Lab 2

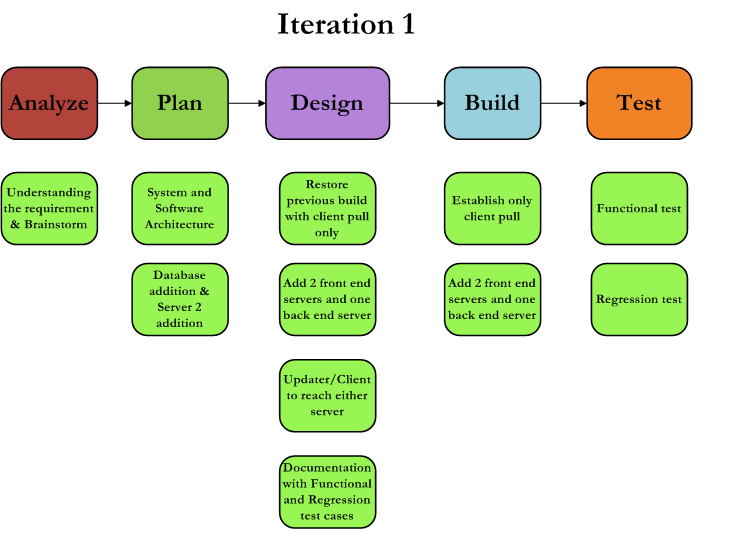
Asterix and the Olympic Games

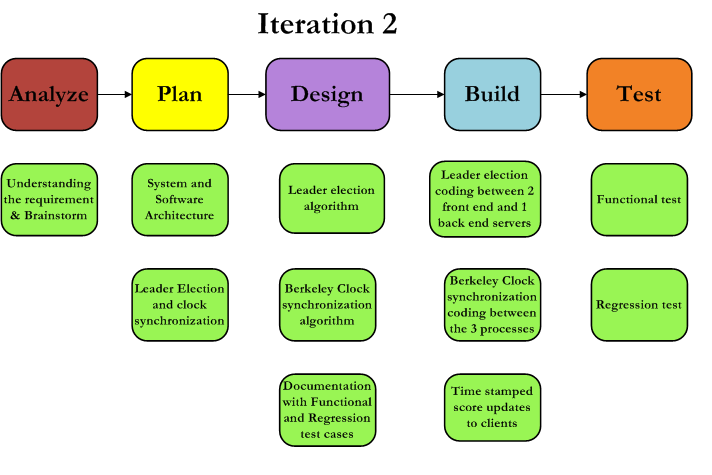
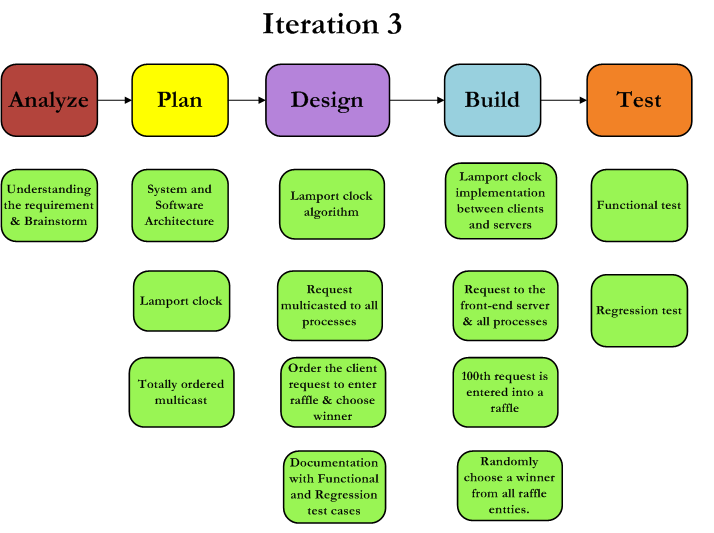
Ashish Jain

Sinthu Muralitharen

**Section1. Introduction**

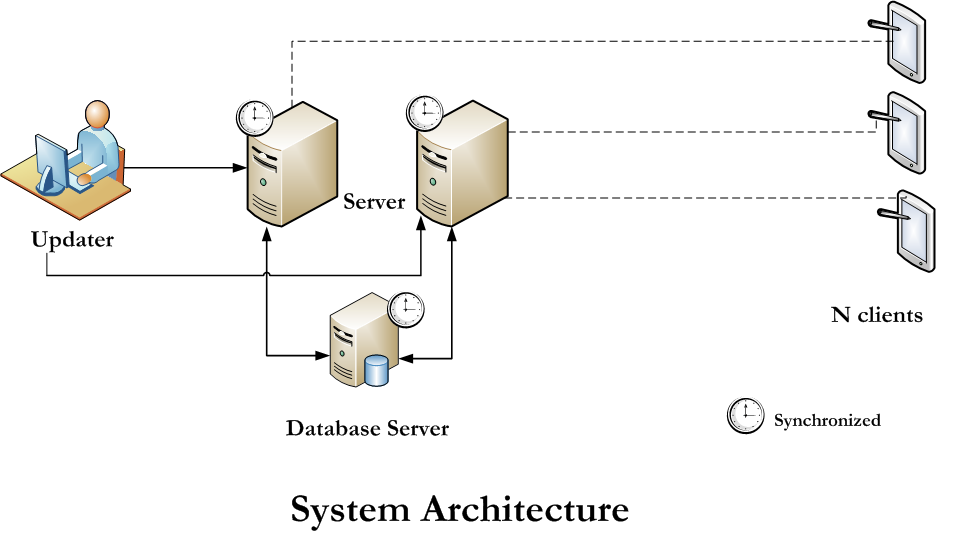
Asterix and Olympics game (Summer Games Edition**)**, in this release we have developed a duplicate server (server 2) to handle the surge in the number of clients accessing the original server (server 1). The system has two front-end servers that receive updates from the updater and requests from the clients. A backend database is present in the system that receives read or write requests from the front-end servers. The system is designed for the efficient score updates for the clients based on the conditions n the problem statement.

**Section2. Design**

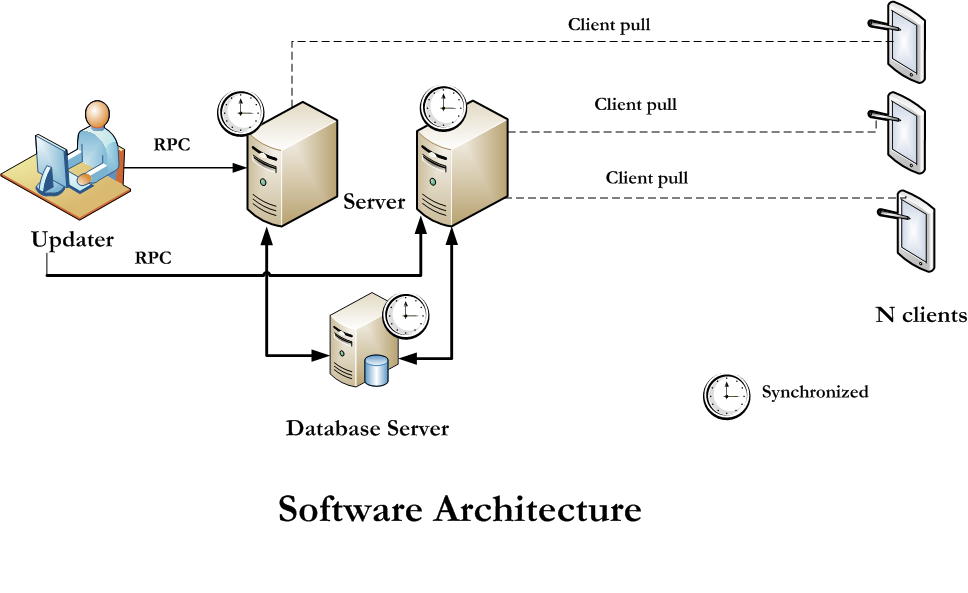
****

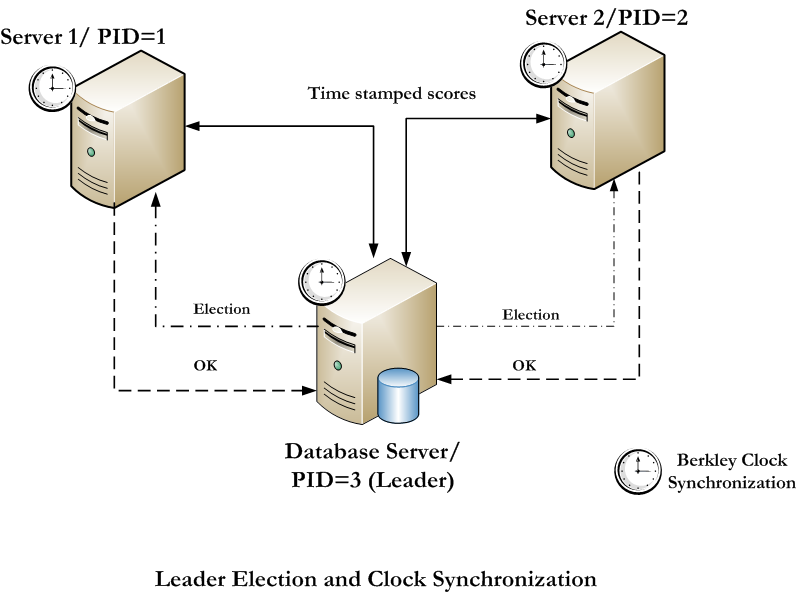
We used the Agile model of the SDLC to design the system. Agile Methods break the project into small incremental builds often called as Iterations . Each iteration typically lasts from about 4-7 days. Agile uses adaptive approach where there is no detailed planning and there is clarity on future tasks only in respect of what features need to be developed. The project is tested in every iteration which yields greater accuracy in the output.

In the Iteration 1 of our build, we analyzed the problem statement and planned the software and system structure and understood that new architecture to be developed by reusing certain architecture from previous lab (LAB 1). The design phase involved identifying what parts of previous build to be restored like RPC, client pull only. As part of the new design, the number of servers and database to be added to the design, the entities present and their access to the system were also framed. We also designed the Functional test cases to test the new implementation and Regression test cases to ensure that the previous build was still working without being affected due to the change in the system and software design. All these design were implemented and tested producing way for further development i.e Iteration 2. Iteration 2 involved building the part 1 of the project Leader election and clock synchronization The Iteration 2 design diagram features the detailed explanation of the various stages involved to complete it thereby leaving way to Iteration 3 which is part 2 of the project Implementation of Logical / vector clocks and we developed Lamports clock between the servers and the client. The detailed design diagram of Iteration 3 depicts the various stages involved in completing the project.

**Section 3. System Architecture**

**Section 4. Software Architecture**

****

****

**Section 5.1. Functional Test Cases**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **Test Scenarios** | **Priority** | **Input** | **Expected Output** |
| 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 | Score to be updated with timestamp.  Next question to be triggered |
|  |  |  | Any other input | Error message to be popped |
| 1.2 | Enter the Score | 2 | Valid Number | 1) An update in the Data Base server, DONE UPDATE  Should be displayed  2) Periodically time stamped score updated to the clients. |
|  |  |  | Any other value than a valid number format | Error message to be popped |
| 1.3 | Enter options to update 1 to update score or 2 to update medal tally | 1 | 1 to update score 2 to update medal tally | Score to be updated with timestamp  Next question to be triggered |
|  |  |  | Any other input | Error message to be popped |
| 2 | Presence of Server 2 | 1 |  | Server 2 is ready |
| 3 | Presence of Database server | 1 |  | Database server is ready |
| 4 | Check if all the servers are synchronous | 1 |  | All the servers should have same time |
| 5 | Check if the client requests are ordered | 1 | Start the different instances of client | All server should pick the same client as winner |
| 6 | Check if every 100th entry enters the raffle | 2 |  |  |
| 7 | Check if a winner is chosen | 2 |  |  |
| 8 | Check if leader is re-elected when previous one goes down | 1 | Start leader election again | All servers should be notified. |
| 9 | Check if servers are totally ordered. They are processing request in same manner | 1 | Keep request count synchronized | All servers should be process request in same manner. |
|  |  |  |  |  |
|  |  |  |  |  |

**Section 5.2 Regression Test Cases**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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,  DONE UPDATE  Should be displayed  2) 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 | 1 |  | Server 1 is ready |
| 3 | Presence of Client | 1 |  | Client Process is running status is displayed for every client created |

**Section6. Test Case Designs:**

1. **Clock Synchronization:** In this test case, we show that even on closing one of the running servers, leader election will reinitialize the leader by re-running the leader election algorithm.

* **Run the databaseServer.py**. It will run the leader election and choose itself as leader.
* **Now run server1.py** on different terminal. It will look for leader in system and finds that database is already there.
* **Now run server2.py**. Kill databaseServer.py by using **kill -9 <processId>.** Now system2 will run leader election and select itself as leader.

1. **Totally ordered Multicasting:** We will follow the steps below to see that the two server process maintains the similar process queue and displays the 100th request clients in the same order.

In order to show the above mentioned implementation, we run database and two servers. Run the multiple clients registered to one out of the two front-end server. The raffle entry winner displayed on two front-end server will be same and in the same order. This shows that the request Queue between two servers is synchronized.

- **Run the client.py** script twice with different client id as input.

E.g. ‘python client.py 3’ and ‘python client.py 4’

- Let 3 be registered with server1 and 4 be registered with server2. The raffle winners displayed on both the servers will be same and in the same order. This ordering is achieved due to total ordered multicasting.

1. **Load Balancing:** For load balancing, we monitored the load on two front-end servers by measuring the number of request served by each. The one with least requests is given the new request to serve. However, in case of tie, random number generation is used to break that up. By this methodology, server load remains evenly distributed. The load balancing can be seen by the number of process request on each server output messages. If the request keeps on increasing on both servers, it means that they both are handling it efficiently and processing new requests

**Section7. How To Run System:**

Same instructions can be found in Readme.txt with code:

* **Run nameserver**: python nameserver.py (**Note:** In case this command throws error, move to next step because nameserver is already running.)
* **Run database server**: python databaseserver.py
* **Run server 1**: python server1.py (dicts.py and lock.py should be in same folder as this script file)
* **Run server 2**: python server2.py

**Note**: These two are two front-end server which you start in any order.

* **Run clients**: python client.py <id> (Any number of client can be run with client ids assigned. Ids can vary from 3-10. Run it on different terminals and you can see the input. **Example: python client.py 3**).
* **Run Updateserver:** It will read input from input.txt fle : python updateserver.py

**Note:** You can start databaserver and front-end servers in any order. Leader election algorithm will choose leader based on the order of starting. If you kill any of these servers, leader election will find new servers.

**NOTE:** We are doing raffle entry for 20th request so as to see output early rather than waiting for every 100th request.

**Section8. Design Tradeoff considered and made**

* Most important decision was to make leader election take care of any order in which front end and database can run. We have made system handle any failure of server to select new leader.
* Berkley algorithm always checks for timeserver availability before running. If it is not up, we call up leader election to get master server.
* 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
* 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. Locks are put in database server to keep transactions follow ACID properties.
* We are doing raffle for every 20th request so as to see output early.
* We have stored output into db.p filename. This is the pickle file, hence a user can’t read it. This is the pickle file dump which we can reload to read the previously sent updates.

**Section 9: Future Improvements and Extensions**

* Instead of probing server for load and assigning request, we can use the round robin algorithm which keeps server much more balanced.
* Acknowledgments from clients should also be considered. Though it is not required for achieving total ordering but it is more reliable when system scales up.
* In future improvement, we would like to integrate database server to a MySQL database to store data instead of using pickle file.

**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 (Average Response Time):** 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: **0.05 seconds**. However, average is higher than the last time. In previous lab it was around **0.0017 seconds**.
* 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.