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: 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 with outside ports:

- Specify the security group for all ports

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
    - 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
      - 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.
  - **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**:
- **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.
  - There's a concurrent hash map for the sessions. We chose this over a priority queue because we believed it better to be better for lookups.
  - The sessionId is UUID (in the future with multiple servlets, the sessionID will be also include the servlet number, which is simply zero in this case). The version number and the location metadata are so far ignored in this project.
  - The cookies passed to the client is a regular java cookie with the value as the class name of CS5300PROJ1SESSION and the value as the sessionId, the version number, 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.

- **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).

The HTML - renders the front-end index.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.

4 Concurrency

The session table is a concurrent hash map. For all corresponding read/write to the hash map, there is synchronized (hashmap) surrounding the statements.

Cheers, Kevin, Horace, and Sweet