# Project 1b: Scalable and Available Website

Alice Renegar arr233
Spencer Steel shs257

March 26, 2015

## 1 Solution Structure

### 1.1 Session Structure

**Session IDS** consist of a `long sid = <serverID, sessID>` where `sessID` is locally unique to the server the session was created on and the `serverID` is the integer representation of the IPV4 address. These two 4 byte integers are concatenated into an 8 byte long.

Sessions also contain a **version number** `VN` where each `VN` increases. `VN` rolls over to 0 after `MAX_INT`. A version number of -1 indicates a retired session. No non-retired session may replace a session with a higher or equal version number or a retired session, and all retired sessions are considered equal. This is considered sufficient for this assignment.

The last two variables in a session are **exp**, the expiration time in milliseconds, and the message which can be up to 485 characters in the ASCII range (with some limits set by `OWASP` for security). This is somewhat defined by UDP size (we could have fit an additional 69 characters by reducing each to the 7 bit ASCII value, but we thought 485 was enough.

Messages can include `img` links and other elements for fun, within reason (`OWASP sanitized`).

### 1.2 Cookie Structure

The cookie structure is done with `JSON` for code and cookie readability. For a practical application the values of the cookie would be encoded with a higher base and using deliminators instead of labels, for space. There is no distinction between primary and backup servers, the local server is always tried first if it appears. The retrieved cookie is assumed to be at least the version number in the cookie. The version number in the cookie is used to debug. It would be removed in a 'real' application.

### 1.3 RPC Message Structure

We have three main message request types `read, write, and merge`. `Read` and `write` are initiated by client connections and are used to fetch and store sessions in remote servers respectively. Both calls favor servers where the session was previously stored, but both over request and take the first response. After 3 seconds without a response the calls time out and servers are set to `DOWN`. We chose to implement the project this way to make it fast, at the cost of efficiency. However with insufficient servers this extra traffic would slow the system down. We did not anticipate that we would generate much traffic.

`UDP` messages are represented as a `< callID int, operationCode byte, message>`. The operation code can be a request (`READ, WRITE, MERGE VIEWS` or a response (`FOUND SESSION, SESSION NOT FOUND, STORED SESSION, NEWER VERSION IN TABLE, MERGE VIEW RESPONSE`).

Depending on the operation code the messages appended to the `UDP` packets are as follows:

1. `READ: <sessionID long>`
2. `WRITE: <sessionID long, exp long, vn int, msg length short, msg >`
3. `MERGE VIEWS: <length short, <serverID int, time long, status byte> list>`
4. `FOUND SESSION: <sessionID long, exp long, vn int, msg length short, msg >`
5. `SESSION NOT FOUND: <>`
6. `STORED SESSION: <>`
7. `NEWER VERSION IN TABLE: <>` The session data wont be needed by the server anymore, so we don't return information about the new version
8. `MERGE VIEW RESPONSE: <length short, <serverID int, time long, status byte> list>` This can hold up to 38 servers, but prioritizes severs that are `UP`. We did not anticipate having more than 38 active servers.

## 1.4 Session Expiration Guarantees

Sessions are guaranteed for an hour, which is sufficient for a 'fake' application.
**Delta:** because sessions received via UDP are not given a new expiration time we need to primarily worry about the `UDP` response time during reads (sent in parallel) in delta. This is anticipated to be 3 seconds maximum, with an additional 50 milliseconds to reach the point of the code where the expiration timeout is checked and for system clock alignment.

## 1.5 Expired Session Collection

Garbage collection is done once every 5 minutes, in an effort to test code. If this were in use we would set it to once a day. It is a background task.

## 1.6 Gossip Frequency

The gossip protocol is carried like the project specifications discussed. In our code submitted and while testing GOSSIP_SECS is set to 30 seconds. We realize that in a larger system this could be too often causing issues, however for such a small application and for a testing environment we had it set to 30 seconds.

### 1.7 Debugging Output to Hosted Page

Much of this output reduces parallelism by read-locking portions of tables. Set `DEBUG` to false in the session handler to remove this.

### 1.8 Implementation breakdown

***for further details please see javadocs included at: docs/index.html***

## 2 AWS Elastic Beanstalk

### 2.1 Setup Procedure

To get our application running on AWS Elastic Beanstalk follow the steps below:

1. Create and Deploy War file. There are two ways we did this:
   - Create then deploy VIA AWS Management Console
     (a) Create War file – [warfile].war via Eclipse or command line
     (b) Log into AWS Elastic Beanstalk and create/configure new environment
     (c) Upload war file to environment.
   - Deploy to AWS via Eclipse
     (a) From Project to deploy, select AWS -¿ Deploy to beanstalk
     (b) Select new environment or existing beanstalk environment.
     (c) Eclipse deploys to the environment with incremental update enabled.
2. Ensure environment has property security groups to allow all UDP incoming traffic from all IPs. (Without this step, the servers will not be able to merge views or sent session information)
3. Once beanstalk is up and running you can view the site at the given beanstalk URL with "/SessionHandler" appended to the end of it. See below
   - Beanstalk gives url [ProjectSpecificEnv].elasticbeanstalk.com
   - URL to view our project: [ProjectSpecificEnv].elasticbeanstalk.com/SessionHandler

### 2.2 Testing Resiliency

Testing resiliency is an important part of this project. We tested for resiliency through the following steps:

1. In beanstalk environment configurations set minimun servers to at least 2
2. Connect to the environment from several different clients
3. Verify which specific servers are executing client requests

4. From AWS EC2 Management Console, right click on specific instance and terminate the instance.
5. Check clients that were connected to the terminated server and extend their session.
6. Verify their session was persisted and that the cookie updated to new servers, or the null server if only two instances were running.
7. After a short time, beanstalk should create a new instance to ensure that the minimum umber of instances are running.
8. Once the new instance is up and running, clients should be able to connect to it and it should be integrated into the environment.

Following the above mentioned steps, we were able to feel confident that our database is 1-resilient.

## 3 Explanations

### 3.1 SimpleDB Races

We chose to allow races to happen and not retry gossip exchanges with the database. With the gossip protocol we set up, overwrites can happen, but the overwritten data will still be in the server. This data will be passed around to other servers and eventually will be placed in the database.