Distributed Chat Server

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

Overview

The chat server that we implemented from phases 1 to 3 now operates with full functionality in the cloud. However, at any one time we only have one chat server running, with all of our clients going to that one server. This introduces at least two disadvantages.

First, it forces us to impose a maximum limit on the number of clients that can be connected to the chat server at any one time. If such a limit did not exist, the number of clients on our one server can explode, causing latency to skyrocket and eventually crashing the server. This is obviously even less desirable than imposing the hard limit. Here we find a way to remove such a limit to allow even more users to connect at once, at the same time without having to worry about the load on the server.

The second problem follows from the above issue. Since we operate over a network on the cloud, the chat server is not 100% stable. It may freeze or crash at any time, whether or not it is too full of clients. This cannot be avoided, but the effect on the connected clients is disastrous. Once the server goes down, all clients are disconnected; from their point of view, this is a massive failure on the server's part. We need a way to rectify this; in the face of a server crash, how can we make it such that the clients can continue with their normal activity?

The most obvious solution is to have more than one active chat server at any time. The number of clients that we can ideally support is then multiplied--a huge gain. If one chat server can safely support 100 clients, then ten chat servers can support 1000. In addition, if a server goes down, we can migrate all its users to another server behind the scenes, and they would not even notice that anything happened.

Considerations

One major issue is the increased overhead in maintaining the multiple servers. Before, we only had to worry about the users, groups, and message sending on one server, and we already had several data structures to keep track of all that information. Now we will have multiple copies of each of these. Obviously, our current implementation will be insufficient and hugely inefficient for our new ideas. What we can make use of is our database, since it will be shared. We also need to add new information into it to keep track of the servers themselves.

Another consideration is load balancing. An equal distribution of users among servers is an idealized vision, but we need a method to actually implement this efficiently. One naive way to distribute the load would be to examine all servers upon user login and redirect the user to the one with the lowest load. This will not scale, as it requires that we make a large number of queries for every single user. Rather, consistent hashing will have to be used. If the hash is well-constructed, then the servers will be closed to equally balanced, which is considered good enough for our purposes.

Related to the hashing is what action to take when one server goes down. All its users must be moved to another server, with more than one possible choice. Which one do we choose? The easiest way is to just redistribute them to a neighboring server. That way, when the server who went down comes back up, we only have to ask its neighbors to get its users back. While this would be a heavier load on the neighbors to take in all the refugees, server failure occurs rarely enough for this issue not to really matter.

One last major consideration is message sending, the most basic functionality that our server should support. It is likely that messages will be flying back and forth across many different servers. When all this activity took place on one server, we did not have to worry about this issue, since our one server had knowledge of all users connected to the system. Here we would have to allow our servers to communicate with each other by initializing sockets among the servers.

Through all of this, the clients should never notice that server switching occurs. That is, if a client's server goes down, the client should connect and log in to a backup server behind the scenes. So the client should not see any "login OK" or similar responses. Likewise, a client should not know that he is logging out from a backup and moving back to the home server.

Database Changes

Our database from the last phase included several tables, all of which we plan to continue using. The Users table kept track of all registered users, along with their encrypted password information, in the form of a salt and a hash. The Memberships table showed all group memberships for the users, while Messages stored messages for offline users. All the different servers will have access to these tables, so that information sharing will be easier.

We will also need to add a new table server_info, keeping track of all the active chat servers at any time. This will help clients know how to connect to the right one. Fields will include the following: Name, IP address, client port, and server port. The names are unique identifiers for each server, and will serve as keys for this table. Clients can look up the right server in this table to find the IP address and port number to use.

There are two port numbers for each server, and usage of the correct one depends on the socket that we are using. For server to server connections, we pull the server port; for server to client connections, we pull the client port number. This information is located on the database so that external servers and clients know how to connect to a particular server. The port numbers may also be supplied at the command line upon server creation, in which case we just ignore the parameters in the database.

Consistent Hashing Scheme

We deal with both the above table and server hashing in our DBHandler class from the last phase. We have a method getServerNames(String username) that returns both a home server and a backup server given the username. This method essentially takes the given String and hashes it to the correct two servers according to the method described below.

We hash using the Java-provided MessageDigest class. We take the username and hash it using the SHA-256 algorithm; this was our arbitrary choice, as we could have used any well-known hashing scheme. We convert the result into a BigInteger, which we remember as the hash value of the username. In the next step, we go through the servers in the database and hash each of the server names in the same way as above. Then we simply sort this list of BigInteger hashes and see where the user fits in. The two largest right above the user (wrapping around to the smallest ones if the user is too high up) are the two servers that we want.

At this point, the method returns the two server names that the particular client should go to. He can then just get the ports, IPs, or whatever else that is necessary to establish a connection.

ChatClient

There are several changes that were implemented in the ChatClient class. The first is that adduser, login and connect are all merged into one login command, which take a --username flag to specify the username. The new login command connects to the correct server and attempts a login with the given username and password. If the username supplied is not registered, the username is registered, which is what the old adduser command did. If the username is registered but the password is incorrect, the client is rejected as usual.

Unlike before, the ChatClient will now connect directly to the database server to determine which ChatServer instance to connect to, using the consistent hashing scheme. To accommodate the partitioning of users across separate servers, each client will maintain a connection to the database to access the table of servers. During operation, each client will first hash its own username to obtain a mapping from unique username to unique id as described above. After hashing, the client will utilize its database connection to obtain information about the servers and then decide which server is responsible for itself.

The client will then connect and log in on the correct server as a single operation by fetching the IP and port number from the database. It first tries the home server. If this connection fails, then it must be down, and so the client tries again with the backup server. After this initial login procedure, the client will interact with the server without any regard to the existence of other servers in the system.

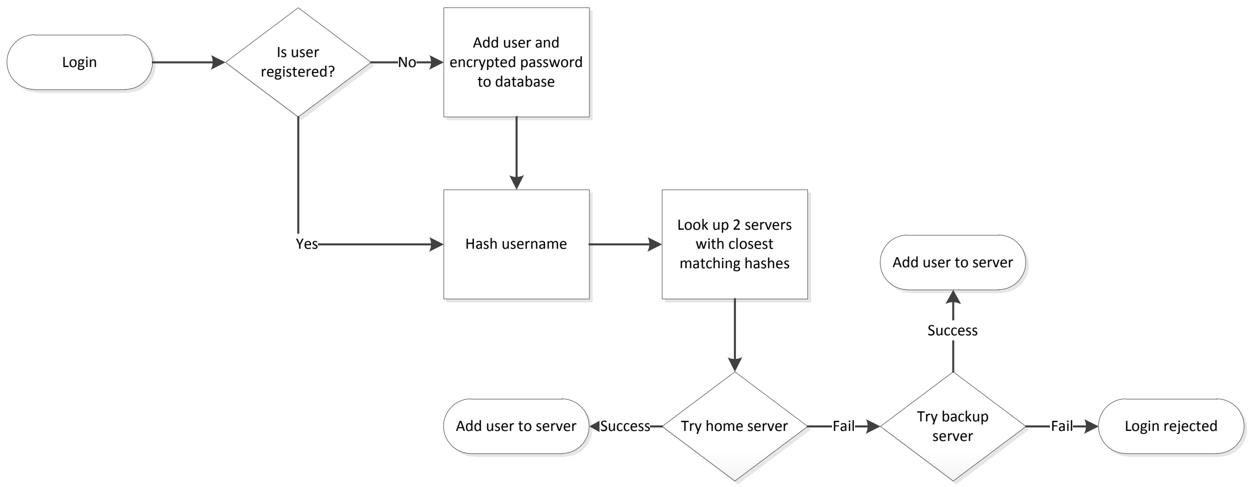

Figure 1: Connecting and logging in

A major change that has taken place in the ChatClient is that it now continually polls its home server, whether or not he is logged on to it. The reason is because he should always want to gravitate toward that, so that if for some reason he is on the backup and the home server comes back up, the client will automatically find out that he should reconnect to home.

We poll every second using a new thread that we create in the client in addition to our two before. While the client polls, he cannot do anything else. So what we do is have this thread alternate between sleeping and waking up. If we ever find that the home server is up and we are not on that server, then we go through the process of disconnecting here and reconnecting on the home server, described below. All of this is, of course, done in the background; the actual user should not notice any of the attempts to switch connections.

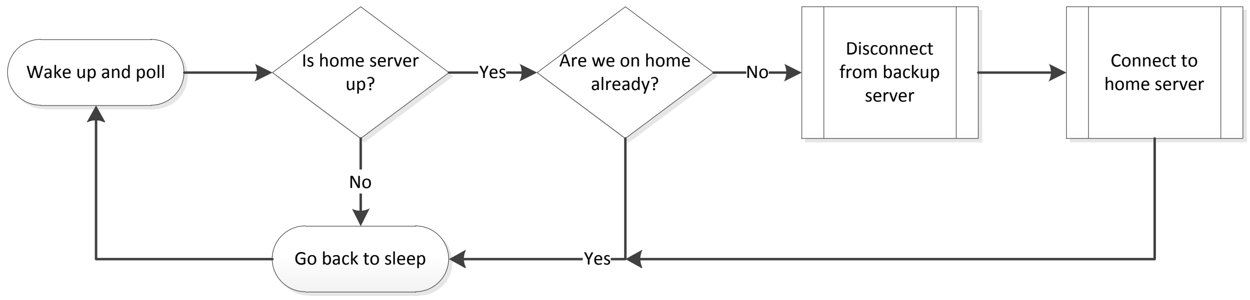

Figure 2: Polling periodically for the home server

ChatServer

The chat servers now no longer need most of their in-memory data structures. Information such as chat groups now longer makes sense to be stored on a particular chat server, since the members might be spread across multiple servers. (We still use our ChatGroup class to handle chat groups, however.) Instead, much of this information must be queried from the database. We can still keep information such as which users are on this particular server at the moment, but a particular server should not know anything about other servers, like who is online.

In starting up chat servers, we keep track of specific information, mostly the fields included in the database table. Each server has its own unique name, IP address, and assigned client and server ports. The name is the identifier, while other clients and servers need to know the IP address in order to find the server. In addition, the port numbers may be stored in the table for external connections, but they may also be supplied from the command line.

For handling cross-server messages, each instance of a chat server will have to maintain connections to the other servers. These connections will be created on server startup. This allows a new server to create the connection with the old servers and allow dynamic handling of additional servers without the overhead of periodic checking. These connections to other servers will be used to pass messages across the different servers. These occurrences happen when a user sends a message to a user on another server. In this case, the server responsible for the source user will pass the message to the server responsible for the destination user. The server connections are then used to maintain communication between the servers.

If the message goes to a group, then the process is much the same as the above. We first query the database to find all users in that group. Then we just repeat the above operation for each of the users that we get out.

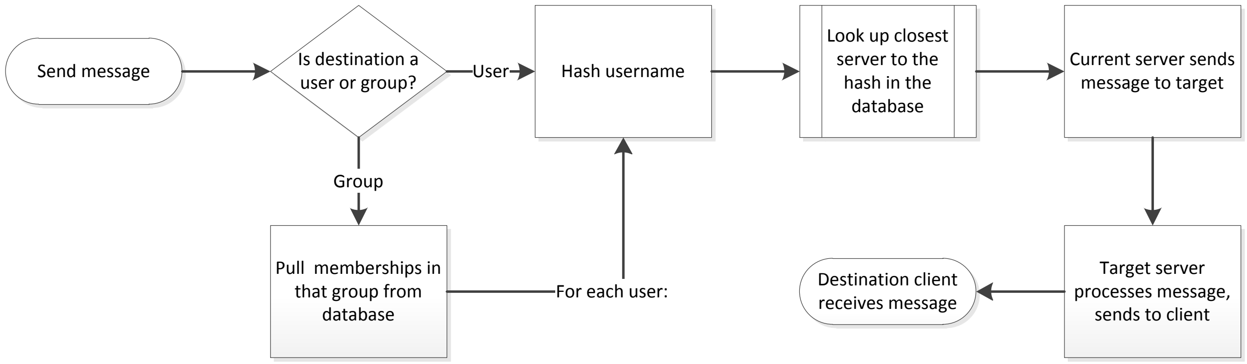
Send message
Is destination a user or group?
User
Hash username
Look up closest server to the hash in the database
Current server sends message to target
Group
Pull memberships in that group from database
For each user:
Destination client receives message
Target server processes message, sends to client

Figure 3: Sending a message between two chat servers

Implementation

In order to obtain the hashing properties such as one-way, collision-prevention, and randomness, the chat server uses the SHA-256 hashing algorithm, which is available as part of the Java API through the MessageDigest class. The hashed version of the client's username is unique because of the collision properties guaranteed by hashing and can therefore be used for calculating the corresponding server. This can be done through examining the server ranges obtained from the database and finding the range that the user belongs to. After matching the server, the ip can be used to connect to the server. By using this method of hashing, the corresponding server will be the same when the client's username is rehashed. This preserves the decidability needed to determine consistent server partitioning among users.

The chat servers open and set up additional sockets to maintain connections to the other servers. Each new socket has two threads to maintain proper operation. One of the these is responsible for reading messages from the socket, and the other thread is responsible for writing messages to the outside server. When a message is received, it is handled as though the source user was on the current server and dealt with accordingly. Conversely, when a message is sent out, the server writes to the socket corresponding to the server of the destination user as though the server was the source client. Since messages are passed among the chat servers, the client is unaffected and the illusion of a single server is preserved.

We created a new class called ServerConnection to handle these new threads and sockets. Each server keeps a list of ServerConnections. If a server connection goes down, this will eventually be found out when trying to use the associated socket. An exception will be thrown, and upon catching it, other servers remove it from their lists. This way, the servers keep track of which servers are up. Because of the way that this is done, there are potential delays between the time that a server goes down and the time that another server finds out. We were fine with this result; it does not matter that everyone knows that a particular server is down as soon as it happens. Rather, they will find out when it is necessary, such as trying to forward a message or another request. On the same note, even when a server finds out that its companion is down, not all other servers will know. That is okay as well, as they will find out when needed.

Because of the distributed nature of the chat server, modifications to group messages are needed. When messages are sent to groups, the groups will now lookup the users in the group in the database, and hash each username to determine which server to send the message to if the user is not on the same server as the source user's server. It then forwards the message using the correct ServerConnection. The same is done for one-to-one messages.

Dealing with failure

By adding additional ChatServers, we can now distribute the clients among our instances so that each ChatServer will be in charge of handling a subset of all users. Ideally, the users are distributed uniformly among all ChatServer instances. Because we have multiple ChatServers, we must now handle the case where one or more ChatServers go down and when the ChatServers come back up. In the first case, we must migrate the users on the downed ChatServer to another ChatServer in such a way that the disturbance on a client's functionality will be minimized. In the second case, when the ChatServer comes back online, the clients that originally belonged to the downed ChatServer should be reassigned to that ChatServer.

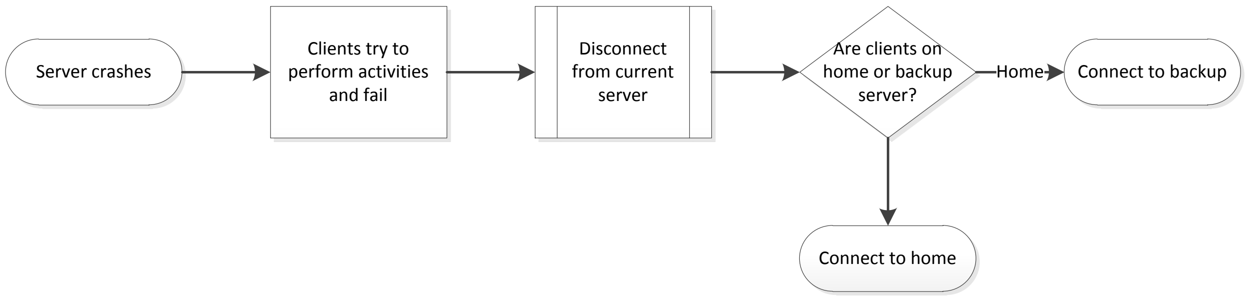
Server crashes
Clients try to perform activities and fail
Disconnect from current server
Are clients on home or backup server?
Home
Connect to backup
Connect to home

Figure 4: Dealing with server failure

When a ChatServer goes offline, our implementation moves the clients belonging to that ChatServer to the appropriate neighboring ChatServer. Here we use our ring of ChatServers as well as our consistent hashing scheme. For all the users that belonged to the downed ChatServer, we hash their username to determine their location within the ring. The user will be migrated to the ChatServer to which it is now closest. This may be the home or backup server, depending on which one the user is currently on. Of course, this assumes that no more than one server is down at any one time (according to the specs).

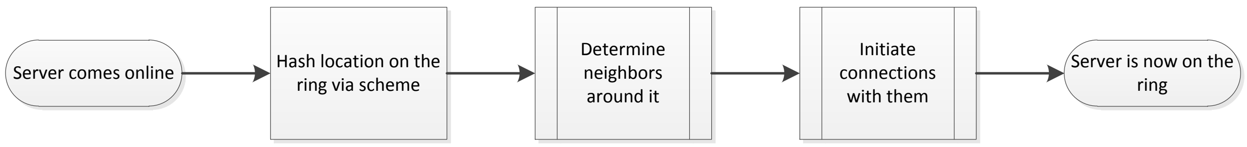
Server comes online
Hash location on the ring via scheme
Determine neighbors around it
Initiate connections with them
Server is now on the ring

Figure 5: When servers come back up

When the crashed ChatServer comes back online, its location on the ring must first be determined. This is done via consistent hashing. On startup, the ChatServer will make requests to connect with all other ChatServers. Clients who belong to this ChatServer will log out of their current server and log in to this one. The ChatServer does not try to reconnect with clients when it comes back up; rather, the clients on back up servers tries to connect with the server they belong to every second.

Testing

We performed the following tests locally first and debugged before running through tests quickly on the cloud. Locally, we started out with one chat server to test old functionality. We then moved up to two chat servers to simplify the testing of crashing since all users should move to one server. We performed the rest of the tests with three servers locally. On the cloud, we started four servers on four different instances and started multiple clients on another instance.

Old tests
A) Login/Authentication
B) Groups
C) Sending messages
D) Readlog for offline messages

New tests
A) Creating new users, storing username and passwords, logging in.
- logging in creates connection to server
- logging out disconnects from server
- logging in with new username and password adds to user and logs the user in
- can log in as different user after logging out
- logging in as old user with wrong password is rejected

B) Groups
- have users on the same server join a common group
- have the above users send messages to each other
- have users on different servers join a common group
- have above users send messages to each other

C) Messages
- have two users on the same server send messages to each other
- have two users on different servers send messages to each other
- crash a server, migrate a users, and have them communicate again

D) Readlog and offline messages
- have a user send to an offline user on the same server
- have a user send to an offline user on a different server
- log in the offline user, readlog, and make sure that the offline messages appear
- make sure that the messages are wiped after a readlog

E) Crashing

- log users in and choose a server to go down
- check if users migrate from server to server
- test if users can still issue commands on backup servers
- test if crashed server comes back up, users move back to it appropriately
- test if users can log in with their home server down

F) Multiple servers
- start up all servers, check for connections between them
- test if users end up in different servers
- test if users can send messages to users on different servers
- test if users can join groups created on another server
- check if same users consistently log in to the same server if the server is up
- users across multiple servers in same group can send to each other

In addition to the regular chat server commands that the command line should take in, we added several more to aid in debugging. In particular, the thread count was extremely helpful in helping us find where runaway threads, deadlocks, and concurrency bugs were hiding:

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

Security

While we brainstormed and thought about implementing security procedures into our server, we never actually got a chance to try it out. Nevertheless, we describe our proposed scheme below.

In order to ensure the confidentiality of our clients, in addition to encrypting their passwords in the previous phase of the project, we will be encrypting messages that clients send using symmetric key encryption. More precisely, we'll be encrypting everything sent between clients and the servers, as well as among the servers. However, we must establish the shared key securely between two parties before we can use symmetric key encryption.

We will use the Diffie-Hellman key exchange scheme to accomplish this. Three parameters, g, p, and g^y mod p are kept "public," hard-coded into the client and server classes. When the client requests a connection with the server, an x is generated on the client side, and g^x mod p is sent to the server. Then x is only known to the client and y is only known to the server, but both can generate g^(xy) mod p, their shared key. We will use this shared key and known encryption functions to encrypt and decrypt future messages between the server and client.

Confidentiality between the servers is easier to implement as we can hard-code the shared key. This scheme is based on the assumption that no adversary can access our servers' code. With access to our servers, an adversary can get the hard-coded value y and decrypt any message our servers receive.

To implement this in Java, we will make use of the KeyGenerator class, the Cipher class, and the cipher input and output stream classes. Given the parameters, we can generate the keys using the

KeyGenerator class specifically for the Diffie-Hellman scheme. CipherOutputStream and CipherInputStream will take as arguments our old input and output streams and a Cipher object. The Cipher object will be initialized once using the shared key, "DES" as the encryption algorithm, "CBC" as the mode, and "PKCS5Padding" as the padding scheme. To initialize a connection, however, we must first use non-cipher streams since we do not have the keys set up yet.

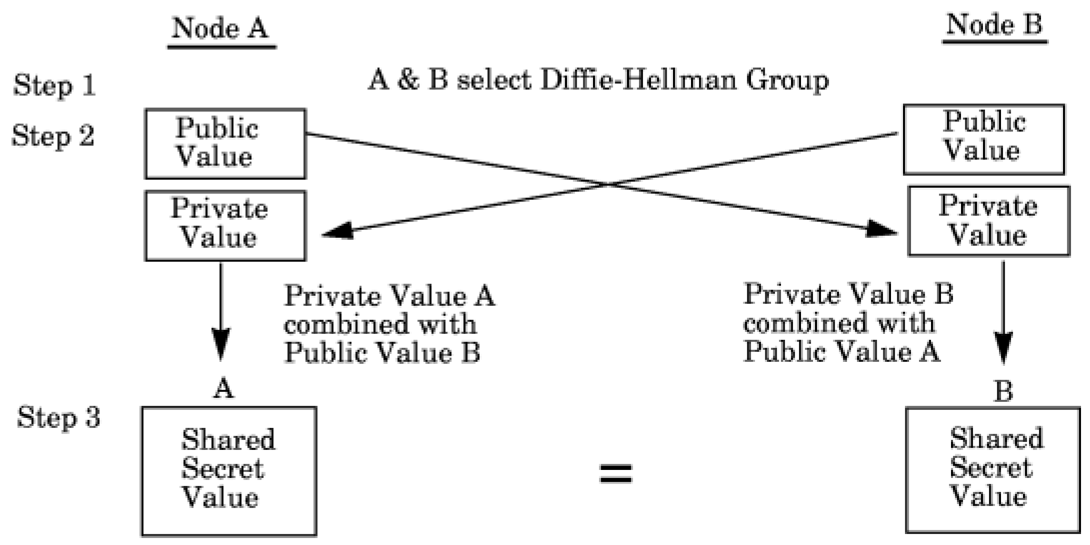
Node A
Node B
Step 1
A & B select Diffie-Hellman Group
Step 2
Public Value
Private Value
Public Value
Private Value
Private Value A combined with Public Value B
Private Value B combined with Public Value A
A
B
Step 3
Shared Secret Value
=
Shared Secret Value

Figure 6: Diffie-Hellman Key Exchange

Conclusions

At this point we now have in our hands not just a simple chat server that can support several clients, but many chat servers that can interact and cooperate with each other to ensure that even more clients can send messages and perform group activity. When we started off in phase 1, all we had was a simple local server that could support a few clients on the same machine. We next proceeded to upgrade it to support networking capability, so that clients on different machines can also connect onto the server. At this point, the server began operating realistically.

Then in phase 3, we added a database backend to the chat server. This now meant that it can store state, and that it can remember information such as users, groups, and offline messages. Users were given a level of security not seen before, by requiring them to log in with passwords. These, along with offline messages to users not currently online, could all be stored in the database for future use. That way, the server would not lose anything if we were to shut it down. Furthermore, we migrated our work over onto the cloud, so that it was truly a networking system.

Finally, in this document we present our design for transforming our humble chat server into a full-fledged distributed system. We can now support many more clients, since we can increase our server count to as many as we want. Along the way we faced challenges such as concurrency, synchronization, network failures, and database consistency. But we were able to solve all of these using clever design techniques and efficient representation schemes. We are thus left with a fully operational chat server with unmatched functionality relative to our prior prototypes.