Cloud Technologies

Tony Dear, Jason Jee, Harry Jiang, Dustin Kwong, Paul Ruan
cs162-ba, cs162-ab, cs162-ai, cs162-at, cs162-bb

Overview

In this phase of the project, we modify the ChatServer to include a backend database, as well as deploy it to Amazon's EC2 cloud. The former is designed using the entity-relationship model and implemented using MySQL. With this added functionality, we can now store state information that will persist across instances of the server and allow the server to reuse user, group, and message information. After setting up the database, we deploy the server and run it on a virtual machine instance in the Amazon EC2 cloud. This allows multiple client instances to start up and access the server and simulate client-server interaction. We were then also able to perform several benchmarking tests on its operation.

Specifications

To accommodate the addition of the database, a MySQL instance runs and houses the tables for the database. This MySQL database runs separately from the server and is accessed using the Java Database Connectivity (JDBC) API. Our server functions as it did in the earlier phases of the project, but it obtains state from the database when it starts up and continue to write data back to the database during operation.

The ChatServer class has changes to deal with message handling and semantics of a logout. Message handling will change to allow offline message acceptance, so that users who are now logged off can still receive messages. As for logging out, a user will no longer be forced to leave its groups, and its username will continue to stay reserved for the specific user. Then, if a registered logs in again, he should be able to restore his state from the last time he logged on.

A couple of new commands on the command line now had to be recognized by the server. In particular, *adduser* allowed one to actually register a user to the server with a password, since any user without a record in the backend database is automatically rejected. The *readlog* command allows for a user to retrieve all messages sent to him while he was offline. Upon invocation, all these messages should be cleared out, and they should not be shown again if the command is called for a second time.

Considerations

Because database queries may involve accesses to disk, they can potentially be very slow and inefficient. To make sure the database performs optimally, we must make sure that the entities and relationships are well-designed and use benchmarking to test our design decisions. One main decision was how to handle the database storage alongside our in-memory data structures. We already had lists and structures to hold our data in memory from phases 1 and 2, but now that we have a database we had to decide how to handle both appropriately. At any point in time we may have inconsistencies between the two sides, and if the server crashes then it may be difficult to restore the correct state.

In addition, we considered the following issues in our design:

- Other than message handling and logout changes, specifications from phase 2 hold.
- Messages can be received while offline.
- Logouts do not result in leaving groups.
- Logins result in a check against the database.
- Database can be accessed/queried quickly.
- Server and Client can be deployed to the cloud.
- Server state persists across instances.

Implementation

We divide our implementation into several stages. The first is design of the database. Here we will have to decide what tables to have in order to take care of the above points. The main challenge is to ensure cooperation between our ChatServer and the database tables. This included both the problem of table design and interfacing with them in the ChatServer.

To this end, we created a new class called *DBHandler*. It is not instantiated; instead, it contains a number of different static methods that can be called externally to read and write to the database. This would allow our main ChatServer to communicate with it. Such an interface is based on the parameters passed to these methods and the return values back up to the caller. Inside each of these methods, we write SQL statements that execute on our database. Then after the correct read or write, we return any results back up.

The second stage was deployment to the cloud after we ensured that everything functioned correctly locally. Once this was done, we were then able to do some interesting performance tests. In particular, we modified and added some new code to our server to allow it to record and present information regarding how it is doing on handling users and send requests.

The Database

Our goal was to make our database as simple as possible, so that a minimal amount of information would need to be stored. In addition, this would probably cut down on the number of accesses we do to the database, an operation that is certainly much less efficient than simply operating in memory. Thus, we only included three tables to hold all the necessary information.

The *Users* table contains three fields: username, salt, and hash. It stores user data in an alternate form of what one normally thinks of as the username and password. Here, we store a salt value and a hash of the original password instead of the password itself, for security reasons that we explore below. This table is modified whenever a new user is added via the command line on the server, as well as whenever a user attempts to log in. Because all usernames must be unique, the username field became a natural primary key.

Memberships simply kept track of which users belonged to which groups. With two fields representing the username and gname, this table was simply a two-field list of users and groups. Now because many users can belong to many groups, it did not make sense to make either one a key by itself. Instead, each pair is its own key, since all mappings should be unique anyway. To obtain a simple list of chat groups, one can simply select distinct gnames from this table.

Finally, we have a table for offline *Messages*. There was a lot more flexibility here. We noted that for *readlog* to operate successfully, we needed to keep track of sender, recipient, sqn, timestamp, and of course the message itself. One slight problem was ChatGroups—because we would not be able to differentiate between two different members of the same group reading the log, we decided to add an additional field called destination, primarily to indicate whether the message was intended for the chat group. Thus, we assigned our primary key to be the combination of sender, receiver, and sqn, since sqn should generate uniquely for each sender.

The DBHandler class

As mentioned above, we added this new class to interface between the ChatServer and the database. The methods we have include the following:

addUser(String username, byte[] salt, String hash)
addToGroup(String username, String gname)
removeFromGroup(String username, String gname)
writeLog(Message msg, String recipient)
readAndClearLog(String username)
getSalt(String username)
getHashedPassword(String username)
getUsers()
getGroups()
getMemberships()
getUserMemberships(String username)

Most of these methods are relatively simple. For example, the first four involve writes to the corresponding table, whether through INSERTs or DELETEs. The adder methods do exactly as they are named, while writeLog extracts the fields from its Message parameter to write into the Messages table along with the recipient. Both removeFromGroup and readAndClearLog use the DELETE FROM command to clear data from their respective databases. The rest of them are just getter methods that use SELECT statements.

For most of these, writing the SQL query was not too much trouble. One change we made was to change all Statement objects within these methods to be PreparedStatements. In particular, this was important for all the methods which take a parameter input. Operating under the paradigm that we can never trust a user, we cannot be sure that a user will not try to supply executable SQL code in the parameter. Thus, we use PreparedStatements to first fix the query that we should execute, forcing the input to be always treated as a String no matter what. Then an attacker would not be able to inject malicious SQL code into our parameters.

Ensuring Consistency

Consistency between the backend database and the in-memory data structures was one of the bigger challenges of the design. Our goal was to make as few changes to the existing structure already, but at the same time keep in mind the efficiency and performance of our chat server. In particular, we had to keep in mind the separate notions of registered users versus online users, as registered users may or may not always be online at the moment.

Because of our desire for performance, we decided to continue utilizing our in-memory structures, such as a users list, chat group list, etc. We also added a new list to keep track of all registered users over time. This aided in the performance of logins, since we would not have to call up the database to check if a particular user exists. Otherwise, all other lists refer to active users and memberships.

At the same time, we were extremely wary of extreme conditions, such as the server crashing. If there were changes in the in-memory structures that did not make it back to the database, then functionality would be impaired. Thus, we decided to make all changes write-through as much as possible. Whenever a new user is registered, for example, the change is made both to the registered users list on the server as well as the Users table in the database. Thus, if the server is ever shut down, we need not worry about cleaning up its records.

To facilitate this process, we also found it necessary to add a couple new methods in the server. If the server were to start up while data lived inside the database, then the server needs to restore all of its known state. Thus, in our startup method, we automatically tell our server instance to check the database for any stored information and to load it all into memory. This includes all registered users, existent chat groups, as well as memberships. In addition, we have a method that restores all of a user's chat group memberships upon login.

By forcing the database and the server to constantly update one another, we can be relatively safe against server anomalies. At the same time, this design is rather lightweight and minimal, since we have in-memory duplicates of database information, saving us unnecessary accesses.

User Authentication

For user authentication, we store all necessary information in our database—in particular, the Users table. The straightforward method simply includes both the username and a password associated with that username in each record. They are both just strings, with the username being the unique key identifier. However, this is not secure and can be very vulnerable to different types of attacks. In addition, this gives the system administrators of the server too much power in being able to simply see the passwords of each user.

Instead of performing authentication this way, we did so similar to the way that the Unix system does it. We do not store the password in plaintext in the database; rather, we store a "salt" and a hash value. The salt is just a random string, selected by the system whenever we create a new record for a user. The hash value is then calculated using both the provided password and the salt as input. To authenticate in the future, we use the salt and recalculate the hash with the provided password. Then if this matches the hash stored in the database, we can be confident that this is the right password (and user).

While we can discard the salt and just use the password by itself to hash, this is not secure enough. An attacker can perhaps try inputting different passwords and recording what the hash values come out to be. This can be especially useful for common passwords such as "qwerty" or "password." Sooner or later, the attacker would be able to guess what a user's password is by simply examining the hash value. The salt also prevents an attacker from being able to tell if two different users have the same password, since they will ideally have different hash values.

We assume here that we have a secure enough hash function to provide the security properties that we expect. Such a function should be efficient to compute, since we do not want to impact the performance of the chat server whenever someone logs in. But it should be extremely difficult to do the other way around; given a hash value and salt, it should be nearly impossible to guess the password. In addition, the hash function should map as evenly as possible, such that it is hard to find any two pairs of inputs which hash to the same value. Such a hash function is difficult to come up with on our own, and we plan to use well-known and well-trusted implementations, such as SHA-256, which is the scheme that we use.

This scheme provides several advantages. First, any administrator of the server who has full access to these tables would not be able to see the user's passwords. Because this can be done at this lower level with passwords only, the scheme can then be scaled to protect other potentially more valuable information. While it is not necessary to do so in our chat server here, it better matches the methods that are actually implemented in real systems. Thus, we have gained a better sense of privacy with our chat server.

We also have a better notion of security. Suppose that our database was hacked by an attacker, perhaps by SQL injection or some other method. Then what he would see is the username, the salt, and the hash value. But nowhere would he see a password. Even with this information, it would be extremely difficult to try to assume the identity of one of the victims or try to hack into one of these accounts, as the authentication system asks for a password, which the attacker lacks. While this is still not completely safe, we have set up an additional barrier to our database.

In summary, we implemented a Users table with three fields: Username, salt, and hash value. The username is just a string, and will act as the key for this table. The salt and hash are just simple values (such as in hex); all three have non-null constraints and must be determined when the new record is created for that user. The hash value is a hash of the combination of password and salt. To authenticate, we recompute a hash value with the provided password and stored salt, comparing the output to the stored hash. We hope that such a scheme will provide much more security than one expects with just a simple plaintext password.

We must also modify our ChatServer program appropriately. When logging in, we first check the username against all registered names. If it does not appear in that list, then we reject their login attempt. Otherwise, we attempt to verify their identity. We take their provided password and the salt stored for their record (using DBHandler) to compute a hash value. This operation is done in a separate method that we defined, using a MessageDigest object to perform the hashing. If this hash matches the one stored, then the client is logged in as before. We also reinitialize his chat group memberships from database records.

Finally, the last change is that upon logoff, a user needs to remember that he was part of some set of chat groups. Because we did not add any new data structure representing "all members" for a chat group, we did not actually have to change anything here. A user will be removed from the old lists, which only represent online users. So this functionality is correct. When the user logs in again, part of that functionality will be to restore all his chat group online memberships.

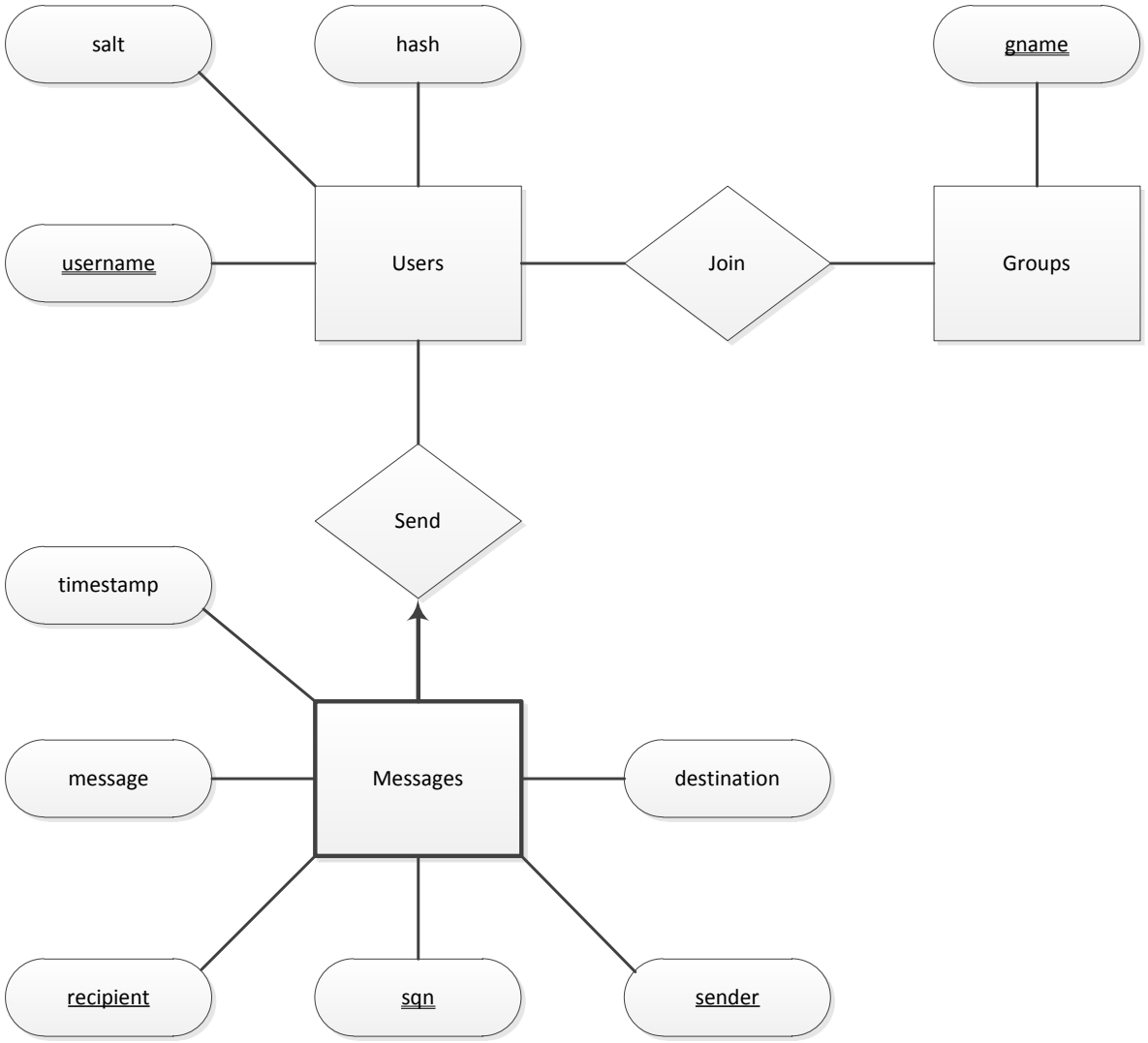

Figure 1: Simple entity-relationship model of our database design

Logging Offline Messages

Because of the additional requirement in which logged off users must be able to receive all messages that he has missed while he was offline, we have decided to add a Messages table. This Messages table will log all messages sent to users that are currently offline, but not messages that are sent to online users successfully. The recipient is the name of the user that was logged off and unable to receive the message. We have a separate destination field to determine whether this was a group message or an individual message that was missed. This is necessary so that the client will receive the correct output from the message.

If a user wishes to check whether he has missed messages, he runs the *readlog* command. This command will run a query against our database, specifically the Messages table, which then returns all tuples with recipient equal to the user's username. The results are also sorted according to the timestamp to ensure that the messages will be received in the original

ordering. This will prevent out of order messages. The tuples are processed as if the messages were received, after which these tuples in the Messages table are deleted. This keeps the Messages table at a reasonable size.

We have made the design decision to store multiple copies of a message to a ChatGroup, one for each user belonging to the ChatGroup that is not currently logged in instead of just storing one message, as this will simplify the process of determining when the message has been received by all users belonging to the ChatGroup. In other words, we have decided to trade off space requirements for simplicity of code and logic.

Local Testing

We completed the following tests locally before deploying our chat server to the cloud. As we made major changes, we ran our old tests to ensure that everything is still intact. The appropriate new tests were run as we make the corresponding additions to our chat server. Once on the cloud, all the tests were run again to ensure that proper operation did not actually depend on it being deployed locally.

Old tests

A) Connecting, disconnecting, logging out
B) Login queue
C) Timeout
D) Groups
E) Sending messages

New tests

A) Creating new users, storing username and passwords, logging in.
- Attempt to add a username without a password.
- Add a username with a password.
- Attempt to add a user that already exists.
- Log in with just a username.
- Attempt to log in with a username and a bad password.
- Attempt to log in with a non-existing user.
- Log in with a username and the correct password.
- Log in with a username and someone else's password.

B) Storing groups and membership information in the database.
- Join groups, log off and check if still in same groups.
- Join group until full, log off and have another user attempt to join full group.
- Leave group and check if still in group.
- Leave group, log out, and check if still in group.

C) Readlog and storing messages while logged off
- Try readlog command when there are no messages to be received.
- Readlog when there are known messages received while logged off.
- Try readlog after a readlog.

D) Crashing
- Log some users in, join some groups, send some messages to users that are logged in and off, shut the server down, and restart it; check the following:
  - Ensure that all users still can log in
  - Ensure users are still in correct groups.
  - Ensure users who received offline messages can still use readlog to see them.

Also, to aid debugging, we added a thread to the chat server to handle some commands from command line. A tentaive list of commands to add follows:

users	//prints all users
users *groupname*	//prints all users in the specified group
groups	//prints all groups
active-users	//prints all users who are logged in
thread-count	//prints the number of active threads

Benchmarking Methodology

To do benchmarking tests, we made special classes and added extra functionality to our chat server. In particular, we have BenchmarkReceiver and BenchmarkSender classes. Because they are really just normal ChatClients that take note of metrics such as RTT, we decided to abstract our old ChatClient class. Then we had the two benchmark classes extend this abstract class.

On the database side, we added a Rtt table. For each message sent over the server, the round trip time, between sending out and receiving an ACK, was recorded into this table. Recording data this way was simple, since all we needed to do was to use our DBHandler class to insert the appropriate numbers at the appropriate time. Thus, we could easily extract statistics, such as the average and standard deviation, from the table using SQL commands.

On the cloud, we had a couple of scripts to generate a specific number of users, one to create receivers and one to create senders. For example, in measuring latency versus number of clients on the server, we changed this number in our scripts several times to gather sufficient data. If we were testing 100 users (the maximum capacity of the server), we would make 50 senders and 50 receivers. We also have another couple of scripts telling the online users to send and receive. The senders would send messages to random receivers in some specified interval length. For example, we set this time to 100ms for the initial benchmarking test. This functionality was done by having the senders alternate between sending messages and sleeping for the given time.

Average RTT

This was the standard benchmarking test that we did—finding average RTT over the number of users in the server. To do so, we take the initial setup described above: We set the number of senders and receivers to be some number between 0 and 50 each, and tell the senders to send messages at 100ms intervals. In addition, we arbitrarily set the number of messages for each user to send to be 1000, so that we can get a good representative sample. We did this for several inputs, ranging from 2 users on the server to 100 at maximum capacity.

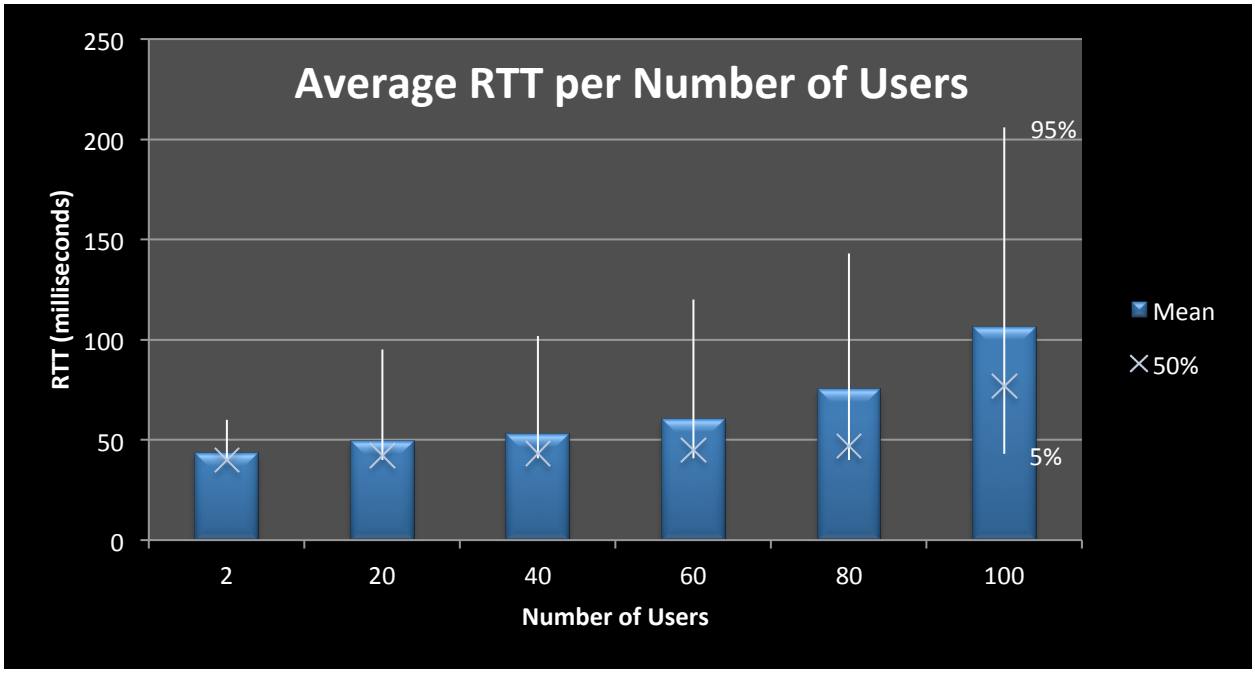

Figure 2: Graph of average RTT as a function of the number of users on the server

As can be seen on the graph above, we have samples for every 20 users that we inserted into the sever (we obviously could not sample with 0 users, so we used a minimum of 2). The blue bars indicate the mean RTT time for each sample. The crosses indicate the median, while the whiskers extend to the 5th and 95th percentiles of RTTs. As can be seen, both the mean and median RTT remain relatively constant as the number of users increases up to 80.

At maximum capacity, we see that both increase noticeably, but this behavior is much better than what we expected. It would appear that the RTT has a slight linear trend, but the slope is very small, and the increase is nothing close to exponential.

Another observation is that our maximum (95th percentile) increases steadily, with somewhat of an exponential characteristic. The effects of such random outliers are expected to become more noticeable as the number of users go up. Also, we tested so that each sender sent out 1000 messages, no matter how many users we had on the server. Thus, at the low end of the spectrum, we gathered 1000 data points (1000 for the 1sender), but at the high end we had 50000 (1000 for each of the 50 senders). So with more data points, our averages should in theory be more accurate by the law of large numbers. But we also expect to see more extreme outliers, since more samples also means more chances for these outliers to show up.

Fairness of Throughput for Users (Bonus)

As a second test, we decided to test fairness among our users. In other words, is the server processing each user's requests in a fair manner? To do such testing, we decided to record both the RTT and the username in our table. Then in analyzing our numbers, we just group all the RTTs by user and take the average for each one. We can then take these averages and plot them over a histogram to examine it.

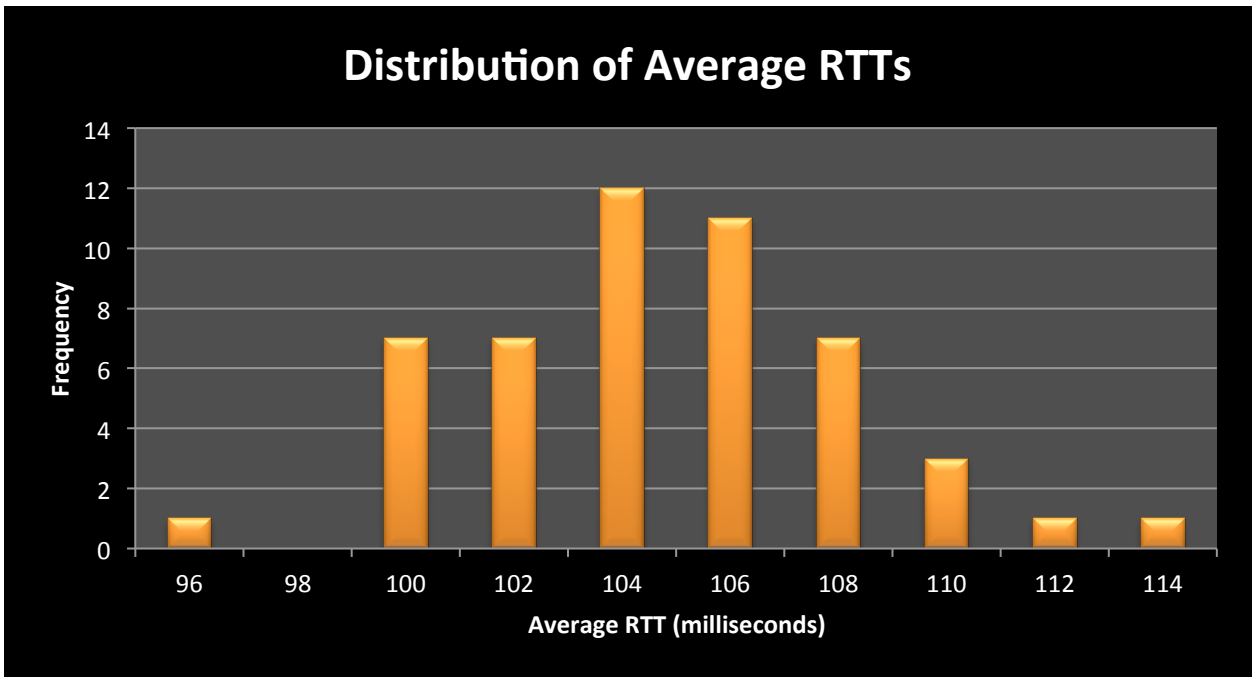

Figure 3: Histogram of average RTTs
Mean: 103.6
Standard Deviation: 3.5

This test was done with 100 users on the server, with 50 senders and 50 receivers. Thus, we have 50 data points, each one being an average for a particular receiver. As we can see on the plot above, the distribution is relatively normal, with a small standard deviation. This is extremely desirable. The bulk of the users on the server all have about the same RTTs, showing that the server tries to be as fair as possible for all users. There are several outliers on either end, but of course that is to be expected.

Throughput versus Number of Users (Bonus)
In this test, we analyze how throughput changes along the dimensions of number of users versus the rate of sending. More specifically, if we increase the number of users by some amount and then decrease their sending rate by the same factor, does the throughput stay constant or not? If the chat server were perfect, then the number should be relatively stable, analyzed using simple math as follows. In a set amount of time, the number of messages sent out (the throughput) is just the product of the number of users and the rate of messages sent out. So if the number of users goes up by a factor x and the sending rate goes down by the same factor, the product and thus the total throughput remains the same.

Our methodology for measuring this metric again reused our Rtt table in the database. This time, we also recorded the timestamp for when the RTT entry is recorded. Each sample involved changing the number of users as well as their send rate. Starting at 20 users sending at 100 ms intervals, we gradually increased the number of users to 40, 60, 80, and 100, increasing the send intervals for each to 200, 300, 400, 500 respectively. Then after recording all the RTTs for some number of users, we simply count how many of them fit into a 10 ms window on average.

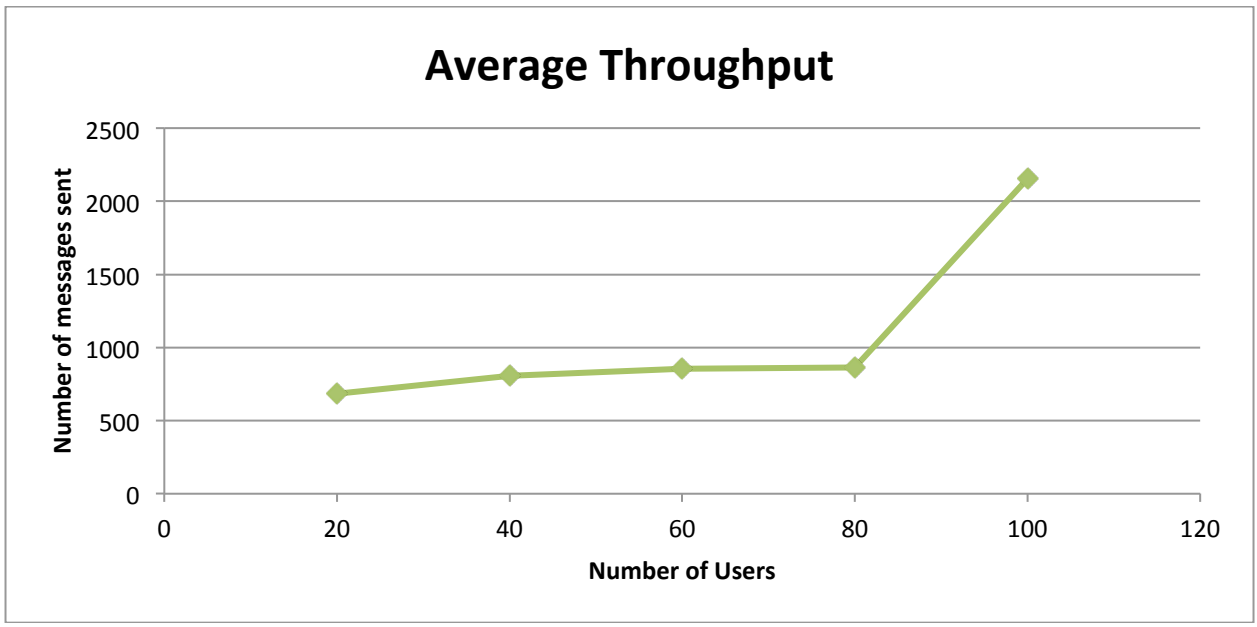

Figure 4: Plot of average throughput (over 10 ms) as a function of the number of online users

The results plotted above are mostly as expected, except for the spike at the end. If we keep the number of users around 80 or fewer, it appears that the throughput remains relatively constant, as expected. This means that the chat server's performance is not overly affected by either the send rate or the number of users online; it also indicates that we did not overfit our design for one optimization in favor of another.

The more interesting point at the end of the spectrum, when our server operates at nearly maximum capacity. The throughput spikes up about 250% when we have that many users online. We do not have any immediate explanation for this behavior; it is somewhat counterintuitive to have the throughput get better when the number of users increases to maximum capacity. We do speculate that this may be because of uncontrollable fluctuations on the server. We sampled the last datapoint right after the server came back up after its hiccup at 9:00pm the night that the code was due. Since it was likely that the number of students using the server was a lot lower at that point in time, this may have allowed our throughput to jump back to a higher number.

Conclusions

Our ChatServer now operates with full functionality on Amazon's EC2 cloud. In addition to being able to carry out its features and jobs from the first two phases, it can now do so with the support of a database backend. This thus allows it to store and restore state as users log in and log off, as well as provide backup in the scenario of a server crash.

The chat server also operates more realistically, since users are now required to have a login and a password. During our design we kept security principles in mind, taking care to encrypt sensitive information such as the password and prevent attacks such as SQL injection beforehand.

Finally, we were able to perform some performance tests after cloud deployment to make sure that its efficiency was not impacted. Based on the results of our tests, we conclude that it does everything as expected as optimally as we could have made it.