

Asterix and the Olympic Games

Programming Assignment 2

Prabhat Bhatt & Apoorva Saxena | Distributed & Operating System | 4/3/18

# Introduction

Our design has nine major entities:

* FEP (Front End Processors)
* Load Dispatcher
* Raffle lottery Sport\_server
* Raffle Winner
* Server
* Clients(Client1,Client2,…,Client6)
* Cacofonix
* Database
* RestArt

The functionality of the above-mentioned entities is explained below:

* **FEP (Front End Processor**): In our design implementation we have included two FEPs which shares load from the load dispatcher. FEPs each has its own queue which is handled by them and the load dispatcher. FEP checks its queue regularly for any clients. If found it process request using Total Order Multicasting, i.e. multicasting data to other FEP with its local clock. FEP will process a request only when it receives an acknowledgement from the other FEP. A FEP sends an acknowledgement to other FEP if and only if it is no longer busy. To manage the concept of total order multicasting each FEP has its own socket waiting for other FEP to connect. Other than socket, each FEPs has its own logical clock which provides total ordering. The processing of request by FEP is done by sending the incoming request to backend server and returning the result back with the proper timestamp (using Berkley’s Clock).
* **Berkeley Clock synchronization algorithm** implementation in our code uses a ring algorithm for leader selection. This is followed by the master server requesting local time from slave servers and by using Cristian's algorithm calculating final time of slave servers. Finally using Berkeley algorithm, the average time is calculated and each slave server is notified the time difference.The Berkeley synced time is printed with each request received and processed by the servers. Our implementation involves communication between different services using Restful web requests as well as socket communication.

We have a function called Casting function which initiates voting protocol by starting FEP and FEP2 time servers. **BACKEND TIME SERVER ALWAYS RUN**. FEP2’s time server initiates the voting procedure. It casts its vote which includes the name and id of the server. ID is a randomly generated integer between the range 1 to 10. The vote is passed on to FEP2 and then to Backend Server which then performs leader selection based highest id value. The leader is notified by FEP2 and begins the synchronization process. The leader requests the local time from slave servers using restful web requests, calculates average time and responds with time difference using socket connection communication.

* **Total Order Multicasting:** We have implemented this method by starting FEP’s socket server and adopting few strict policies which are explained below.
  + FEP1 will not process a request until FEP2 has sent and acknowledgement.
  + FEP1 or FEP2 can send an acknowledgement if and only iff the FEP is now busy
  + Messages are exchanged between the servers along with local ticks(local clock) to maintain total ordering..
* **Logical Clock algorithm:** We have FEP1 with local clock starting from 1.0000 and incrementing in 1.0001,1.0002 fashion. Similarly, for FEP2 local clock starts at 2.0000. We define events as all request between clients and Sport server. When a message is sent it add its local clock along with it. This clock is taken out and checked with the local clock. The maximum of two with an increment as well will be the new local clock value for the receiving event and clock will move forward from that value as usual.
* **Load Dispatcher:** Load dispatcher like FEP is a part of our SportsServer. The main function of load dispatcher is to balance the load, in our case load is client not its request, and dispatch it to FEPs. Load balancer maintains its own queue and regularly monitoring it for load which is the client. Once it sees any load in its queue it forwards it to our FEPs. We have implemented load balancing by simply sending the load first to FEP1 and the next load to FEP2 and so on. We can always improve on load balancing technique such as giving load to the FEP(replicated server) which has least work or can be done by defining a threshold. The basic job in a simple load dispatcher is just to take the client and forward it to FEP and let FEP and Client do the talking.
* **Server:** The server is nothing but a basic multi-threaded socket which is listening to any client which wants to connect. Whenever a client connects it creates a thread. The thread just adds the client to the dispatcher queue. Again, like dispatcher its function is just to **forward the client not the requests.**
* **Clients :** We have implemented 6 clients interested in almost different things. The client code will spawn 6 threads for our 6 clients. The behavior is as follows a client comes checks score or checks medal tally of their favorite team and the closes its connection. More about what client is doing is explained in the code.
* **Cacofonix:** Cacofonix is our Scorer and Updater who is keeping a close eye on all the events happening at the Olympics. He is updating the scores and tally whenever there is any update. Cacofonix like clients is spawned inside client code it does its job and then closes the connection and then starts again in case of updates. More about Cacofonic is explained inside the code.
* **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 can access the file and all the updates provided by the update process or Cacofonix are authenticated by an Authorization ID.
* **Raffle Lottery :** Raffle lottery is constantly monitoring list called ‘raf\_list’ which is updated by FEPs whenever they process a request. It takes out 100th, 200th, 300th … , 1000th client and add it to ‘winners list’.
* **Winner :** Winner thread keep on monitoring ‘winner list’ until the it’s is filled up with 10 contestant. As soon as the list is filled with all 10 clients it randomly chooses and declares a winner.
* **RestArt :**  In our previous submission we used Flask API for our Restful web services by mistake, we ensured that we implement our Restful web services using RestArt python api for this lab. We have used restful communication for Berkeley clock sync algorithm as well.

What we did so far after lab1

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

* **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. We have implemented the concept of locks, sleep and broadcast to provide perfect thread synchronization.

* **Socket Programming for Python:**

Socket programming provides the coder the freedom to build the system from scratch. Its a best way of communication that can be used in our project. It’s easy , simple and with threading very effective.We have implemented multi-threaded socket programming.

* **Requests for Python:**

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

* **RESTART for REST API:**

**Since we used flask instead of implementing our own REST API in the previous assignment, this time we have implemented our entire REST API using RestART python module.**

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

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

{

"Clients": [

{

"port": "5050",

"clientID": "1"

},

{

"port": "5000",

"clientID": "Prabhat"

}

],

"Team": {

"Rome": {

"Stone Skating": {

"Score": "50"

},

"Gold": 16508,

"Stone Throwing": {

"Score": "300"

},

"Stone Curling": {

"Score": "300"

},

"Silver": 817,

"Bronze": 7

},

"Gual": {

"Stone Skating": {

"Score": "50"

},

"Gold": 17328,

"Stone Throwing": {

"Score": "600"

},

"Stone Curling": {

"Score": "600"

},

"Silver": 2,

"Bronze": 8

}

}

}

How it works

**Obelix(Sport\_server):**

Sport\_server has 5 major functionalities:

1. Creating socket and listening for connections.
2. Spawning one thread per client, this thread just add the client information (socket,addr)

Into the dispatcher’s queue.

1. Starting a thread which starts REST API(our implementation) , which is our end server in our case.
2. Starting FEPs.
3. Starting each FEP’s socket for Total Order Multicasting.
4. Starting Raffle thread, which chooses the contestant.
5. Starting winner thread which picks the winner

**Clients:**

A client sends request over socket to Obelix which takes that request sends it to load dispatcher. The dispatcher sends that request to FEP. The request is then processed, and the result is returned to FEP which is sent back to client.

**Cacofonix:**

Cacofonix in our implementation acts as a client, it sends update request to Obelix(Sport\_server) which passes its request to dispatcher and thereby to FEP. Front End Processores sends the request to BackEnd which checks the authentication\_ID ,which in this case is “Cacofonix” for simplicity, and update the database. The updates can be random in case of scores. The medal tally is always incremented by 1.

# What can we do in future

* We can scale our FEPs to more than 2, which will provide good load balancing when there will be many clients with many requests per clients.
* Our implementation of Berkeley Clock synchronization uses Restful requests for its communication. The requests have timed out in certain cases which is why we have used try-except around all code snippets relating to the Berkeley Clock synchronization implementation. In future we can implement a stronger model for this system.
* When doing total order multicasting we are sending data 3 messages per request processing and we receive 3 messages. This makes the total of 6 message per request handling, this increases the latency significantly. If we send data in package, we can reduce this bottleneck.
* Currently we have used hard coded values for port numbers. In future we might extend our code to dynamically allocate port numbers to clients.
* We want to provide proper authentication check for updating the database. Right now we are just using the name of the scorer i.e. Cacofonix.

# How to run it

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

* threading
* time
* requests
* random
* re
* json
* socket
* restart
* queue
* time
* datetime

1. Run SportsServer.py
2. Run cclienttab.py
3. **Run them in different terminal to observer parallel activities of each Clients/Server.**