CS5300-Project1b: Scalable and Available Web Application

Giri Kuncoro (gk256), Yihui Fu (yf263), Shibo Zang(sz428)

1. Overall Structure

There are four parts of our solution:

- Remote Procedure Call (RPC) infrastructure
- Session Management Logic
- Installation/Reboot Scripts
- User Interface

1.1 Remote Procedure Call (RPC) Infrastructure

The communication between different nodes are through RPC call. There are two types of RPC call: session read and session write. Session read call is used for retrieving session information from a particular node, and session write call is used for updating session information in different nodes and determining which nodes store the particular session.

1.2 Session Management Logic

The core of our solution is the session management logic in SSM protocol. It defines how our application works, such as how to store user session in the server side, how to replicate and backup sessions in the distributed system and make the system K-resilient.

1.3 Installation/Reboot Scripts

The installation script is used to do preparation jobs in each EC2 instance like installing dependencies, set up configuration files etc., and the reboot script handles the situation when one server crashes. The scripts are using AWS S3 and SimpleDB to centralize the process and launching the application.

1.4 User Interface

Interaction for user to perform `replace` , `refresh` , and `logout` action, and see the updated session information. This is implemented using Javascript, jQuery and Bootstrap framework.

2. Formats of cookies and RPC messages

For cookies, the delimeter is #, and the format is: `SessionID#versionNumber#locations`. This is translated with `_` on client side.

More specifically, for session ID, the delimiter is also #, and the format is: `serverID#rebootNum#sessionNum`. This is also translted with `_` on client side.

For RPC messages, the delimiter is `_`, and the format is: `CallID_OperationCode_SerializedSession`. The serialized session object is just concating everything in session by `#`.

3. Explanation of source files

3.1 Java package structure and class

`proj1b.rpc` package
- RPCClient.java: Perform session read and session write operations. - RPCServer.java: Listen on the requests sent by rpc client. - RPCConfig.java: Configuration about rpc clients and rpc servers. - RPCStream.java: Marshall and unmarshall serialized session objects.

`proj1b.servlet` package
- BackgroundThread.java: Another thread running RPC server. - proj1bServlet.java: The controller that handles the logic of application.

`proj1b.ssm` package
- Session.java: Class that holds session related information, such as sessionID, version and message. - SessionCleaner.java: Garbage collector that cleans up expired session on `sessionTable`. - SessionInServer.java: Class to find which node found the sessionData from and forward to session controller that currently handles the session. - SessionManager.java: Class that holds sessionData in sessionTable using sessionID and version as keys, to keep old session data.

`proj1b.util` package
- Constants.java: Global constants to configure the application parameters and path. - Utils.java: Utility helper functions to initialize the app and getting related info from the kreatifile system.

`proj1b.test` package
- Several helper test classes for unit testing and integration test.

`WebContent` files
- index.jsp: Main page where user operates the app features and see the relevant information. - logout.jsp: Page when user decided to logout, user can login back by clicking the link here. - error.jsp: Page when system is failing because not enough running instances to support F ressilient. - assets folder: Various Javascript, CSS sheet, images, fonts to build up the page.

3.2 Important implementation remarks

- **RPC server thread cannot be started by servlet**, since servlet won't be initialized until the first HTTP request from client. This will cause issue since servlet will call `sessionRead` and `sessionWrite` that expect response from RPC server, but the server is not ready. The solution is to start RPC server in `BackgroundThread` class which implements `ServletContextListener`, this will make sure RPC server starts when the application is started by Tomcat server.

- **`SessionManager` is implemented using singleton pattern**, it involves one class which is responsible to instantiate itself, to make sure it creates not more than one instance and in the same time it provides a global point of access to the instance. This should guarantee one hashmap that stores sessionData in one node.

- **Garbage collector is started by scheduler**. Since RPC server gets the `SessionManager` instance, which initialize garbage collector, this will cause issue since there's no data yet to be cleanedup. Thus, by using scheduler, the garbage collector is delayed to start.

- **Input validation is handled by the client page** before sending the message to server, to make sure the provided characters do not break our encoding/decoding and marshalling/unmarshalling implementation (i.e. `#`, `_`, empty spaces, null message are cleaned up first).

- **JSON request/response for communication between client and server**. Instead of traditional JSP that loads new html for each update, by performing AJAX call for each action button, and getting JSON response back from server to load necessary contents, we are able to improve the client page performance.

4. Changes for exra credit

4.1 Supporting F > 1 Failures

Our application complies with the SSM protocol. There are two parameters we need to configure before the instances run: `N` and `F`. `N` is the number of instances we have on hand, and `F` is how many nodes can crush down before the system fails. From these two paramters, we are able to compute other parameters as folows:

```
R = F + 1
WQ = F + 1
W = 2 * F + 1
```

Our system support F > 1 failures successfully since we are not hard coding anything while programming this system. And we tested our system successfully when `N = 5, F = 2, and N = 7, F = 3`.

If you want to test our application, try to modify `N` and `F` in the `launch.sh`.

4.2 Installation Script Failure

We should be able to recover when installation script fails during the instance initialization. So when the instance is up, we could just run `reboot.sh` to rerun the `install.sh` and increment the reboot number. As `install.sh` might stop anywhere when it fails, it's important to handle cases as below: - Ignore writing to simpleDB if the domain already contains N instances of IP and serverID pairs - Ignore creating `rebootNum.txt` to retain the current reboot number - Ignore - Use `sudo` to run commands with root permission, since when instance is up, it's going to run script as `ec2-user`. - There's no harm in reinstalling Java and tomcat if it's already been installed - The necessary files that Java code access are having `777` permission, so the normal `ec2-user` can have access without issue.

Demonstration of F=2 Resillient

If `F = 2` and `N = 5`, then `R = 3`, `WQ = 3`, and `W = 5`. We need to set up 5 instances and run the `install.sh` and `launch.sh` in each one of them. To prove that our system is 2-resilient, please see the snapshots we take. We follow the following steps:

```
1) Connect browser to the public DNS name of Server 0
2) Type a message into the text field and click 'Replace' (screenshot 1) - Should see
 the new message with the same session ID and new version number
3) Reboot Server 0 and Server 1 without running reboot script
4) Connect browser to the public DNS name of Server 2 (screenshot 2) - Should see the
 same message with the same session ID and new version number
5) SSH to Server 0 and execute the reboot script with sudo
6) Connect browser to the public DNS name of Server 0 (screenshot 3) - Should see the
 same message with the same session ID and new reboot number and same version number
7) Reboot Server 3 without running reboot script
8) Connect browser to the public DNS name of Server 0 (screenshot 3) - Should see the
 error page because we have three nodes failing (1,2,3)
```

Since time out logic is basically the same no matter `F` equals what, there's no need to test session time out here.

How to run

Launching instances

1.) Configure AWS credentials file located in `~/.aws/credentials` using following format:

```
aws_access_key_id = YOUR_AWS_ACCESS_KEY_ID
aws_secret_access_key = YOUR_AWS_SECRET_ACCESS_KEY
```

2.) Open `launch.sh` and configure the parameters on top of the file as below:

```
# number of instances to launch, default is 3
N=3

# resiliency to maintain, default is 1, but F>1 is supported
F=1

# S3 bucket name to bring war file and other stuffs in
S3_BUCKET="S3_BUCKET_NAME"

# keypair to ssh instance, important for reboot process
# .pem extension not required
KEYPAIR="proj1bfinal"
```

Make sure to provide the keypair name without `.pem` extension and the S3 bucket name. `N` and `F` have to be valid numbers, i.e. `N = 2F + 1`.

3.) Open `install.sh`, then configure AWS credentials and S3 bucket name on top of file as below:

```
# AWS credentials to connect with aws cli
AWS_KEY="YOUR_AWS_ACCESS_KEY_ID"
AWS_SECRET="YOUR_AWS_SECRET_ACCESS_KEY"

# S3 bucket name to bring war file and other stuffs in
S3_BUCKET="S3_BUCKET_NAME""
```

Make sure credentials provided here are same as the one in `~/.aws/credentials`. This is important since simpleDB manages the tables for same AWS access key. Also, provide S3 bucket name as the one configured in `launch.sh`.

4.) Configure the `default` security group on AWS management console, to open all necessary ports, as below:

- Port 80: for TCP
- Port 8080: for Tomcat
- Port 5300: for RPC communication
- Port 22: for SSH
- Port 443: for HTTPS
- All ports for UDP traffic

5.) Configure the bucket policy on the provided S3 bucket name, allow everyone to download/upload files. The cofiguration shouldn't matter as long as the provided AWS credentials own the bucket and have S3 full access policy.

6.) Execute the `launch.sh` to launch instances by typing below: `$ ./launch.sh` Make sure `launch.sh` is executable by typing `chmod +x launch.sh` if the script is not.

7.) Go to AWS EC2 console and N numbers of EC2 instances should start initializing. Find the public DNS of each intance and register the domain at `bigdata.systems`.

8.) Once EC2 instances are running and completed the checks, open up browser and locate one of the instances domain at `proj1b` url, e.g.: `http://server0.sz428.bigdata.systems:8080/proj1b` The project's main page should be opened, enjoy!

Rebooting instances

1. Navigate to AWS EC2 console, select instance and press reboot. This will reboot the instance without running tomcat. Don't reboot by stopping and starting the instance, since this will change the instance's public DNS and we need to register the domain again.

2. Wait for couple of minutes until the instance is up. When it's ready, SSH as `ec2-user` into the instance using the same keypair when launching the instance: `$ ssh -i <keypair name> ec2-user@ec2-123-123-123.aws.amazon.com` Easy way is to press `connect` button on the instance in EC2 console, and copy paste the SSH command into the terminal.

3. Execute the reboot script in `/var/tmp`: `$ /var/tmp/reboot.sh` Reboot number is now incremented and tomcat service should be started again. Navigate to the `proj1b` url to start playing with the project again!

Acknowledgements

```
http://research.microsoft.com/pubs/74713/ssm-nsdi.pdf
Robert's hello example
```