Architecture of Large Scale Information-Systems **Assignment 3**
CS 4820 Spring 2013 **Horace Chan, Kevin Chen, Shihui Song**
Monday, March 31, 2013 **hhc39, kfc35, ss2249**

1 To run:

On localhost: Sorry but we only included the Amazon optimized .war file. It should be the same (hopefully), but please ask us if you need the localhost one specifically. You can probably download the .git commit file that's timestamped. Afterwards, simply deploy the .war file into the tomcat7 webapps directory. To allow access from outside sources:

- Bypass the firewall for port 8080 TCP
- For the server UDP port that is opened (You can determine this by going to http://localhost:8080/mumble and you'll see a a line "This server's IPP port: IPxxx_portNumber", the 5-digit portNumber is the one you want), also bypass the firewall for this portNumber for UDP communications. Note that this may need to change everytime the app is started.

On Elastic Beanstalk: Add the .war file as a version into one of your environments. To allow communications among the instances within the environment:

- Specify the Elastic Beanstalk security group to accept ALL UDP ports (changeable in the AWS Management Console → EC2 → Security Groups)

2 Work-arounds for problems

ss2249 could only use WebApps 2.5: This means that we could not use @WebServlet notation straight from the Servlet. For some reason, ss2249 could not upgrade to WebApps 3.0, so everybody downgraded. So web.xml was modified so that all requests will be directed to the Servlet, but that caused infinite recursion. Instead, the workaround is: create an empty index.html page in the home directory. Map all requests to that index.html page (which is all accesses to the home directory of mumble) to the Servlet. This works fine both locally and on Amazon Elastic Beanstalk.

Everybody was using Java 1.7 instead of 1.6 (which is what the Amazon Elastic Beanstalk used): Our code depends on Java 1.7. There is a file named java7.config included in our WebContent's .ebextensions directory. This is a file is run automatically whenever the .war file is uploaded to Amazon Elastic BeanStalk. The file downloads java 1.7 and uses 1.7 instead of 1.6.

3 Code

The java code - is also separated into RPC code vs Session Code:

- **RPC:**
  - **MESSAGE** - The message class is an object that represents what is going to be sent for RPC messages: Here is the composition of the various types of message:
    - DELETE messages will have the form:
      - for send: S˜D˜callID˜port˜SID˜version

    - for receive: R˜callID˜port˜SID˜version
    - version number is a boolean bit of 1 = success, -1 = failure.
  * READ messages will have the form:
    - for send: S˜R˜callID˜port˜SID˜version
    - for receive: R˜D˜callID˜port˜session
  * WRITE messages will have the form:
    - for send: S˜W˜callID˜port˜session
    - for receive: R˜W˜callID˜port˜version
    - version number is a boolean bit of 1 = success, -1 = failure.
  * The notations above are:
    - S denotes a SEND message and R denotes the RESPONSE (RECEIVE) of a SEND message.
    - D = DELETE, R = READ, and W = WRITE the type of message as the second variable in the message.
    - $CallID$ is the unique message number. Each $callID$ of the response must match with the $callID$ of the send message.
    - *port* is the port number of the server so that the receiving server knows which port to send the response message to.
    - SID is the session id that is searched for. $SessionID$ is comprise of *sessionNumber˜SessionOriginIPP*. $SessionNumber$ is the number of sessions that have started at that $SessionOriginIPP$ and $SessionOriginIPP$ is comprised of the *Address_PortNumber*.
    - *session* is the complete session information that is to be written.
  * The message is serialized to 512 bytes and de-serialized when a message is received.
  * Our Timeout between RPC Messages is defined here in this file to be 10 seconds.
- **CLIENT**:
  * Whenever a message is to be sent, a client is created. This client has the destination IPP, the message, and it has three main methods (read, write, and delete).
  * It'll send a message to the destination and wait for a response. If it times out or other exceptions, it'll assume that the destination is offline and delete from the memberSet. If it receives a message, it'll add the destination to the memberSet (note that in Java ConcurrentHashMaps, a removal of a non-exitent item does not throw an error and a put of an existing item simply overwrites the old version).
  * Every message sent must have a corresponding callID back.
- **SERVER**:
  * The RPC Server is created upon servlet creation, and persists throughout the servlet's lifetime.
  * It runs in a separate thread apart from the Servlet, but accesses both the Session Data Table and the Member Set.
  * The servlet's IPP is created by concatenating the servlet's external IP address (which cannot be found from the RPC Server Datagram Socket function getLocalAddress. We used a Stack overflow solution to deal with this.) with the port number assigned to the RPC Server socket.
  * The RPC Server code is straightforward, mostly taken from the project description. It accepts RPC Client requests from other servlets, and responds according to protocol. The only changes come from:
    - Implementing Crash: The RPC Server code can potentially exit before and after receiving a message, as well as before sending a message.

        - Member set management: In the cases of writes, the primary and backup IPP of the session to be written are added to the member set (provided they aren't null and not equal to yourself). We also add the RPC Client's IPP to the member set.
      - Client requests with a call ID less than a previously seen call ID from them are not responded to nor change the state of the server, because this means the request may be old.
- **Session:**
  - **TERMINATOR** (this code is unchanged from part a)
    - A thread that removes expired session from the concurrent hashmap table of sessions.
    - This runs every 2 minutes, as every cookie expires in 2 minutes.
    - Every SERVLET owns its own terminator, and it is synchronized.
  - **SERVLET:**
    - The main instance and logic for this project, it is called in both GET and POST instances. GET is for first time access and refresh, which we determined were very similar. POST is for Log out and Replace. It is also for the crash function.
    - When there is a crash, doGet and doPost methods both immediately return. A flag is also turned so that the RPC Server knows to exit its loop of servicing RPC requests.
    - There is a concurrent hash map for the sessions. We chose this over a priority queue because we believed it better to be better for lookups.
    - There is a concurrent hash map for the member set. We chose this because we need to keep track of whether an older messages is being sent to the RPC Server because of delays in the network.
    - The cookies passed to the client is a regular Java cookie with the name CS5300PROJ2 and the value as the sessionId, version, and the location. In the case of Log out, a max age of 0 is sent.
    - The code is divided into many sub-functions that prevent repetitive code. The main difference that we've found is the different actions on what to do if a session is found. This is resolved with enums.
    - At the end of every doGet and doPost, many attributes in the Servlet Context are set so that information can be sent to our main jsp page.
  - **SESSION:**
    - In addition to its inherent values as a cookie, it also has a string sessionId, a string message, location, and a long end (expiration time).
  - **IPP:**
    - A basic object that contains the String IP and String Port.
  - **Location:**
    - A basic object that contains 2 IPP's: One for the primary IPP, which is NEVER null, and the other for the backup IPP, which may or may not be null.
  - **SessionId:**
    - A basic object that includes a session number and the IPP at which this session originated from.
  - **Cookie:**
    - A basic object that extends Cookie. It includes the session ID, the version number, and the location of the session information.

The HTML - renders the front-end

- CS5300PROJ1index.jsp - The html is rendered per response. There are various getServletContext requests to fill out the information to be updated, such as the message, the expiration time, and the address. This is called every time there is a request, and it is sent through the requestDispatcher.

- index.html - A blank placeholder for the workaround above.

4 Concurrency

- The session table is a concurrent hash map.
- The member set is also a concurrent hash map, as it can be accessed by both the servlet and the RPC Server.
- For all corresponding read/write to the hash map, there is synchronized (hashmap) surrounding the statements.
- We agreed on an ordering of the objects to prevent deadlock: the session table must be locked first before the member set is ever locked.

5 Extra Credit

None :)

Kevin, Horace, and Sweet