all types of read messages requests the system can do by having 3 parameters: “sender id”, “queue id” and “only peek. How input parameters influence the response:

- “Only peek” is a Boolean that determines if the message will be peeked at, not removed from queue after reading, or popped, removed.
- “Sender id”, “queue id” and “reader id” determine which message will be read. At least one, “sender id” or “queue id”, must by specified.
- If “sender id” is null, the oldest message in the queue determined by “queue id” is read.
- If “queue id” is null, the oldest message sent by “sender id” is read from any queue.
- If both “sender id” and “queue id” are defined, the oldest message sent by “sender id” is read from queue with id “queue id”.
- Additionally, any message read must have “recipient id” equal to null or equal to the account id of the client making the request.

The possible error responses are:

- “Queue does not exist” when “queue id” is specified and it does not exist in the system’s records.
- “Queue is empty” when “queue id” is specified and no message can be found that match the criteria specified above.
- “Bad query” when neither “sender id” nor “queue id” where specified.
- “No message matching query” when “queue id” is not specified and no message can be found that match the criteria specified above.
- “Failure to read” when there is an error finding or reading the message.

2.7. Query for Waiting Messages

This feature is implemented by a request called “find queues with messages”. This request takes no parameters and returns the ids of all queues that contain a message that could be read by the current logged in client. What defines if the client could read a message is stated in the section “Read Message” above. The only possible error response is:

- “Failure to find queues” when there is an error in the query for queues.

3. Correctness

For every functionality mentioned above, there is a JUnit test verifying that the system returns the expected response in the normal case and exceptional cases. These 23 tests can be found in **itest/client/SystemTest.java** and they have over 80% coverage on the whole system. To run them, one must have a running database, preferably on localhost.

Each test case clears the database, initializes it and starts a thread to run middleware code and another thread to run client code. Having this setup was very helpful to be able to quickly test every case and to make sure that later modifications did not break anything. The JUnit was only used to check correctness and was developed to run on the local machine so no experiments were conducted with it.

4. Experiment Level Definitions

This section describes the workload used in the conducted experiments.

4.1. Message Size

2 messages sizes were chosen for experiments:

- Small: 200 characters
- Large: 2000 characters

The levels were chosen as suggested by the project description document (1). When used, the messages always have the same content and are always read from memory.

4.2. Dataset Size

The dataset is the amount of data that exists in the applications database table before an experiment starts. 2 levels were chosen:

- Empty dataset: empty message table, 30 queues and 256 accounts.
- Full dataset: Message table with 12.000 messages in 10 different queues from 30 different senders. Half the messages have recipient specified as the sender id and the other half have any recipient. The dataset also contains 30 queues and 256 accounts.

In the use case where messages are immediately read after they are sent, it is reasonable to assume that the message table is near empty. In the use case where a producer sends many messages for a period and then a consumer starts reading, there will be a certain amount of messages in the database. The dataset can be configured to contain small or large messages. The stored procedure `fill_db(TEXT)` is responsible for creating the full dataset. It can be found in **sql/procedures.sql**.

4.3. Workload

One basic set of requests and ordering was determined as the workload for all experiments. This workload is intended to be general and created with the following in mind:

- Use every feature at least once
- Focus on features that probably would be used more in the real world (send and read message)
- Make requests that will not have error responses
- Cause a certain amount of concurrency between clients
- Database should not fill up or empty even if one client is faster than the other

To achieve this, the workload below was created. Consider the id of the client executing the workload equal to "client id":

1. Send message to any recipient on queue 1
2. Read(peek) message from any sender on queue 1
3. Read(pop) message from any sender on queue 1
4. Send message to recipient "client id" on queue 1
5. Read(peek) message from sender "client id" on queue 1
6. Read(pop) message from sender "client id" on queue 1
7. Create queue (consider new queue id equal to "new queue id")
8. Send message to any recipient on queue "new queue id"
9. Find queues with messages
10. Read(peek) message from sender "client id" on any queue
11. Read(pop) message from any sender on queue "new queue id"
12. Delete queue "new queue id"

When the workload is executed, each client has a unique id, logs in and runs this workload in an infinite loop. The workload will only stop if the instrumentation determines that enough samples are collected or if the user shuts the application down. The workload also allows the experiment to define if small or large messages would be used.

Curiously, it was discovered that, in practice, the item "Make requests that will not have error responses" was not guaranteed by the workload due to race conditions. The workload was meant to be set up so that there is always a message for a client to read when it requests a read however at most 1 out of 100 thousand requests return with an "empty queue" error. After careful analysis, a sequence of requests was discovered that could cause the error. This sequence is described in the table below with 2 clients. The numbers in the request column match those in the workload description.

Request	Contents on queue 1 after request. Left to right is oldest to newest, separated by comma
Client A - 1	(msg from A to any)
Client B - 1	(msg from A to any), (msg from B to any)
Client B - 2	(msg from A to any), (msg from B to any)
Client B - 3	(msg from B to any)
Client B - 4	(msg from B to any), (msg from B to B)
Client B - 5	(msg from B to any), (msg from B to B)
Client B - 6	(msg from B to B)
Client A - 2	(msg from B to B) – EMPTY QUEUE ERROR (no message that Client A can read)

TABLE 4-1

In words, this error happens when a client's message is read by another client and then is only left with the other client's messages to read. This is unlikely because one client would have to execute multiple requests before the other one and that is why these errors are so infrequent. When processing the measurements, these errors are simply discarded since their effect on the actual performance is insignificant.

4.4. Instance Types

2 instance types were chosen for the experiments. These descriptions were retrieved from Amazon Web Services documentation (2):

- Baseline: General Purpose instance m3.xlarge with 4 vCPUs, 40 GB SSD drive with 1200 provisioned IOPS and high network performance.
- Large: General Purpose instance m3.2xlarge with 8 vCPUs, 80 GB SSD drive with 2400 provisioned IOPS and high network performance.

The baseline configuration was chosen because it is a good general purpose configuration that could be used for any component of the system. The large was chosen because it had double the computing and database capabilities than baseline.

5. General Building and Running of Java Programs

As required, all Java programs in the project can be built via ant. To do this, simply run `ant dist` at the base directory of the project. This command will not only create a Jar file for each program, but it will also create a zip file containing the Jar plus configuration files. These configurations are used, for example, to determine the database connection configurations that program needs to use.

To run the programs on the local machine, the ant targets run-middleware and run-client were created. When the programs were needed to run on remote machines, bash scripts were used which effectively started the programs with java -jar.

6. Common Instrumentation

6.1. In Code Monitoring

All monitored Java programs implemented, that will be described in detail later on, extend an abstract class called MasterMonitor (found in **src/shared/MasterMonitor.java**). This class is responsible for:

- Persisting the starting time of the Java program and of the experiment to a database table
- Aggregating all measurements of the Java program in memory during an experiment
- Persisting measurements to a database table after enough measurements are obtained
- Terminating the Java program after the end of experiment

Each experiment is saved on 2 tables. The first table, “experiment”, saves timing values and the experiment id. The second table saves the actual measurements. Each subclass of MasterMonitor defines its own measurement table with its own specific columns, this will be discussed later in more detail but some columns are common to all measurements like elapsed time, client id and request type. The fact that all Java programs with the same MasterMonitor subclass save their measurements to one table facilitates processing the measurements afterwards. All tables are defined in **sql/monitor-tables.sql** and can be installed on a database using:

- `ant –Dbuild.properties.file=localhost.properties drop-create-monitor-db`
- `ant –Dbuild.properties.file=localhost.properties drop-create-monitor-tables`

The `experiment` table has the following schema:

- `experiment:`

- `id BIGSERIAL PRIMARY KEY`
- `db_query_start TIMESTAMP default current_timestamp`
- `jvm_startup TIMESTAMP`
- `jvm_query_start TIMESTAMP`
- `start_error`

Persisting the starting time of the Java program and of the experiment is important to be able to synchronize the times between different Java programs on different machines and the time on the database machine so the system's trace could be more accurately portrayed. This is needed because Amazon Web Services does not guarantee that all machine instances will have synchronized time. Also, the Java programs started one after the other, so they do not start at the exact same time. What the MasterMonitor does is it check the JVM's current time and then creates an entry on experiment with default values. This fills the column `db_query_start` automatically with the current time of the database machine. Later fills in the other columns with the JVM times. So effectively `db_query_start` and `jvm_query_start` represent the same time on 2 different machines and allows synchronization. `start_error` is the time it took to create the entry in `experiment divided` by two. It represents the difference between the JVM and database synch times. In practice, this was around 5 milliseconds.

Each thread that has information that is measured in the Java programs possess an implementation of ThreadMonitor (interface defined in **src/shared/ThreadMonitor.java**). The ThreadMonitors create measurements and add them to a thread safe queue in MasterMonitor. Meanwhile, MasterMonitor, with its own thread reads the queue and places the measurements in an array. When the size of the array reaches the intended sample count (which is an input parameter), MasterMonitor saves all measurements in each Java program's appropriate measurement table by the use of batching and then shuts down the application. If the application starts without monitoring, MasterMonitor is never activated and therefore will not shutdown the application at any time.

6.2. JVM Garbage Collection Logs

The JVM Garbage Collection log registers the moment a garbage collection event happens and its duration. This information gives valuable insight on how a Java program behaves during time. To activate this log, a JVM variable is set: `-Xloggc:/path/to/log/file`

6.3. Retrieval Scripts

At the end of every experiment, the measurements and the garbage collection logs are retrieved. To facilitate this process, the scripts **bash/amazonSQLToCSV*.sh** were created. The script is responsible for:

- Transferring the garbage collection logs to local system
- Merging garbage collection logs
- Requesting the remote database to create a CSV(Comma separated values) file of the measurements
- Zipping the CSV file on the remote system
- Transferring the CSV file to local system
- Unzipping the CSV file

The script reads an auxiliary file **bash/amazon-ips.txt** to get the actual public IP of an Amazon machine instance. This file was created because the public IPs of the Amazon machine instances changed every time they were started. To avoid having to modify multiple files every time an instance was started, this file was created. All bash scripts that have to access an Amazon machine instance, including deployment scripts, use this file and they reference it by line number.

The garbage collection logs a merged in a way that logs from machine instances that were running the same Java program are grouped together. An identifier at the end of each line is added to mark where the log line came from. Furthermore, young generation garbage collection events were filtered out and only full garbage collection events were kept. This was done because it was observed that only the latter had a substantial effect on system performance.

The CSV file was created using the Copy command in PostgreSQL. To expedite the transfer of the CSV to the local system, it was zipped. The query that was done to create the CSV file is based on stored procedures in **sql/monitor-procedures.sql**. They were created to order all measurements in synchronized elapsed time. This means that their elapsed time was modified so that it was based on the first Java instance to make an entry in the `experiment` table.

7. Measurements processing

As described in the previous section, all measurements were eventually downloaded to the local system as CSV files. To process these files, python was used. A python module, matplotlib was used to create all of the graphs. All python scripts are contained in the folder **python/**.

In general, 12% of the initial measurements and 5% of the final were always discarded to remove effects like cache warm up or one client terminating before the other have on the performance. These amounts were validated visually by observing each experiment's trace.

Measurements of response time are always done per request, while throughput is calculated by counting the amount of complete requests each second and therefore, throughput measurements have the same amount of samples as the amount of seconds elapsed in the experiment.

Most performance characteristics measured have 95% confidence interval below 1% of the mean. Only measurements done for network performance have a worse confidence interval, but still under 10%.

8. Database

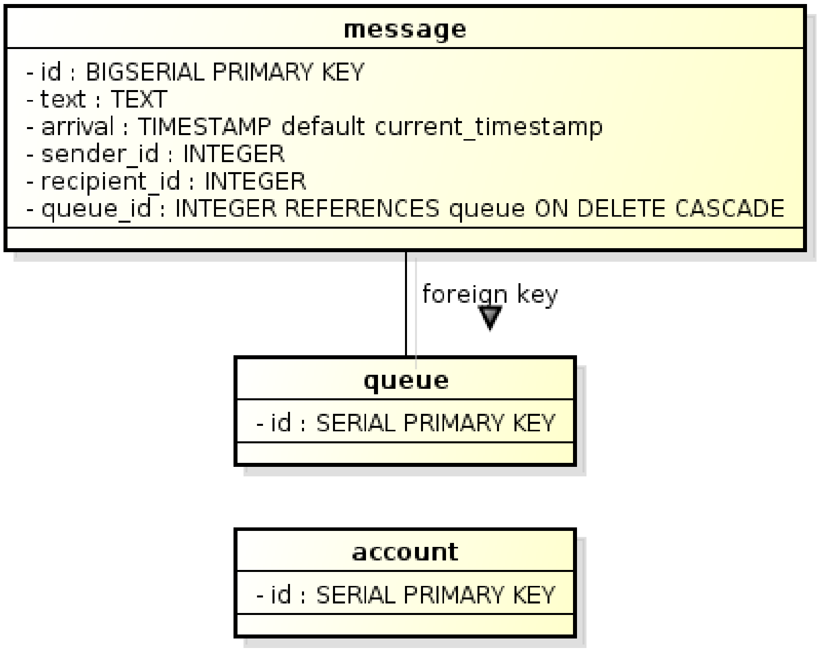
8.1. Design

Schema

The database schema was designed for simplicity, taking into account the features that system needs to implement. It can be found at **sql/create_tables.sql** and can be installed on a database using:

```
-   ant –Dbuild.properties.file=localhost.properties drop-create-db
-   ant –Dbuild.properties.file=localhost.properties drop-create-tables
```

There are 3 tables:


message
- id : BIGSERIAL PRIMARY KEY
- text : TEXT
- arrival : TIMESTAMP default current_timestamp
- sender_id : INTEGER
- recipient_id : INTEGER
- queue_id : INTEGER REFERENCES queue ON DELETE CASCADE
foreign key
queue
- id : SERIAL PRIMARY KEY
account
- id : SERIAL PRIMARY KEY

A foreign key constraint could have been added between `message.sender_id` and `account.id` but it was chosen not to in favor of performance. Since the account table rarely changes, there is no need for database to verify the constraint for every insert/update, the middleware takes care of checking the `sender_id`. The queue table on the other hand might change frequently, hence the foreign key between `message.queue_id` and queue.id. The system allows messages to have `recipient_id` of an account that does not yet exist in the account table so a foreign key between `account.id` and `message.recipient_id` would not make sense.

In PostgreSQL, primary keys are automatically indices (3). Other indices added if they were actually used when the database contained a large dataset. This was done by adding multiple different indices, running EXPLAIN ANALYSE on every SQL query contained in the stored procedures and checking their execution plans. The parameters of each query were also changed since that can change the execution plan. The indices that were actually used are on:

- message.queue_id
- message.sender_id

Stored procedures

For every request type the system can handle a separate stored procedure was created. They can be found in **sql/procedure.sql**. Each stored procedure has an instrumentation component that will be described later in section 8.2. The correctness of each stored procedure was verified manually and by automated JUnit tests in **itest/** as part of the full functionality of the system. Important information for each procedure can be found below.

check_account(INT)

Checks if a certain id exists in the `account` table. Used when a client logs in to the system. Does a simple `SELECT` query.

create_queue()

Used to implement the create queue request. Inserts an entry in the `queue` table with a default generated id.

delete_queue(INT)

Used to implement the delete queue request. Tries to delete an entry in the queue. If no rows are deleted, it infers that the queue did not exist to begin with and returns a "queue does not exist" status. If it is successful, it returns an "ok" status. Since `message.queue_id` references queue and is `ON DELETE CASCADE`, any message that referenced the deleted queue is also deleted automatically.

select_message(INT,INT,INT)

This procedure is used by others to select a specific message. Its inputs are `recipient_id`, `sender_id` and `queue_id` where one of the last 2 might be null. It implements the criteria stated in section 2.6 to find a message. It does this by choosing to execute, depending on the input, 1 out of 3 possible `SELECT` queries.

These queries simply match the input parameters to columns in the `message` table, order the results by id and take the first row. The oldest message is the message with the lowest id, since the column type is BIGSERIAL PRIMARY KEY which means it is unique and monotonically increasing. If the system would ever reach the top limit of BIGSERIAL, the database would throw an error (4) however this is very unlikely since the sequence starts at 1 and it would take the system over 100 thousand years to reach the limit of 2^63 (5) if it had a throughput of 1 million send requests per second.

read_message(INT,INT,INT,BOOLEAN)

Used to implement the read message request. This stored procedure has 2 parts. The first takes care of read message requests that are peek only. To implement this, it uses `select_message(INT,INT,INT)` to select 1 message. If a message is found, it returns its contents. If not, it figures out the reason why no message was returned. If `queue_id` is null, then it simply returns "no message found matching query" code. If `queue_id` is not null, then it checks if the queue exists with a `SELECT`. If it does exist, it returns the "queue is empty" code, if it doesn't then it returns the "queue doesn't exist" code.

The second part takes care of read messages that pop the queue. In this case, the `select_message(INT,INT,INT)` is used to select a message for a `DELETE` statement. This part is especially susceptible to concurrent access and resource contention since two database connections might want to pop the same message at the same time. This part of the stored procedure was implemented to guarantee that for any given number of concurrent connections, it will pop exactly one unique message for each connection, given that there are enough messages, of course.

The implementation is based on the knowledge that, in PostgreSQL's default transaction isolation level (Read Committed) (6), when 2 or more `DELETE` statements that target the same row are executed simultaneously, only 1 will successfully delete it and the others will have no effect. If the result "1 row removed" is returned by the statement, then the DELETE was successful. If "0 rows removed" is returned, then either there was no row selected to begin with or the selected row was deleted by another connection.

To deal with both cases, the procedure creates a loop. First it tries to delete the message selected by `select_message(INT,INT,INT)`, if it is successful, it can break the loop and return the message contents, which were obtained by the same `DELETE` statement through the `RETURNING CLAUSE`. If 0 rows where modified, the procedure tries to determine why. It first checks if the queue exists. If it doesn't, it breaks the loop and returns a "queue doesn't exist" code. If it does exist, then it runs `select_message(INT,INT,INT)` again, this `time` with a count. If count is 0 it breaks the loop returning "no found matching query" code if `queue_id` is null or "queue is empty" code if it is not null. Now, if the count is larger than 0, it means that there is still something that could be deleted (or popped), either because multiple `DELETE` statements conflicted or simply because a new message was added to the queue in the meantime. Either way, the execution loops back to the beginning and repeats until some result is achieved.

This method does guarantee correctness and was tested extensively manually and through JUnit tests in **itest/** however performance can suffer if there is high concurrency. This is because all DELETE statements that happen simultaneously have to wait and see if 1 DELETE statement will actually commit or rollback and then have to try to delete again.

write_message(INT,INT,TEXT,INT)

Used to implement the write message request. This stored procedure simply chooses 1 of 2 inserts, depending on the parameters provided, and executes it. It does not check if the `sender_id` or the `recipient_id` exists in the account table. As mentioned before, the middleware is responsible for verifying the `sender_id` and `recipient_ids` that are not yet in the account table are allowed. The stored procedure does check if a `foreign_key_violation` happens due to `queue_id's` foreign key. If so, it will return the "queue does not exist" error code. If not, it returns the "ok" code.

find_queues_with_message(INT)

Used to implement the find queues with message. This stored procedure does a query on the `message` table based on the criteria stated in section 2.6 to determine which queues contain one or more messages that could be read by the client making the request. The queue ids are returned in ascending order.

8.2. Instrumentation

Stored Procedures

To be able to measure the execution time of each stored procedure without the influence of network or Java procession, an extra output variable was added to every stored procedure that implements a request. This variable contains the value of the time it took to execute the contents of the stored procedure. Basically, a start time is saved in a variable at beginning of every stored procedure and then it is later subtracted by the end time at the end of the function. This information is sent back to middleware where it can be treated similar to the way other measurements are treated.

Other options were considered, like logging the execution times in a separate table or running EXPLAIN ANALYSE on the queries. These methods could be useful, however the first introduces more lag to actual measurement and the latter is not flexible for different experiments, like verifying the performance impact of highly concurrent accesses. Also, information on a simple way to keep the measured times in a global array in pl/pgsql was not easy to find, so a decision was made to send the data back to the middleware. It was considered that attaching the elapsed time to the response of every stored could cause lag but it would be very small since execution time information would only add a few extra bytes to the response that the database already needs to send over the network.

Mocked Stored Procedures

In order to measure the maximum throughput of the connections between a Java program and the database, a set of mock stored procedures were created. The can be found in **sql/mock_procedures_default.sql** and **sql/mock_procedures_large.sql**. Both files contain every stored procedure returning a fixed ok response with the exception of the read_message function where the first file's implementation always returns small messages and second file's implementation always returns large messages. The mock procedures can be installed on a database by running:

- `ant –Dbuild.properties.file=localhost.properties drop-create-db`
- `ant –Dbuild.properties.file=localhost.properties –Dmock.level=[default|large] drop-create-tables-with-mock-procedures`

Java Database Client: DatabaseTester

To call the stored procedures on the database directly with jdbc and without the influence of the code implemented in the Client or Middleware, a minimal Java database client was implemented. Its code can be found in **src/dbtester**. This program allows one to execute the workload defined in 4.3 directly against the database and specify the amount of connections and the message level. Each database connection receives its own client id and executes the workload in loop until the experiment is over. It has an implementation of `MasterMonitor` and `ThreadMonitor`, specified in section 6.1, called `DatabaseTesterMasterMonitor` and `DatabaseTesterMonitor`, respectively. It can save 2 types of measurements, total execution time (that includes java processing, network time and database processing) and statement execution time (time measurement that comes from the stored procedures). These measurements are all saved on the monitor table `database_time`. Its jar (dbTest.jar) is created when running `ant dist`.

8.3. Deployment

The PostgreSQL database was set up manually on the remote computer following the tutorials given in class about the topic. To create the tables, the ant commands mentioned in sections 8.1 and either 6.1 or 8.2, depending on the experiment, were used.

All isolated database tests used the DatabaseTester. **bash/amazonDatabaseTestDeploy.sh** was used to install DatabaseTester on the machine, **bash/amazonDatabaseTestStart.sh** to start the experiments and **bash/amazonSQLToCSV_db.sh** to retrieve the results.

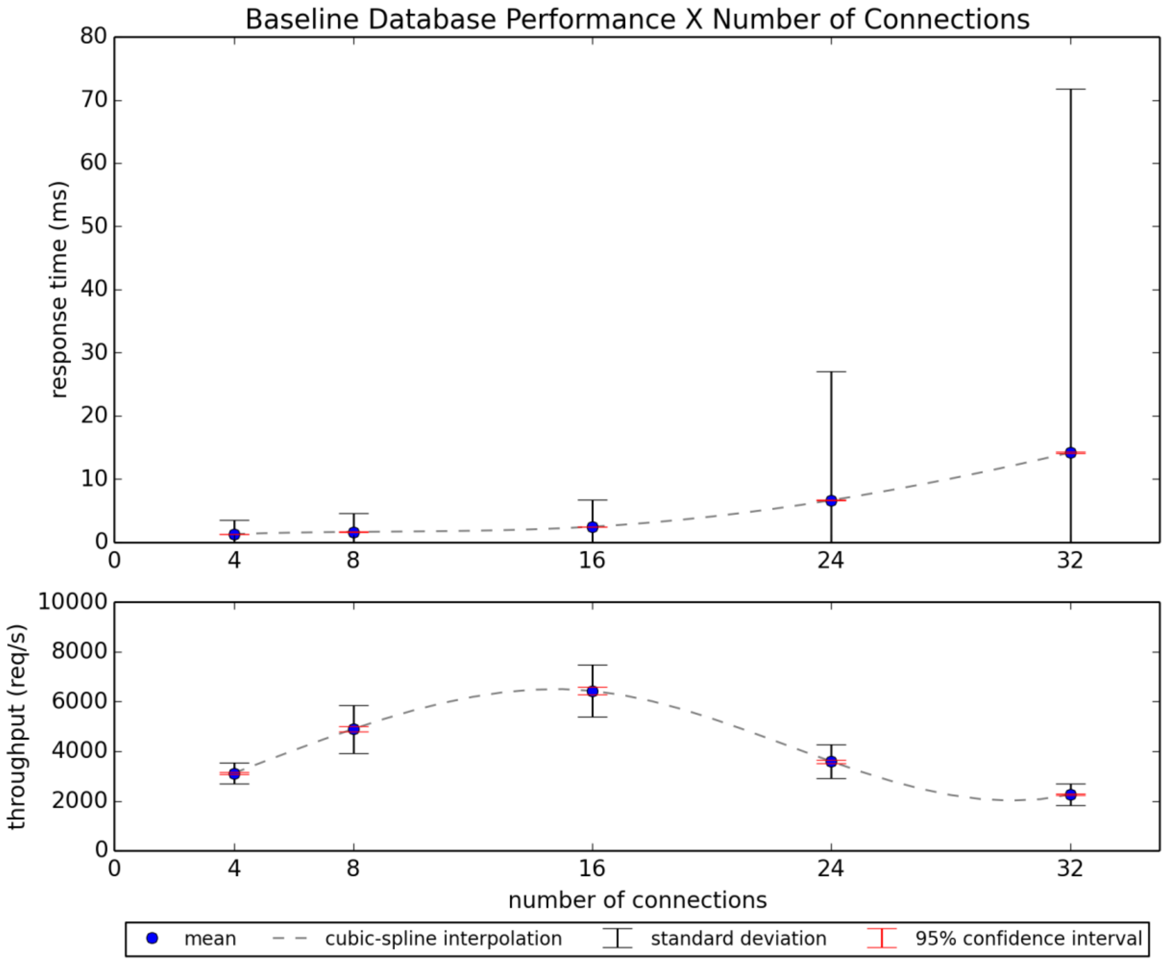
8.4. Performance Characteristics

4 experiments were done to evaluate the performance characteristics of the database, 2 to determine the database performance without the influence of the network and 2 to determine the database network performance. Other experiments, done on the system as a whole, also determined database performance characteristics. They will be explained later in the document. What was expected form the experiment and analysis will be done in 14.1 and 14.2. The maximum amount of connections PostgreSQL with default configuration is around 100.

Database Performance

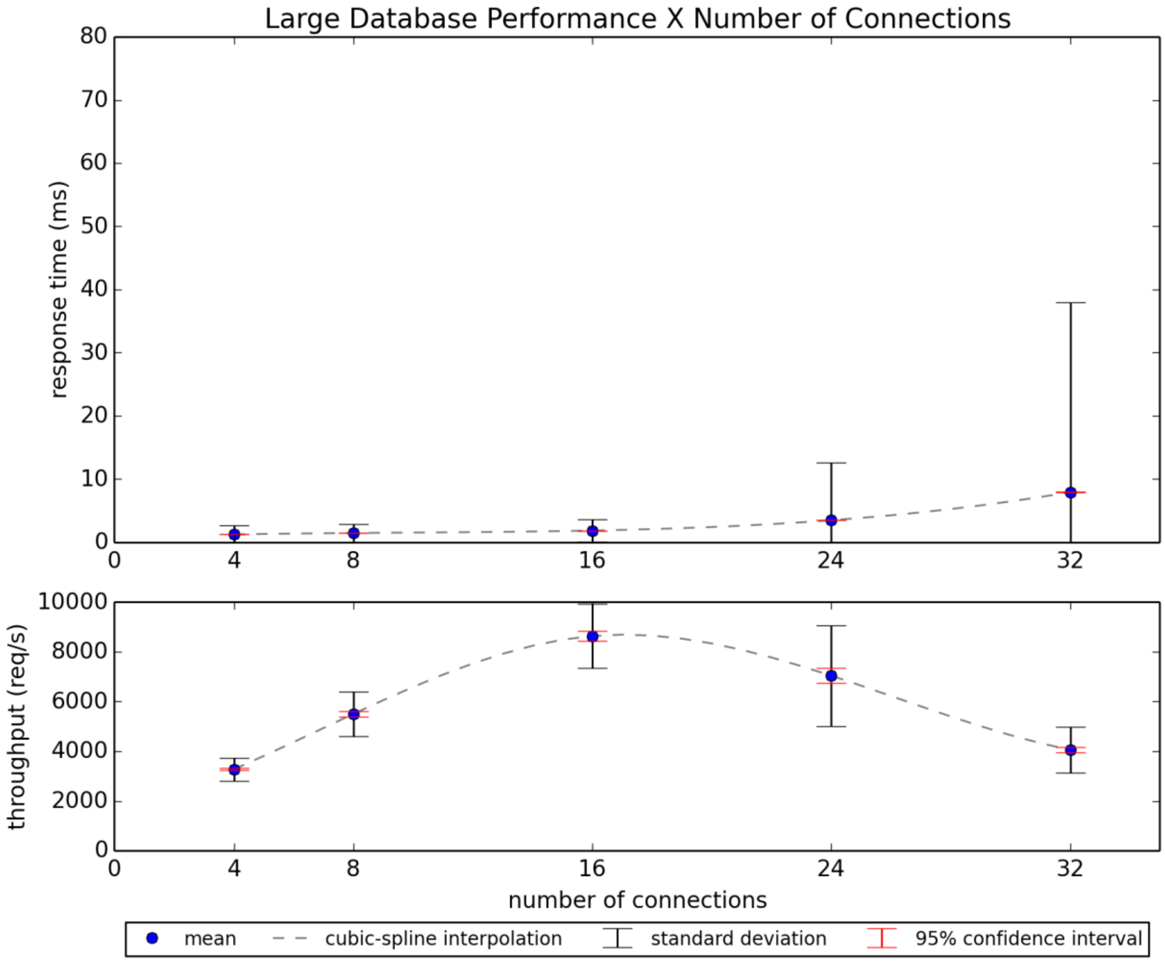
The following graphs show the database performance when running DatabaseTester on the same machine as PostgreSQL. They were put on the same machine with the intent of making performance measurements in relation to the number of connections, without the influence of network communication. The first experiment was done with the database running on a baseline amazon instance while the second ran on a large instance. In both cases, the workload defined in 4.3 was used with small messages and the stored procedures are not mocked.

python/response_time_throughput_consolidator.py was used to consolidate the measurement values located in **measurements.zip/database_perf**. **python/connections_graph.py** was used to create the graphs.



GRAPH 8-1

In Graph 8-1 we can see that throughput on the baseline machine peaks at 16 connections at 6429 requests per seconds with each one taking 2.5 milliseconds in average. Past that amount of connection, response time and its standard deviation grows rapidly while throughput drops.



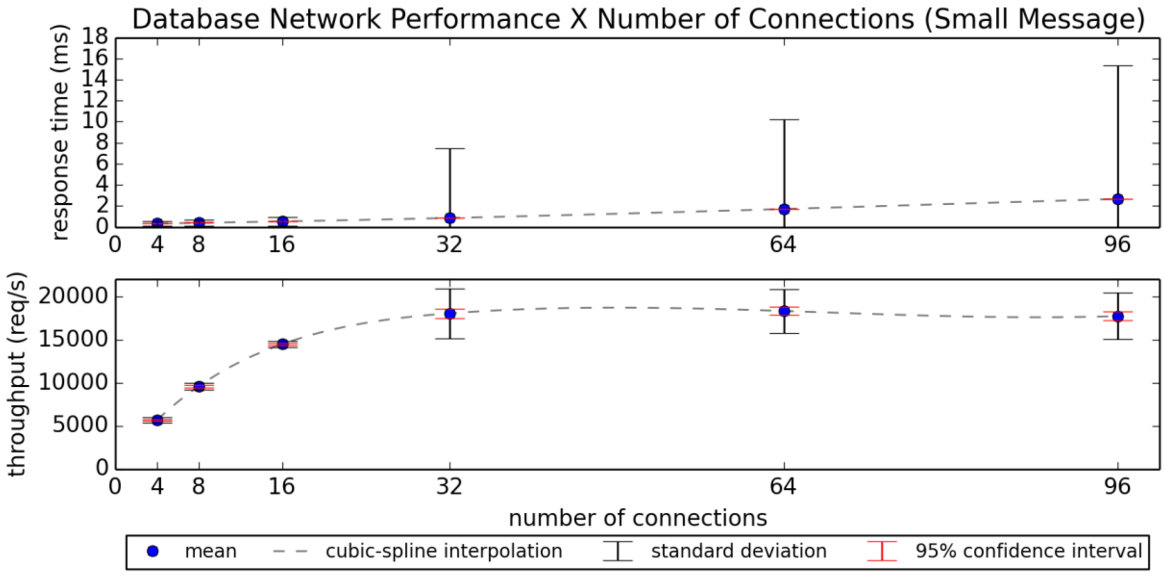
GRAPH 8-2

Graph 8-2 was put in the same scale as Graph 8-1 to facilitate comparisons. We observe the same general behavior in both graphs. Throughput peaks at 16 connections, having 8639 requests per second with each one taking 1.8 milliseconds in average. Using a large instance gave the database a 38% increase in speed a 34% increase in scale at 16 database connections.

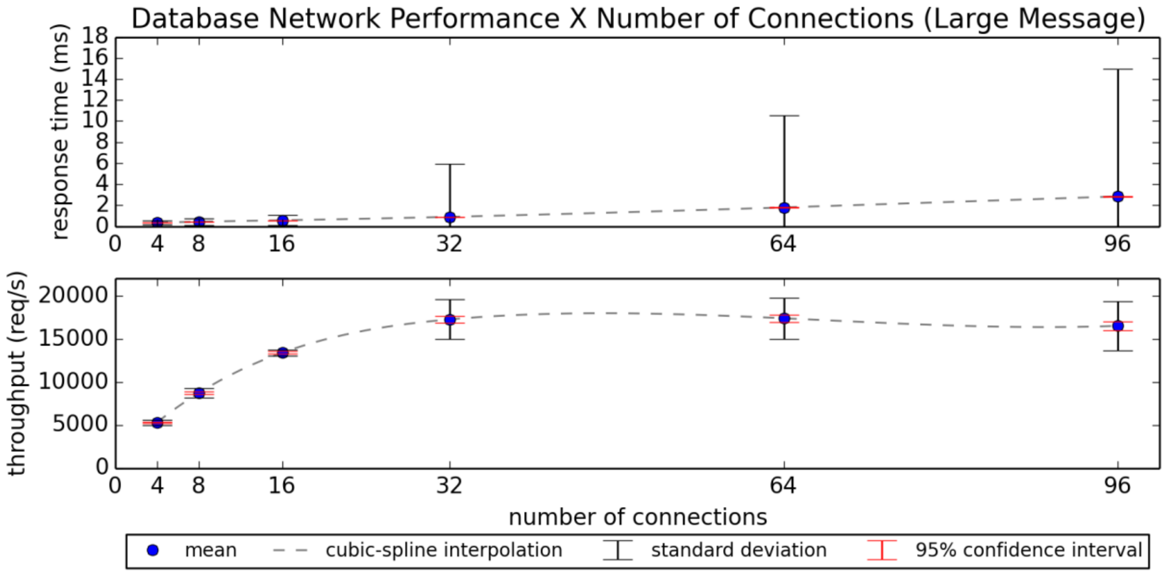
Database Network Performance

The following graphs show the database network performance when running DatabaseTester on a different machine than PostgreSQL. The intent is measuring the network performance in relation to the number of connections and figuring out if it is in any way a bottleneck, so the database uses mocked stored procedures causing their response time to be minimal. This "network" performance time includes the time needed for JDBC to make a request, time for the network to transfer to the request to the database, time for the database to interpret the request and time for the response to go all the way back through the same steps to the Java function call. The first experiment was done small messages and the second with large. In both cases, the workload defined in 4.3 was used.

python/response_time_throughput_consolidator.py was use to consolidate the measurement values located in **measurements.zip/database_network**. **python/connections_graph.py** was used to create the graphs.



GRAPH 8-3



GRAPH 8-4

Graph 8-3 and Graph 8-4 were made with the same scale and have very similar general behavior. Both experiments have near maximal output with 32 connections, a throughput of 18.064 request per second at 0.8 milliseconds per request for small messages and a throughput of 17.267 request per second at 0.9 milliseconds per request for large messages. For more connections, throughput plateaus while response time and it standard deviation grows.

9. Middleware

9.1. Communication between Client and Middleware

Communication between client and middleware is done with TCP through sockets. To represent the types of request and response communication that can be done through the sockets, DTO (Data Transfer Objects) classes were created. They can be found in **src/shared.dto/** and there is one for every functionality stated in section 2.

When one wants to serialize a DTO object, he needs only to call `serialize()` on the instance. This method returns a String. Each class is responsible to implement this method in manner that all of the object attributes are encoded in the returned String. For deserialization, there is a class called `MessageDeserializer` in **src/shared/**, that is responsible for reading a String, determining which DTO class it represents and instantiating that class.

String was chosen for the serialized form of the DTOS because it is succinct, fast to serialize/deserialize and it facilitates the design of the socket readers and writers, but this also leads to a complication. Some DTOs' serialized String must contain message text which may contain any characters, however some characters have special meaning for the socket reader and `MessageDeserializer`. The newline character '\n' is used to determine when one message ends and the other starts by the readers while the pipe character '|' is used to separate attributes in a serialized String. To solve the problem, the super class of the DTOs, `DataTransferObject`, contain methods that can escape and unescape the message text when needed.

All methods related to DTOs are verified with automated JUnit tests in **src/test**.

The classes `MessageWriter` and `MessageReader` are responsible for taking a DTO to write it to an `OutputStream` and reading an `InputStream` to instantiate a DTO, respectively. They assume that what will be written or read are always in the form of String and separated with newline characters.

9.2. Design

As with the other components in the system, the design for middleware is intended to be simple yet implement all features in an efficient way. Code for the middleware component can be found in **src/middleware**. `MiddlewareSever` is the class that contains the "main" method that starts the component. It is responsible for reading and setting up configuration parameters like maximum number of database connections and also creating a `ServerSocket` to handle incoming connection requests. For each new connection/client, a `RequestHandler` class is instantiated and given a thread from an unbounded cached thread pool to handle all incoming requests form a client. The thread only terminates when the client logs off. When the application shuts down, it takes care to close all socket and database connections as well as terminate all threads.

Using `ServerSocketChannel` was considered for some time but it was ultimately discarded since setup is a bit more complicated and `ServerSocket` works perfectly fine for the expected amount of clients that the system will have.

The `RequestHandler` is responsible for reading incoming requests from the input stream with `MessageReader`, acquiring a connection from `MiddlewareDBConnectonPool`, calling the appropriate method on `MiddlewareDBConnecton` and writing the response to the output stream with `MessageWriter.`

The `MiddlewareDBConnecton` is responsible for taking any DTO request, calling the appropriate stored procedure on the database and returning the response in DTO format. When it is initiated it creates a new database connection so only the `MiddlewareDBConnectonPool` may create an instance. `MiddlewareDBConnectonPool` controls the total amount of connections allowed to the database. When `acquireConnection()` is called, if there are any unused available `MiddlewareDBConnecton` instances, one is returned. If not, one is created if there the maximum number of database connections have not yet been reached. If the max connections have been reached and there are none available, `acquireConnection()` blocks execution until one becomes available/is released. After usage of the MiddlewareDBConnecton instance, it must be released by calling releaseConnection(connection). The blocking behavior is implemented with a java.util.concurrent.ArrayBlockingQueue and it set to be fair (first come first serve). Acquiring a connection is equivalent to calling take() on the queue and releasing or creating a connection is equivalent to calling add().

Some configuration parameters like database connection information is read from .properties files that are passed in as parameters for the middleware application. Examples of these .properties files can be found in **config/**.

9.3. Instrumentation

Middleware performance

Instrumentation on the middleware allows to save in memory every request made. It later stores the measurements on database tables when the experiment is over. It does this with implementations of MasterMonitor and ThreadMonitor, specified in section 6.1, called MiddlewareMasterMonitor and RequestHandlerMonitor, respectively. Each RequestHandler thread has its own RequestHandlerMonitor and all measurements are aggregated at MiddlewareMasterMonitor. The following information is saved from each request:

- Client id.
- Elapsed time: time since the beginning of the experiment.
- Response time: the time it takes to start writing the response of a request after finished reading it.
- Request type.
- Response type.
- Serialization time: time it took to serialize the response.
- Deserialization time: time it took to deserialize the request.
- Acquire database connection time: time it takes to acquire a database connection from the pool.
- Release database connection time: time it takes to release a database connection to the pool.
- Statement execution time: time measured in the stored procedures informing how much time the contents of the stored procedure took to execute.
- Database network time: time it took to run a JDBC statement and read the response minus the statement execution time.

These measurements are all saved on the monitor table middle_time.

Middleware Mock

The middleware is also capable of running in mock mode. When in this mode, analogous to the mocked stored procedures in section 8.2, it returns fixed predefined responses for different requests and it is intended to measure the performance of the communication between the client and middleware.

9.4. Deployment

bash/amazonMiddlewareDeploy.sh was used to install the middleware on multiple machines and **bash/amazonMiddlewareStart.sh** to start the experiments on these machines. **bash/amazonSQLToCSV.sh** was used to retrieve the results.

9.5. Performance Characteristics

No experiments were done the specifically isolated the middleware however many measurements were made on it when the system was tested as a whole. Performance characteristic of the middleware strongly depend on the configuration of the system and will be discussed in section 13.

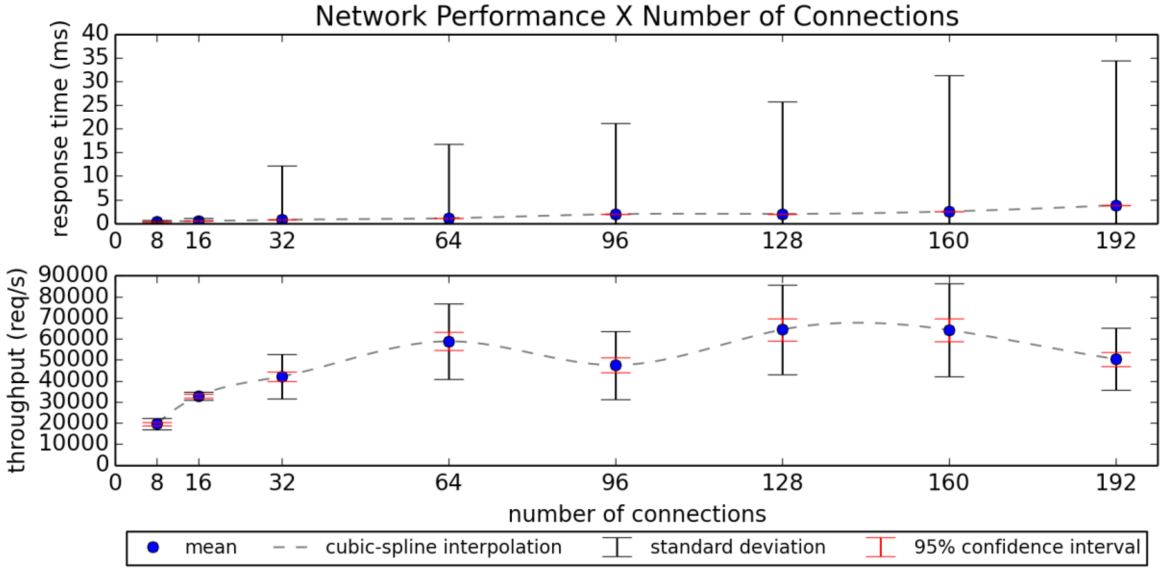
The network communication on the other had was tested in isolation.

The maximum amount of connections each middleware allows is limited by memory. The baseline machine was able to support up to 20.000 connections.

Client x Middleware Network Performance

This experiment is intended to measure the communication performance between client and middleware on 2 different machines as a function of the number of socket connections and also to determine if the network is in any way a bottleneck. The measurement shown here is the time it takes for requests from the client to be written into the output stream, be processed by the mock middleware and their responses to be read from the input stream. Scripts described in section 9.4 were used for the middleware and scripts that will be described in section 10.3 were used for the client. This experiment was run on 2 baseline machines, one running the client and one running the mock middleware. The workload described in section 4.3 was used with small messages.

.python/response_time_throughput_consolidator.py was used to consolidate the measurement values located in **measurements.zip/network**. **python/connections_graph.py** was used to create the graphs.



GRAPH 9-1

In Graph 9-1 the throughput grows consistently from 8 until 64 connections and then it varies between 45.000 and 65.000 requests per second. Maximum throughput was observed at 128 connections with 64.409 requests per second at a 2 millisecond response time. Response time consistently grows with the number of connections. Minimum response time was measure at 0.4 milliseconds (8 connections) and maximum at 3.8 milliseconds (192 connections). There is a sudden jump in standard deviation from 16 connections to 32 connections that will be discussed in the analysis in section 14.2.

10. Client

10.1. Design

As with the other components in the system, the design for the client is intended to be simple yet implement all features in an efficient way. Code for the client component can be found in **src/client**. `ClientWorkloadPool` is the class that contains the "main" method that starts the component. It is responsible for reading and setting up configuration parameters like the connection details of the middleware and for starting the ClientWorkload threads.

When the `ClientWorkloadPool` is started, you inform which range of client ids it will use. For each id in range, a new `ClientWorkload` is created and given a thread so it can open a socket connection with the middleware and communicate with it. This means that for each client java instance there are multiple client accounts running at the same time, each with its own thread and socket connection.

The requests that the `ClientWorkloads` make are defined in DefaultWorkload and match the workload defined in section 4.3 and can be configured to send small or large messages. Some configuration parameters like middleware connection port is read from .properties files that are passed in as parameters for the client application. Examples of these .properties files can be found in **config/**.

10.2. Instrumentation

Instrumentation on the client allows to save in memory every request made. It later stores the measurements on database tables when the experiment is over. It does this with implementations of `MasterMonitor` and `ThreadMonitor`, specified in section 6.1, called `ClientMasterMonitor` and `ClientRunnableMonitor`, respectively. Each `ClientWorkload` thread has its own `ClientRunnableMonitor` and all measurements are aggregated at `ClientMasterMonitor`. The following information is saved from each request:

- Client id.
- Elapsed time: time since the beginning of the experiment.
- Response time: the time it takes to get the response of a request.
- Request type.
- Response type.
- Serialization time: time it took to serialize the request.
- Deserialization time: time it took to deserialize the response.
- Network time: response time minus serialization/deserialization times.

These measurements are all saved on the monitor table client`_time`.

10.3. Deployment

bash/amazonClientDeploy.sh was used to install the middleware on multiple machines and **bash/amazonClientStart.sh** to start the experiments on these machines. **bash/amazonSQLToCSV.sh** was used to retrieve the results.

10.4. Performance Characteristics

No experiments were done the specifically isolated the client however many measurements were made on it when the system was tested as a whole. Generally the think time on the client was very small, less than 1% of the request time as will be seen in section 14.

11. System Design

11.1. Setup

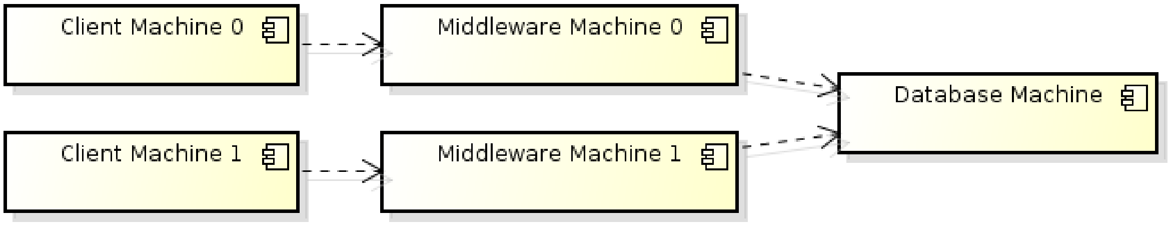


FIGURE 11-1

The setup defined here is based on the restrictions given by the project description document (1) and the experiments that were planned.

The following definitions are true for all configurations used for the experiments:

- Each machine runs only one application, either client, middleware or database
- Each machine runs only one instance of any application
- There is only one database machine
- Every middleware machine connects directly to the database machine
- Middleware machines do not communicate with each other
- For every middleware machine, there is a client machine
- All client accounts are distributed evenly over the client machines, so therefore, also evenly distributed of the middleware instance

The basic configuration used, where some details can change depending the experiment, are:

- 2 client machines, 2 middleware machines and one database machine
- All machines are baseline amazon instances
- The workload specified in section 4.3 is used with small messages
- Maximum of 8 total database connections from middleware machines to the database. (4 for each middleware machine)
- 64 client accounts use the system simultaneously, 32 for each client and middleware machine
- Experiments start with an empty dataset as defined in section 4.2

In terms of instrumentation, all machines running Java programs create a connection to a separate database machine, shown in Figure 11-2. This is so every machine can store their measurements at the end of the experiment in a centralized place, as described in section 6.1.

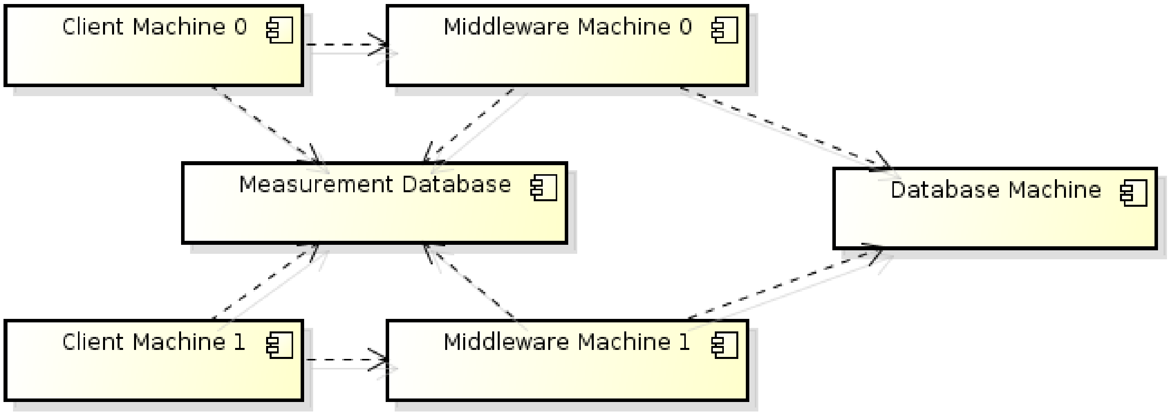


FIGURE 11-2

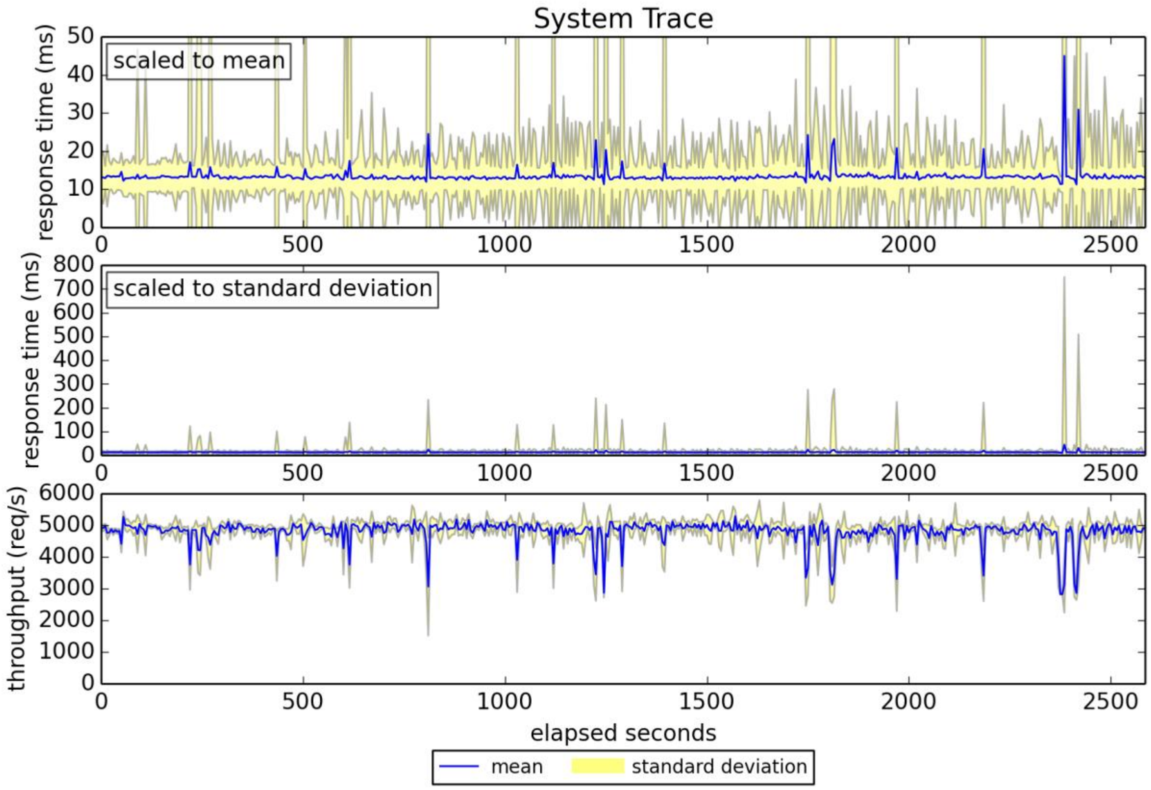
11.2. Deployment

The scripts defined in sections 8.3, 9.4 and 10.3 are used together to create the setups needed for the experiments and for retrieving experimental data. The script **bash/amazonDeploySetup1.sh** installs the full basic configuration and **bash/amazonStartSetup1.sh** runs it and starts the experiment. The starting script allows some parameterization like choosing the amount of client accounts or the maximum amount of database connections. **bash/amazonDeploySetup2.sh** and **bash/amazonStartSetup2.sh** are analogous but they setup a configuration with 4 clients and 4 middleware machines. The database machine is always setup manually but the starting scripts do restart the PostgreSQL instances.

12. Experiments Done on System

12.1. System Stability

The system was run for about an hour with the basic configuration, defined in section 11.1, and the workload defined in 4.3 with small messages. The intent was to see if the system's performance has a certain trend over time and if it has some cyclic or sporadic behavior. The system was setup and deployed with the **bash/amazonDeploySetup1.sh** and **bash/amazonDeploySetup1.sh** scripts. **python/system_trace_consolidator.py** was used to consolidate the measurement values located in **measurements.zip/system_3_factor/01_2_mid_08_conn_norm_db*.csv** and the graph was generated with **python/system_trace_graph.py**.

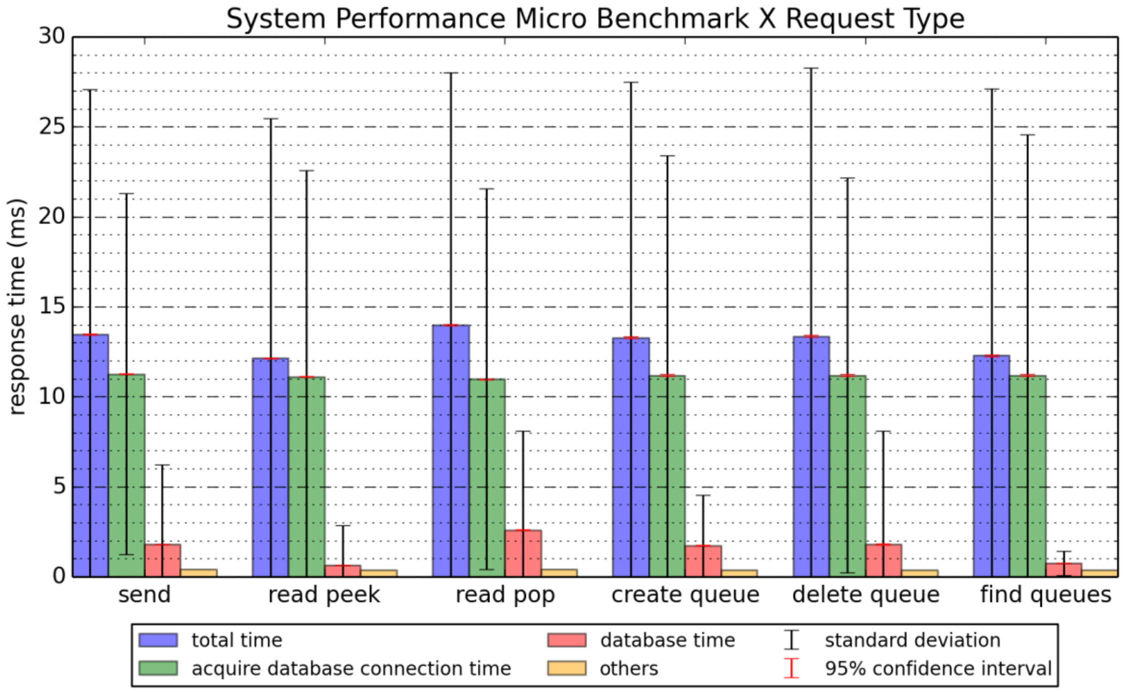


GRAPH 12-1

Graph 12-1 shows the systems performance over time. The first 2 plots show the same response times values at different scales while the third shows the throughput. The plots have a time resolution of 1 point per 5 seconds so it could be readable. Generally the system is stable except for sporadic performance loss spikes that happen simultaneously in all three plots. These spikes seem to happen in average every 2 minutes and will be analyzed in section 14.3. The mean throughput is 4870 requests per second with mean response time of 13.1 milliseconds.

12.2. Micro Benchmarks

Micro benchmarks give insight on where requests are spending time in the system. They were obtained from the same experiment data measured for the system trace and therefore have the same system and workload configuration. The graph data was extracted using **python/per_request_type_micro_benchmark_client_gen.py** and **python/per_request_type_micro_benchmark_middle_gen.py** and the graph was generated with **python/micro_benchmark_graph.py**.

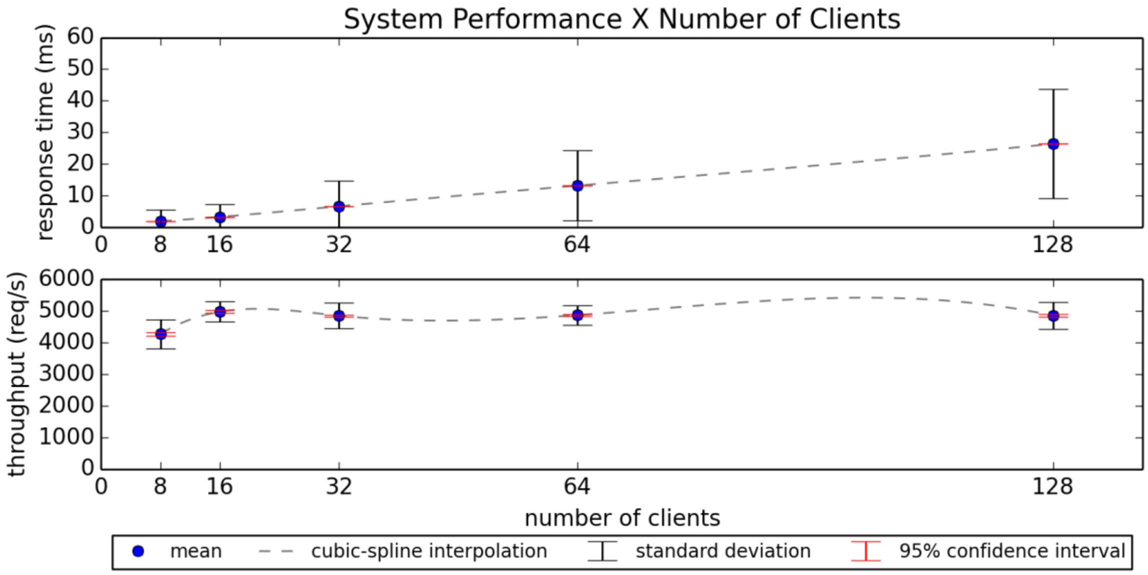


GRAPH 12-2

In Graph 12-2, we can see that for all request types, the most time spent per request on average is trying to acquire a database connection. In distant second, it is the time it takes to do the operation on the database and in third is everything else, which includes serialization/deserialization on both client and middleware and network communication between client and middleware. The fastest operation is read peek, taking 12.1 milliseconds per request and the slowest is read pop taking 14.0 milliseconds per request.

12.3. Number of Clients

The intent of this experiment is find how the system behaves in relation to the number of clients trying to access it. The system was run with basic configuration, defined in section 11.1, with the exception of the number of clients. The workload defined in 4.3 was used with small messages. The system was setup and deployed with the **bash/amazonDeploySetup1.sh** and **bash/amazonDeploySetup1.sh** scripts. **python/response_time_throughput_consolidator.py** was used to consolidate the measurement values located in **measurements.zip/system_client_number/** and the graph was generated with **python/connections_graph.py**.

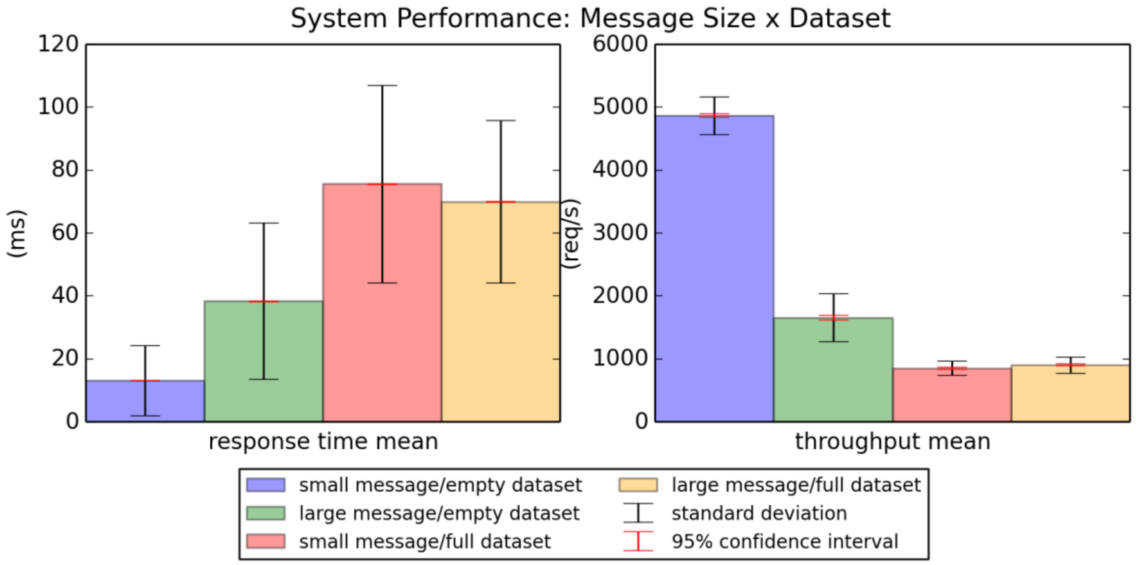


GRAPH 12-3

Graph 12-3 shows that response time and its standard deviation grow with the number of clients. The mean throughput reaches 4.979 requests per second at 16 clients and then plateaus at a level around 4.860. Event though, the cubic-spline interpolation make it look like the values are going up and down for the throughput mean of 32, 64 and 128 clients, they are numerically tied in their 95% confidence interval, which is less than 1% of their value. The mean response time at 16 clients is 3.2 milliseconds while at 128 clients it is 26.4 milliseconds.

12.4. 2^2 Factorial Experiment for Message Size x Dataset Size

This experiment was intended to understand how the message size and the affect the systems performance independently and together. The system was run with basic configuration, defined in section 11.1, with the exception of the dataset. The workload defined in 4.3 was used with small or large messages, depending on the experiment. The system was setup and deployed with the **bash/amazonDeploySetup1.sh** and **bash/amazonDeploySetup1.sh** scripts. The procedure `fill_db(TEXT)` was called on the database whenever an experiment with full dataset was done and was configured to use small or large messages to match the workload. **python/response_time_throughput_consolidator.py** was used to consolidate the measurement values located in **measurements.zip/msg_size_x_dataset/** and the graph was generated with **python/x_factor_graph.py**.

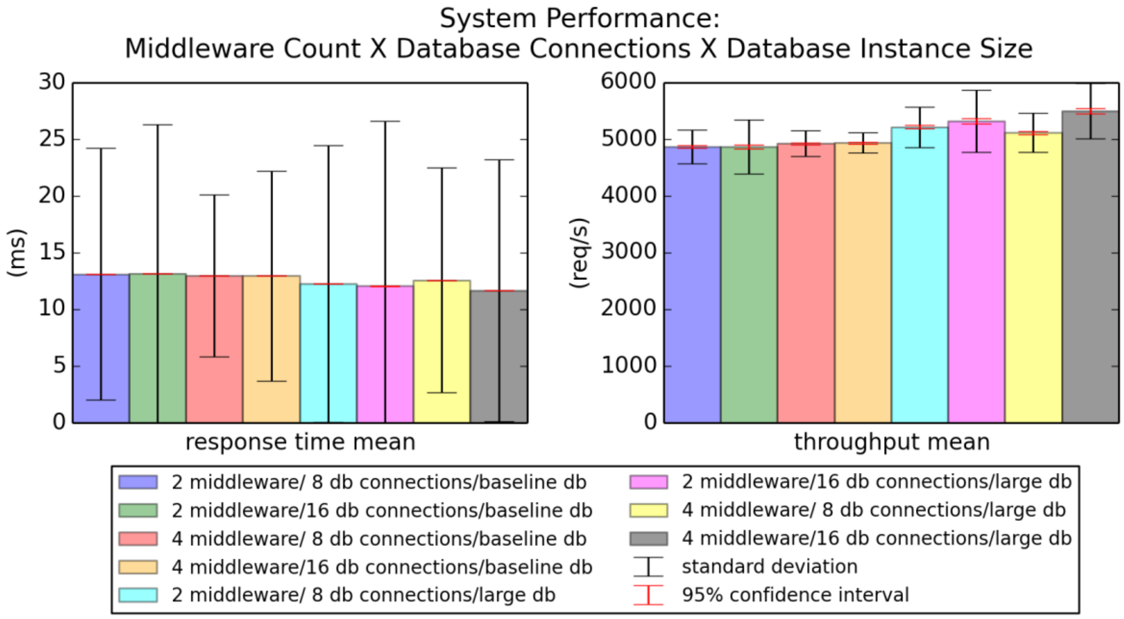


GRAPH 12-4

Graph 12-4 shows that message size and dataset both have an effect on the system's performance. Making the messages large or working on a full dataset independently worsen performance, however, curiously, if one uses both large messages and a full dataset together, there is a small performance improvement in relation to having small messages and a full dataset. The reasons for this is explored in the analysis in 14.6. The mean response time for small message/full dataset is 75.6 milliseconds with a throughput mean of 851 messages per second while large message/full dataset has 69.9 milliseconds mean response time and 904 messages per second.

12.5. 2^3 Factorial Experiment for Middleware Count x Database Connections x Database Instance Size

This experiment was intended to understand how the amount of middleware machines, the maximum number of connections permitted to the database and the performance of the machine running the database affect performance independently and in combination. The system was run with basic configuration, defined in section 11.1, with the exception of the client and middleware count as well as the type of machine that runs the database. . When mentioning numbers of database connections, it is referred to the total amount of database connections, not the amount used by each middleware server. Also, the number of client accounts does not change in this experiment, even though the amount of client machines do. The system was setup and deployed with the **bash/amazonDeploySetup1.sh** and **bash/amazonDeploySetup1.sh** scripts when 2 middleware instances were used and the scripts **bash/amazonDeploySetup2.sh** and **bash/amazonDeploySetup2.sh** when 4 instances were used. The database instance was changed from baseline to large manually, when needed.



GRAPH 12-5

Graph 12-5 shows that there are small differences from the worst configuration. The worst performance configuration belongs to 2 middleware/16 database connections/baseline database and 2 middleware/8 database connections/baseline database which are basically tied at mean response time of 13.1 milliseconds and mean throughput of 4.868 requests per second. The best configuration is 4 middleware/16 database connections/large database which has the mean response time of 11.7 milliseconds and a mean throughput of 5.499 requests per second.

13. System Performance

The maximum throughput configuration found, aside from the basic configuration defined in section 11.1, was:

- 4 middleware and client machines
- Database with a large machine
- 16 database connections in total

Its performance was a mean response time of 11.7 milliseconds and a mean throughput of 5.499 requests per second. In terms of scalability, the speed up was 12% and the system was able to handle 13% more messages per second compared to the basic configuration.

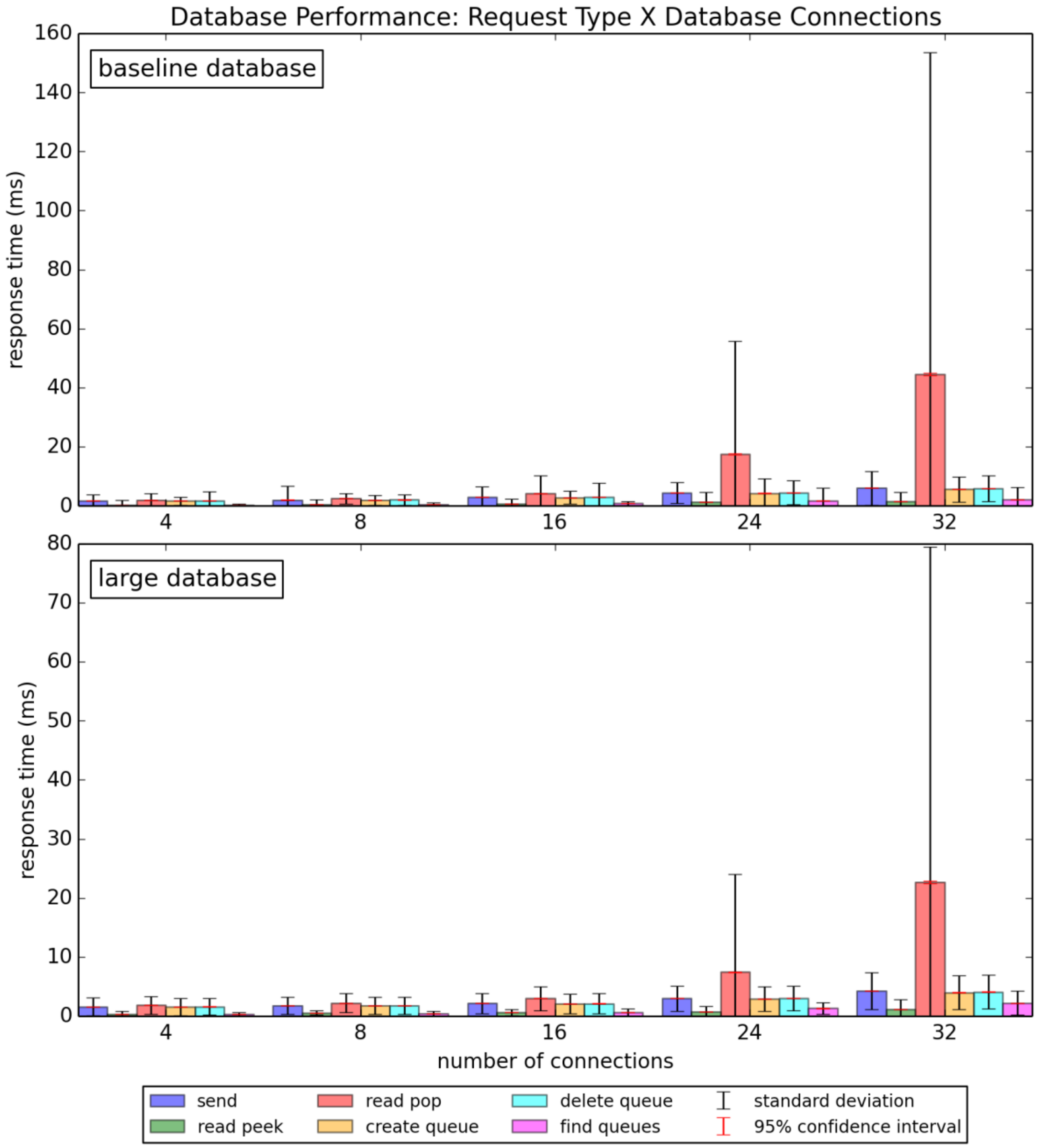
Performance values for various different configurations of the system were stated throughout section 12. Response time for messages of different sizes can be found in 12.414.6, for different number of clients can be found in 12.3 and for different numbers of middleware nodes in 12.5.

14. Analysis of the Performance Numbers

This section will revisit every experiment performed and dig in a bit deeper to understand why the system behaved the way it did.

14.1. Database Performance

When researching the subject of how many database connections should one make, I found suggestion that related the maximum number of active connections to the amount of cores on the machine and the amount of spindles (7). Since the amazon instances use SSDs, the spindle heuristic is clearly not valid but I thought that the CPU would definitely have an effect. When I saw that in Graph 8-1 and Graph 8-2 both instances have a maximum throughput peak at 16 database connections I was surprised. Investigating a bit more, I plotted graphs of the database response time per request type. The data was consolidated with **python/micro_benchmark_database_gen.py** and the graph was created with **python/per_request_type_database_graph.py**.



GRAPH 14-1

Graph 14-1 clearly shows that the performance decrease and increase of standard deviation when number of connections is above 16 is caused by 1 request type: read pop. This makes sense since contention performance issues were already predicted for this operation in section 8.1. The more simultaneous connections we have trying to pop a message from a queue, the more time is lost waiting for other connections to finish. The maximum throughput at 16 connections is most likely due to how the workload is set up and how frequent read pops happen in the same queue.

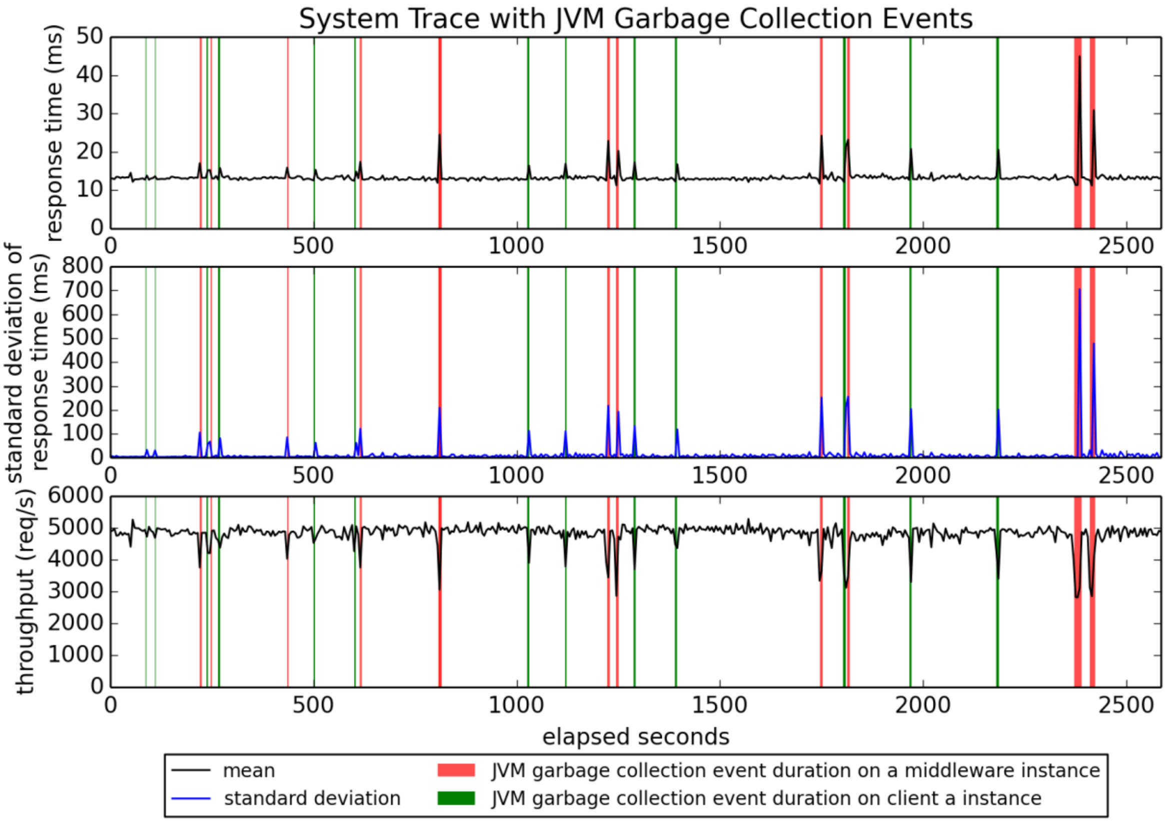
14.2. Network Performance

The network performance shown in Graph 8-3, Graph 8-4 and Graph 9-1 for communication between middleware and database as well as middleware and client behaved as expected. For an increasing amount of connections, the throughput increases and then saturates. One detail about the measurements made for database network, is that from measurements made with 32 connections and above, the standard deviation has a sharp increase. This is caused by the fact that these measurements are based on a larger amount of samples to lower their 95% confidence interval to an acceptable range. These larger samples sets have more chances of variation being caused by external factors, causing the increase in standard deviation.

As amazon stated (2), the baseline machine have high network performance. This can be seen given the low latency and the high throughput. The network clearly is not a bottleneck in the system.

14.3. System Stability

As mentioned before, the system is mainly stable aside from sporadic peaks of lower performance and standard deviation. Looking into the garbage collection logs of the Java application, there seemed to be a correlation with when these peeks happen and Full garbage collection events, so I plotted them together. This plot was created with data from **measurements.zip/system_3_factor/01_2_mid_08_conn_norm_db*.csv** and by the script **python/system_trace_with_gc_graph.py**

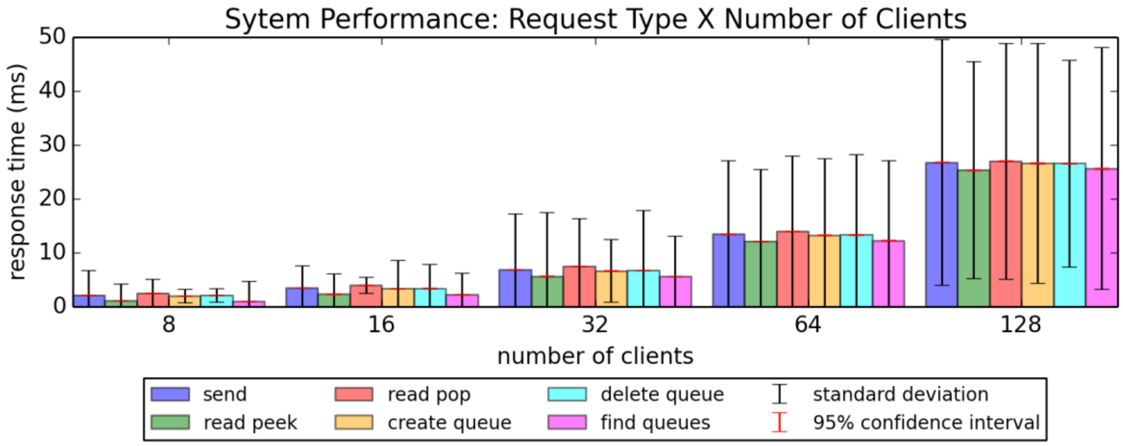


GRAPH 14-2

The green and red bars show full garbage collection events on the client instances and middleware instances, respectively. The bars also show how long each event took, the wider the bar, the longer the event. Graph 14-1 shows very nicely that every major peak can be explained by a full garbage collection on either a client machine or a middleware machine. Also, all full garbage collection events cause peaks. Furthermore, the longer the garbage collection event, the greater the peak, as can be seen at the last 2 peaks.

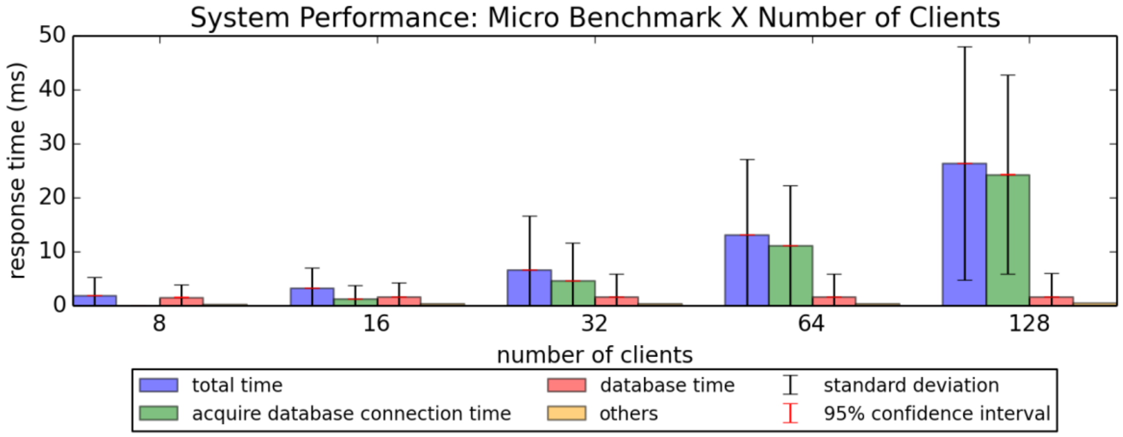
14.4. Number of Clients

In Graph 12-3, we can see that throughput plateaus when the amount of clients is 16 or above. To investigate a reason for this, first I plotted the performance of each request type in relation to the number of clients. The data was consolidated with **python/per_request_micro_benchmark_client_gen.py** and the graph was created with **python/per_request_type_client_graph.py**.



Graph 14-3

Observing Graph 14-3 we can see the all request types get slower with the amount of clients, so this doesn't tell us much about the cause. From that, I decided to plot the micro benchmark in relation to number of clients. The data was consolidated with **python/micro_benchmark_client_gen.py** and **python/micro_benchmark_middle_gen.py** and the graph was created with **python/micro_benchmark_graph.py**.



Graph 14-4

Here, we can see main culprit of the slowdown. When the amount of clients is equal to the amount of available data connections, there is no competition for the resource. As the amount of clients increases, each request competes more and more with other requests for access to the database. This something that happens to all types of requests and that's why we see that they all get slower, independent of what they do in the database.

14.5. Micro Benchmarks and Identifying the Bottleneck

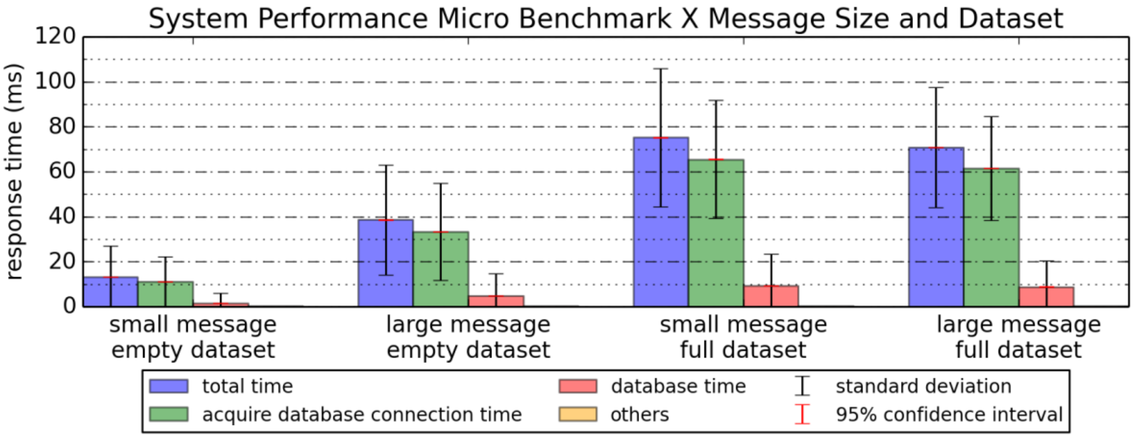
Graph 12-2 shows that the greatest time sink for a request is acquiring an unused database connection however, if the number of database connections is the same as the number of clients, as can be seen in Graph 14-4, this time disappears, confirming that it is not the actual data structure that is used to acquire connections that is the time sink, but the actual availability of free connections. Requests simply have to wait around 11 out of the total 13 milliseconds in the base configuration for other request to finish before they can gain access to the database.

Graph 12-2 also shows that each request types waits on average the same amount of time and that the only difference between their response times is caused by how long the take to call their respective stored procedures.

This leads me to believe that the database is the bottleneck of the system. If the database were able to handle more connections simultaneously with the same response time, there would be a drastic improvement in overall system performance for every request type. Unfortunately, this is unlikely, as Graph 8-1, Graph 8-2 and Graph 14-1 show, even if a more powerful machine is used, the implemented stored procedures are not able to handle an increasing amount of connections when the workload includes the read pop operation.

14.6. 2^2 Factorial Experiment for Message Size x Dataset Size

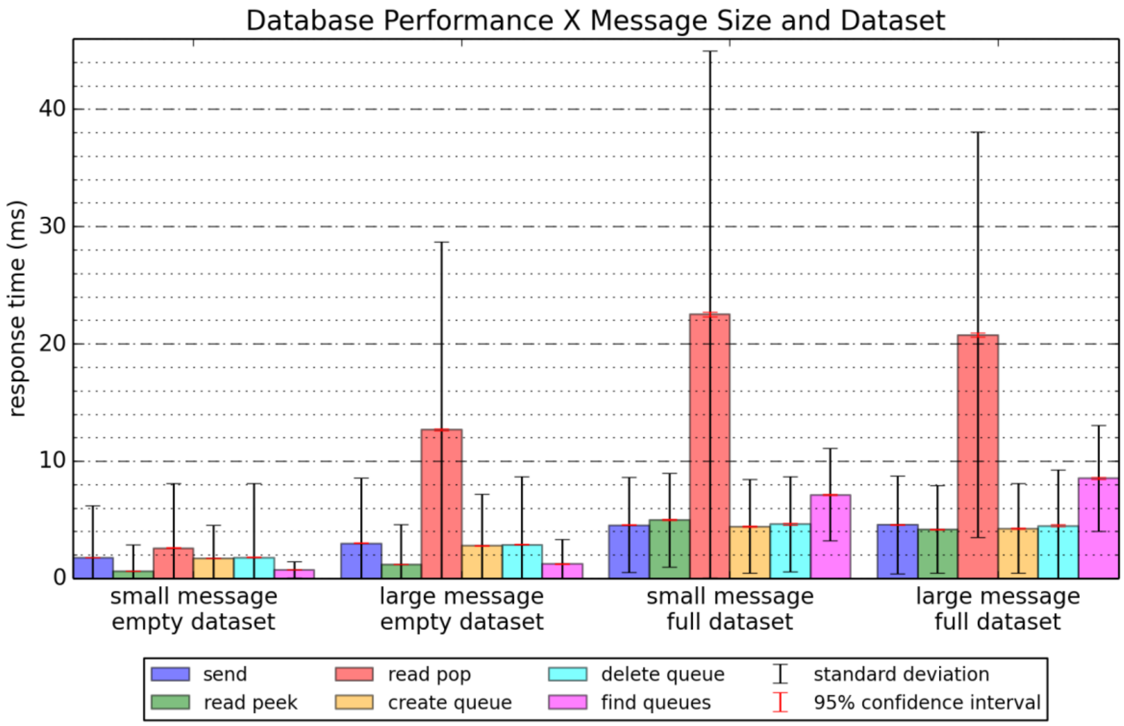
As mentioned in section 12.4, it was surprising for me that a full dataset using large messages had better performance than one using small messages. After I repeated the experiment a few times, I convinced myself that it wasn't a measurement error and that there must be some underlying reason. I started out checking the experiments micro benchmark. The data was consolidated with **python/micro_benchmark_client_gen.py** and **python/micro_benchmark_middle_gen.py** and the graph was created with **python/micro_benchmark_graph.py**.



GRAPH 14-5

Unfortunately, it didn't tell me much. Just that the time spent at each part of the system maintains their relative size. From that, I decided to plot how each request type behaved with the different configurations. The data was consolidated with **python/per_request_micro_benchmark_client_gen.py** and the graph was created with

python/per_request_type_client_graph.py.



GRAPH 14-6

Here we can see some interesting behavior happening. We see that, read pop and find queues are the request types that are most effected by the different parameter levels. Find queues is mostly affected by the levels of dataset, which makes sense since it is a query that has to look at every message the database to determine which queues have a message for the client. Observing, differences between small message/full dataset and large message/full dataset, read pop is largely responsible for the average performance increase. Also, in the small message/full dataset, its standard deviation is much larger than in large message/full dataset, which leads me to believe that it is having less resource contention issues. I believe that, since the selects are generally slower in the large message/full dataset scenario, database threads enter the resource contention part of the code less frequently thus improving performance. Small message/full dataset is able to execute selects faster, so it has more competition for resources and it effectively starts thrashing.

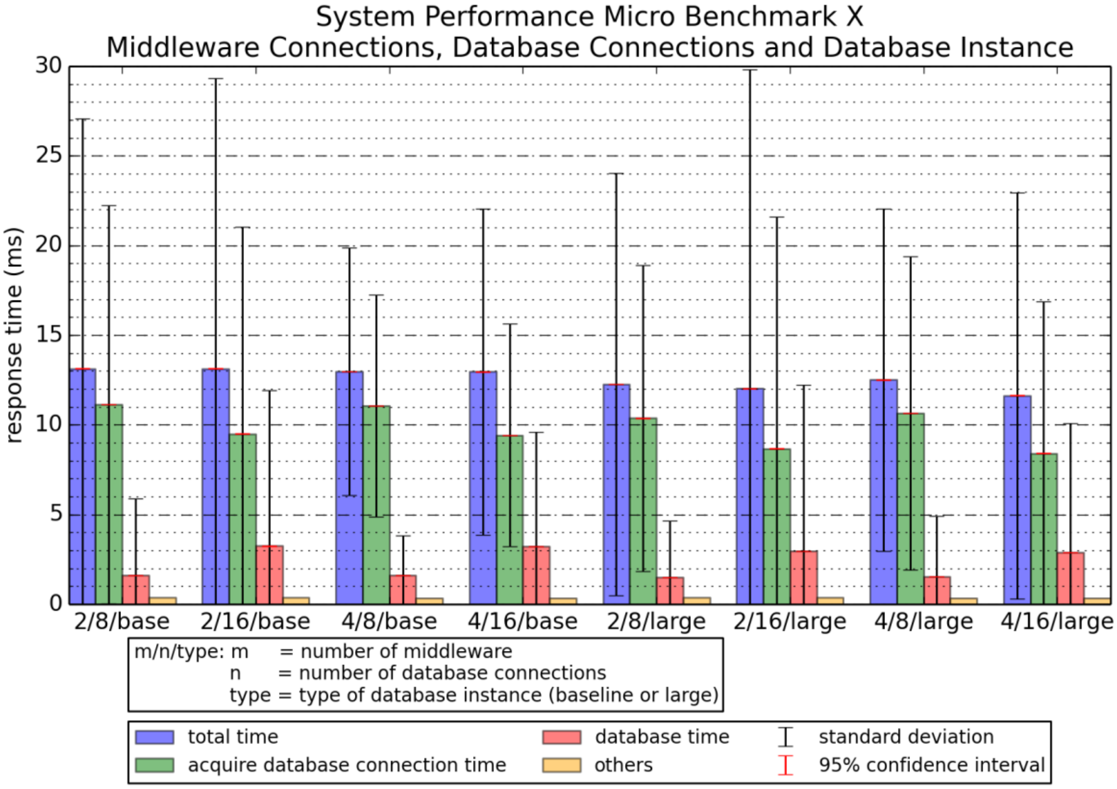
2^2 factorial design with replications analysis was done on these experiments. The effects and the allocation of variation were calculated with this script: **python/22r_design.py**. All effects have 95% confidence level below 3% their value, so all effects are significantly different than 0. The numbers show that:

- The mean performance is 47.0 milliseconds response time and 4928 requests per second throughput
- The effect of the message size is 2.8 milliseconds response time and -766 requests per second throughput
- The effect of the dataset is 25.7 milliseconds response time and -1246 requests per second throughput
- The effect of the message size and dataset together is -5.6 milliseconds response time and 795 requests per second throughput
- Around 55% of the variance is caused by the dataset for both response time and throughput

These numbers confirm observations made looking directly at the graph that and support the analysis.

14.7. 2^3 Factorial Experiment for Middleware Count x Database Connections x Database Instance Size

We can see that in in Graph 12-5, the effects of the parameters chosen for the experiments do affect the system's performance, but not very much. As stated in section 13, the system did manage to have 13% throughput increase but this came at that cost doubling the number middleware instances and doubling the number of VCPUs and provisioned IOPS on the database. To take closer look at the effects of each configuration, a micro benchmark plot was done. The data was consolidated with **python/micro_benchmark_client_gen.py** and **python/micro_benchmark_middle_gen.py** and the graph was created with **python/micro_benchmark_graph.py**.



GRAPH 14-7

We can see that, whenever the number of database connections is increased, the time to acquire a connection decreases and the average time to execute a query increases. This makes sense, since more concurrent operations on the database slows it down, as seen in previous sections. We can also see that using a more powerful database instance lowers database time, but very slightly. In all configurations, the requests spend more time for access to the database then actually executing.

2^3 factorial design with replications analysis was done on these experiments. The effects and the allocation of variation were calculated with this script: **python/23r_design.py**. The numbers show that:

- The mean performance is 12.6 milliseconds response time and 5092 requests per second throughput
- The effect of the number of middleware is -0.2 milliseconds response time and 94 requests per second throughput
- The effect of the number of database connections is -0.3 milliseconds response time and 131 requests per second throughput
- The effect of the database instance type is -0.28 milliseconds response time and 119 requests per second throughput

- The effect of the number of middleware together with number of database connections is -0.1 milliseconds response time and 41 requests per second throughput
- The effect of the number of middleware together with database instance type is 0.0 milliseconds response time and -6 requests per second throughput
- The effect of the number of number of database connections together with database instance type is -0.1 milliseconds response time and 63 requests per second throughput
- The effect of all 3 parameters together is 0.1 milliseconds response time and -36 requests per second throughput
- All effects except the number of middleware together with database instance type are significantly different than 0, due to 95% confidence interval.
- Most variance is caused by the number of database connections and type of database instance.

These numbers confirm observations made looking directly at the graph that and support the analysis.

15. Conclusion

As mentioned throughout the report, it was discovered that the implementation of the read pop operation stored procedure has a significant negative effect on the database performance and consequently on the systems performance. This is due to the fact that, although the read pop implementation correctly handles concurrent access, it does not scale well when presented with larger loads. In section 12.5, we show that even when we double the amount of middleware nodes, double the amount of database connections and double the processing power of the database, we get only a 13% throughput increase.

In section 14.6, we can see that even though only 2 request types are significantly getting slower in the database, all request type get slower for the system. This is because all request types share the same queue for waiting for a database connection.

If I were to implement the system anew, I would invest time in making the read pop operation behave better for concurrent access. Perhaps, instead of actually executing a DELETE, an UPDATE could just mark the row as popped and this could be more efficient. Another idea is to borrow an idea from networking called exponential backoff (8). The idea is, if a thread notices it is competing for the deletion of a row, it waits for a random amount of time that grows exponentially. This can help avoid thrashing.

To solve the problem that all system request types slow down if only a few actually slow down in the database, we could implement different queues for waiting for database connections in the middleware. Request types that have low response time and variance could simply wait in a different, high priority queue, while others wait in a normal queue.

All of these ideas would, of course, have to be validated experimentally if they significantly improve the performance characteristics of the system.

16. References

1. **Alonso, Gustavo.** Systems ETH Zurich. *Advanced Systems Lab.* [Online] http://www.systems.ethz.ch/sites/default/files/file/COURSES/2014%20FALL%20COURSES/2014%20Fall%20ASL/ASL-2014-Project-Course-Description.pdf.

2. **Amazon.** Amazon EC2 Instances. *Amazon Web Services.* [Online] http://aws.amazon.com/ec2/instance-types/.

3. **PostgreSQL.** Constraints. *PostgreSQL.* [Online] http://www.postgresql.org/docs/9.0/static/ddl-constraints.html.

4. —. CREATE SEQUENCE. *PostgreSQL.* [Online] http://www.postgresql.org/docs/8.4/static/sql-createsequence.html.

5. —. Numeric Types. *PostgreSQL.* [Online] http://www.postgresql.org/docs/9.1/static/datatype-numeric.html.

6. —. Transaction Isolation. *PostgreSQL.* [Online] http://www.postgresql.org/docs/9.1/static/transaction-iso.html.

7. —. Number Of Database Connections. *PostgreSQL.* [Online] https://wiki.postgresql.org/wiki/Number_Of_Database_Connections.

8. **Wikipedia.** Exponential Backoff. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Exponential_backoff.