

Asterix and the Olympic Games

Programming Assignment 1

Prabhat Bhatt & Apoorva Saxena | Distributed & Operating System | 3/2/18

# What we did so far

Our design has four major entities:

* Sports Server or Obelix:

Obelix is responsible for all the communications taking place in the system. It uses socket package to implement socket programing.

It has the following functions:

* Update the database through REST API hosted by it
* Host RESTAPIs
* Receives Register, Set, Fetch requests from clients
* Pass Register, Set, Fetch requests to RESTAPIs properly formatted.
* Receives response from RESTAPIs and forward it to correct clients.
* Creating threads per client connection.
* Creating RESTAPI threads per client
* Authorize Cacofonix’s updates through REST API hosted by it
* Cacofonix:

Cacofonix is the only authorized update server in our system. All the request which includes updating database is sent from this server. It is responsible for updating the scores of individual events or overall medal tally. It needs to make sure that all the clients subscribed to a specific event are notified when there is an update on the event.

* Clients:

Our clients mainly have three possible functions:

* Fetch data like scores of individual events or overall medal tally
* Get Registered for instant score updates for Stone Curling event(Optional)
* Receive instant Score updates(optional)
* Database:

Since our system has a limited amount of data and it needs to be accessed regularly, we decided against using a database system and instead used a JSON text file which stores our scores, medal tally and registered clients. The dos\_data.json is our main file where all the above mentioned data is stored. Only the Sports Server or Obelix can access the file and all the updates provided by the update process or Cacofonix are authenticated by an Authorization ID.

We have used Python2.7 and the following frameworks/modules for development in our system:

* Flask RestAPI for Python:

Flask is a micro web framework written in Python and based on the Werkzeug toolkit and Jinja2 template engine.Flask Restful is built over it to provide functionality for restful communications via web services.

* Threading for Python:

Implementation of Threading ensured that different types of requests to fetch and update data are performed seamlessly while protecting data integrity.We have used python threading package for our system.

* Socket Programming for Python:

Socket programming provides the coder the freedom to build the system from scratch and control features like port number etc easily. It also facilitated implementation of Threading over our system which was otherwise complicated with higher frameworks like Flask or Django.

* Requests for Python:

Requests object emulates an HTTP object and performs the function of calling Rest web services in the system.

* Unittest and Mock for unit testing in Python:

The two APIs above have been used for the unit testing of our rest web services. Mock modules helps in mocking external services like file access etc in our testing code.

Our system has implanted all the RestAPI Web Services mentioned in the lab documentation with slight changes in terms of parameters:

* SERVER\_IP:PORT/getMedalTally/teamName
* SERVER\_IP:PORT/setScore/eventType/rome\_score/gaul\_score/auth\_id
* SERVER\_IP:PORT/getScore/eventType
* SERVER\_IP:PORT/incrementMedalTally/teamName/medalType/auth\_id
* SERVER\_IP:PORT/registerClient/clientID/eventType
* CLIENT\_IP:PORT/pushUpdate/variable1/variable2/variable3

All the communications happen in JSON format as specified in the lab documentation. For example:

{"Rome":

{

"Stone Skating": {

"Score": "20"

}

}

# How it works

Our implementation includes 3 major entities, which are Obelix( Sport Server), Cacofonix(Update Process),Clients.

**Obelix:**

Obelix has 5 major functionalities:

1. Creating socket and listening for connections.
2. Spawning one thread per client which starts communication between Client and Obelix
3. Starting a thread which starts REST API for each client once it makes a connection with Obelix.
4. Takes service request from Clients/Cacofonix and process.
5. Updating the JSON database file, updating the scores, medal tally and including registered clients.

**Clients:**

A client sends request over socket to Obelix which takes that request sends it to REST API started for that particular client. The request is then processed, and the result is returned to obelix which is sent back to client. A client may also send ‘Register Request’ which will be forwarded by Obelix and processed by REST API. When registering a client, we are adding clientID and its port to our JSON file.

**Cacofonix:**

Cacofonix in our implementation acts as a client, it sends update request to Obelix which passes its request to REST. In our implementation, we are setting score by generating 2 random values between 0 and 99 and updating it to our database. After update has been done it sends our pushupdate request which sends this data to all the client subscribed to a particular event(We are considering only Stone Curling in our implementation).Cacofonix also update the medal tally , in our implementation the request is passed through Obelix to RESTAPI which after authenticating increment a medalType(Gold, Silver or Bronze) by one.

**Database:**

Our JSON database has two major sectios:

* Teams:

"Team": {

"Rome": {

"Stone Skating": {

"Score": "20"

},

"Gold": 1542,

"Stone Throwing": {

"Score": "200"

},

"Stone Curling": {

"Score": "82"

},

"Silver": 8,

"Bronze": 7

},

"Gual": {

"Stone Skating": {

"Score": "10"

},

"Gold": 1548,

"Stone Throwing": {

"Score": "100"

},

"Stone Curling": {

"Score": "52"

},

"Silver": 2,

"Bronze": 8

}

The above format shows how all the scores of the two teams Rome and Gual for the three events Stone Throwing, Stone Curling and Stone Skating are stored.

* Clients:

"Clients": [

{

"port": "5050",

"clientID": "Client1"

},

{

"port": "5050",

"clientID": "Client2"

}

}

The above format shows how all the clients information is registered.

# What can we do in future

* Currently we have implemented instant score updates for only one event Stone Curling. We can further bring in other sports.
* Our events are hard-coded, and constant based on the current specifications. In future we can scale our work and add process which can add events at real-time, notify the score to subscribed clients during the games and at the end send the final score and update the medal tally according to the final result. Hopefully name it Asterix.
* More up-to-date frameworks for server push architecture can be implemented in future for example SSE, Pub-sub architecture such as MQTT.
* Currently we have used hard coded values for port numbers. In future we might extend our code to dynamically allocate port numbers to clients.

# How to run it

1. Our project requires following packages(pip install/pipenv install)

* threading
* flask
* requests
* random
* re
* flask
* json
* time
* socket

1. Run SportsServer.py.
2. Due to restrictions in the framework, we have hardcoded the port numbers. **Kindly run clients in following order Clienttab,Clienttab1,Clienttab2…Clienttab5**
3. Run Cacofonix.py after clients are running.
4. **Run them in different terminal to observer parallel activities of each Clients/Server.**

* We have ensured the requests issued by the Clients are as random as possible to make up for the hard coded port numbers.