## Lab 1

Asterix and the Olympic Games

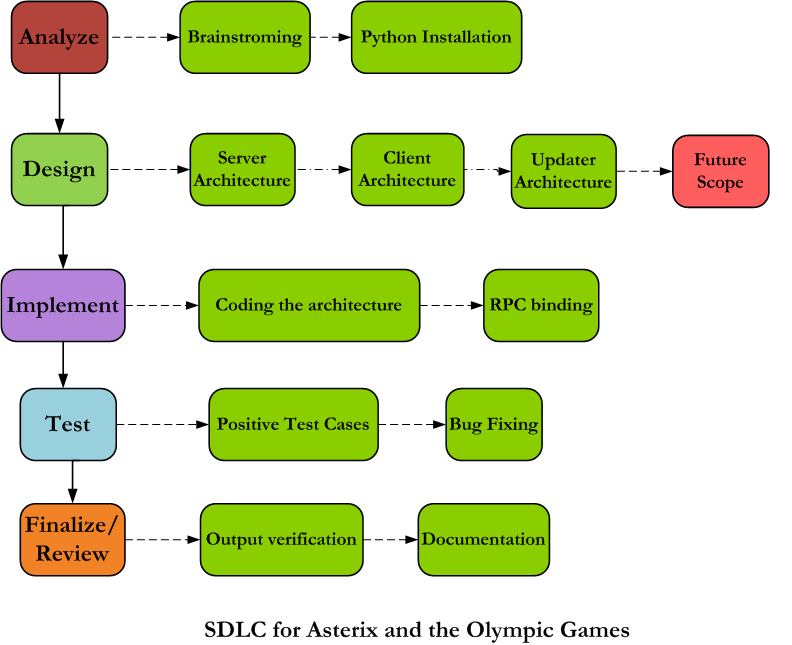
**Ashish Jain (ashishjain@cs.umass.edu)**

**Sweetesh Singh**

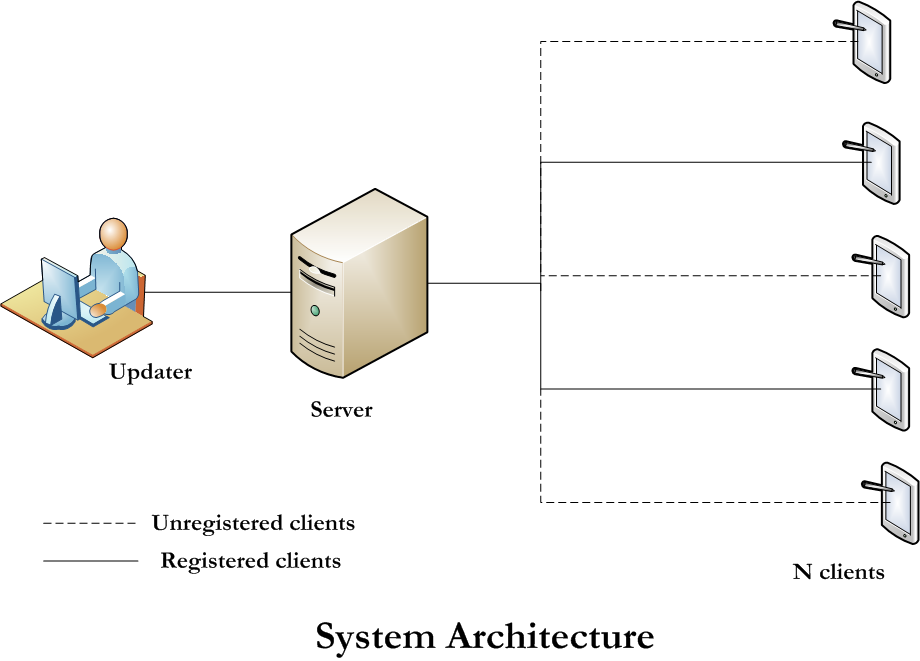
**Sinthu Muralitharen**

**Introduction:** Asterix and Olympics game (Winter Olympics Edition) , in this release we have developed a score viewing system. This involves an updater who updates the server with the cores and these scores are used by few game viewers, which will be denoted as clients. The system is designed for the efficient score updates for the clients based on the conditions and the problem statement. Section1 discusses design we followed, section 2&3 gives overview of system and software architecture. Section 4 gives detailed list of test cases for our system.

**Section 1: Design**

****

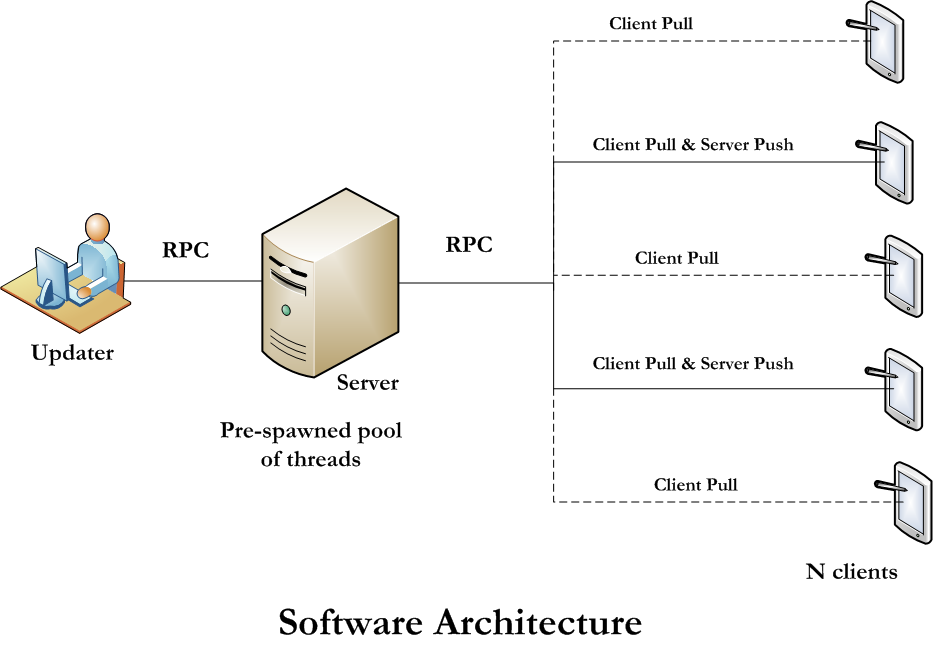
**Section 2: System Architecture**

****

We used the waterfall model of the SDLC to design the system. After thorough analysis of the problem statement, we decided to use the python language and its RPC to establish the client-server architecture. The design phase involved identifying the system architecture and the software architecture for the given requirement and the possible extension for the system in future. The designed software architecture was implemented, i.e the client server, update server and RPC through which they communicate with each other. Effective test cases and scenarios were identified and the code was tested for bug and efficiency. The output was verified and all the phases of work are documented for maintenance and development in future.

**Section 3: Software Architecture**

Our registered clients have capability of periodically pulling the scores too.

****

**Section 4: Test Cases**

In real scenario, we manually give input. Test cases based on that are explained below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **Test Scenarios** | **Priority** | **Input** | **Expected Output** |
| **1** | **Presence of update window** | **1** | **-** | **Update window to be opened** |
| **1.1** | **Enter options to update 1 to update score or 2 to update medal tally** | **1** | **1 to update score 2 to update medal tally** | **Next question to be triggered** |
|  |  |  | **Any other input** | **Error message to be popped** |
| **1.3** | **Enter team name as 'Gauls' or 'Rome'** | **1** | **'Gauls' or 'Rome' in exact format** | **Next Question to be triggered** |
|  |  |  | **Any other value than specified format** | **Error message to be popped** |
| **1.4** | **Enter the event type** | **1** | **Event type in exact format** | **Next Question to be triggered** |
|  |  |  | **Any other value than specified format** | **Error message to be popped** |
| **1.5** | **Enter the Score** | **2** | **Valid Number** | **1) An update in the server,**  **PUSH UPDATE**  **DONE UPDATE**  **Should be displayed  2) Update gets reflected in registered clients.  3) Periodically updated to unregistered clients and registered clients.** |
|  |  |  | **Any other value than a valid number format** | **Error message to be popped** |
| **1.6** | **Enter options to update 1 to update score or 2 to update medal tally** | **1** | **1 to update score 2 to update medal tally** | **Next question to be triggered** |
|  |  |  | **Any other input** | **Error message to be popped** |
| **1.7** | **Enter team name as 'Gauls' or 'Rome'** | **1** | **'Gauls' or 'Rome' in exact format** | **Next Question to be triggered** |
|  |  |  | **Any other value than specified format** | **Error message to be popped** |
| **1.8** | **Enter the event type** | **1** | **Event type in exact format** | **Next Question to be triggered** |
|  |  |  | **Any other value than specified format** | **Error message to be popped** |
| **1.9** | **Enter the medal type** | **1** | **Event medal type in exact format** | **Next Question to be triggered** |
|  |  |  | **Any other value than specified format** | **Error message to be popped** |
| **2** | **Presence of Server** | **1** |  | **Server is ready** |
| **3** | **Presence of Client** | **1** |  | **Client Process is running status is displayed for every client created** |

**Section 5: How it Works**

1. As we can see in Section 2&3, updater is sending the updates to server and server updates it to client depending on the architecture of client i.e. client pull or server push.
2. In case of client pull, we have set the specific time (3seconds) after which client will pull the medal and scores for all the teams, from the data structures of server and display it on client side.
3. In case of server push, we immediately push the update via Server to the client even without the pull request. A pull client can also receive the push update in our design.
4. Pickle Serializer is used to transfer the data over the network.
5. Server objects are registered onto the Nameserver running on the network. Client and update process can access these objects and call their methods.
6. Events considered: Snowboard, Curling and Skating. Teams considered: Gauls and Romans. Event registered for server push updates: curling.
7. For implementing read write lock, I have used open source implementation of it in python.

**Section 6: Design Tradeoff considered and made**

1. Most important design decision was to run client pull and server push architecture on the same client. Currently, it is registering only for ‘curling’ event, but we have made our design flexible enough to incorporate registration for any number of events.
2. We have split the server code into two classes. One part can only read the scores and medal tally from data structures while other class have methods to modify them. Therefore, client has access to first class while update server access the second class.
3. Our server is using the pre-spawned pool of threads to handle client and update server requests. Once a task is submitted to the server, it pick up a thread from the pool and allocates it newly arrived job. It saves us the time for creating, deleting the thread. However, we have to be aware of the traffic limit to create such threads. We might have many threads waiting to be run. We gave preference to performance instead of client starvation in some cases.
4. We have multithreaded each client. Each thread will periodically update its score and medal tally through client pull mechanism.
5. We have designed the locking mechanism for synchronization in such a way that it allows all reads to happen simultaneously. However, only single write can happen at a time.

**Section 7: Future Improvements and Extensions**

1. Currently many of the test cases are not handled in the system. For example, if our Server goes down than client has no method to know about that. It gets hanged currently. Same is with the update server. Therefore these failure semantics needs to be incorporated.
2. Some of the inputs not in the correct format give no error on client and server. Server and client just ignore such input without displaying error messages. Hence they should be made more interactive.

**Section 8: How to Run Program**

Instructions are also present in Readme.txt in the code folder. Maximum of 6 clients will run. It can be increased by modifying startclient.py. However, for testing this assignment, we fixed it to atleast 6, as asked in the assignment.

**Instructions**

1. **Run the nameserver**: python nameserver.py

(**Caution:** In case nameserver is running, this will thrown an error. Go to step-2, you are not required to run nameserver.)

2. **Run server**: python server.py

3. **Run client**: We have six different directories: client-1, client-2, client-3 ... client-6. Go inside each directory and execute "**python client.py <arg>**"(Note: arg will be "1" if you want to register that client for curling event or "0" if you don't want to register.) Output will be displayed on terminal. Writing in file was not possible because daemonLoop is running.

4. **Run updateserver**: python updateserver.py

(**Note:** input.txt should be in the same folder as updateserver.py. It contains predefined medal and score input for the events)

5. **Output:** Server output will be displayed on terminal with informative messages. Client output will be seen on different terminals opened for the clients. You can run clients from different edlabs machine.

**Section 9: Test Ran**

1. We ran the system by giving input values in wrong format. For example, in case of score, we gave some alphabetical string. Our architecture takes care of that by throwing an error or exception.
2. We registered clients to different events and sent the score for one of the specific event. This was done to check whether only the registered client for that event receives the update.
3. We ran more number of client process than threads spawned on server. It blocked the clients which are above a specific thread value. This was done to check the reliability of thread spawning on server side.

**Section 10: Performance Results**

We are using pickle serializer to pass data over the network. Latency is very less because pickle is known for fast data transmission. However, it has security vulnerability but since we are not dealing with security threats as of now, pickle seems as a reasonable option. We can change it to other serializer option in future if security requirement comes up.

**Latency of Client pulling the scores:** Latency for client pulling the score varies each time the experiment is conducted with different number of clients. Sometimes even on increasing the number of client, the latency decreased. This can be widely attributed to load on the edlab server. But the time taken for serving one pull request is on average close to: 0.005 seconds.

**Latency of Update server in pushing scores:** 0.0255 seconds. This shows very less increase to 0.0467 seconds by increasing number of registered clients to 10. The reason can be that we are using light weight pickle serializer.

To be precise, we didn’t get much noticeable difference in performance by increasing the clients. However, we have not run 20-50 clients, so that we have to do in future steps.