

Asterix and the Olympic Games

Programming Assignment 2

Prabhat Bhatt & Apoorva Saxena | Distributed & Operating System | 4/4/26

# Introduction

Our design has nine major entities:

* FEP (Front End Processors)
* Load Dispatcher
* Sport Server
* Clients(Client1,Client2,…,Client6)
* Cacofonix
* Database
* RestArt
* Cache
* Heartbeat socket
* PAXOS(Extra Credit section)

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, takes client socket object out of the queue and starts communicating with client. For fault tolerance we have implemented and additional heart beat socket which when requested for a heartbeat by dispatcher sends a response. A FEP crash will cause the socket to crash, which will be detected by dispatcher. We also implemented global distributed cache in each FEP, which is a dictionary having **key as hashed URL** and **value as response from data base**. The value also contains a valid bit, which tells us that the cache entry has been modified and should no longer be used.
* **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 odd clients to FEP1 and the odd clients to FEP2. 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. For fault tolerance, the dispatcher starts its own heart beat server which continuously checks the heartbeat of our FEPs before dispatching the client to it. If a FEP heartbeat socket is non-responsive the dispatcher will dispatch the client to other FEP until the crashed FEP restarts.
* **Sport 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.
* **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 lab2

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

* Fault tolerance using heartbeat
* Cache implementation (push/pull architecture)
* PAXOS to handle byzantine fault

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.

**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. Configure how many clients you want to run at a same time. **We have 6 predefined client templets which are randomly run each time a new client is invoked.**

Limits on the client can be as many you want.

**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.

**Fault Tolerance**

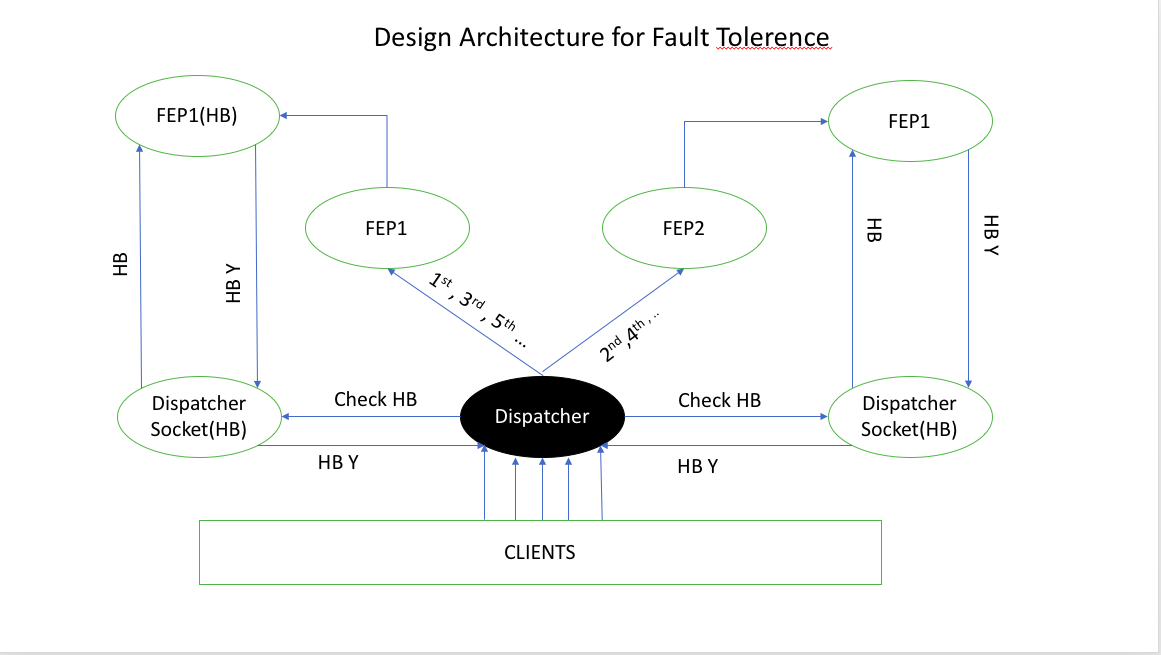
Implementation of fault tolerance has been done in following steps:

Step 1: Dispatcher, FEP1 and FEP2 thread starts. These threads start their heartbeat thread.

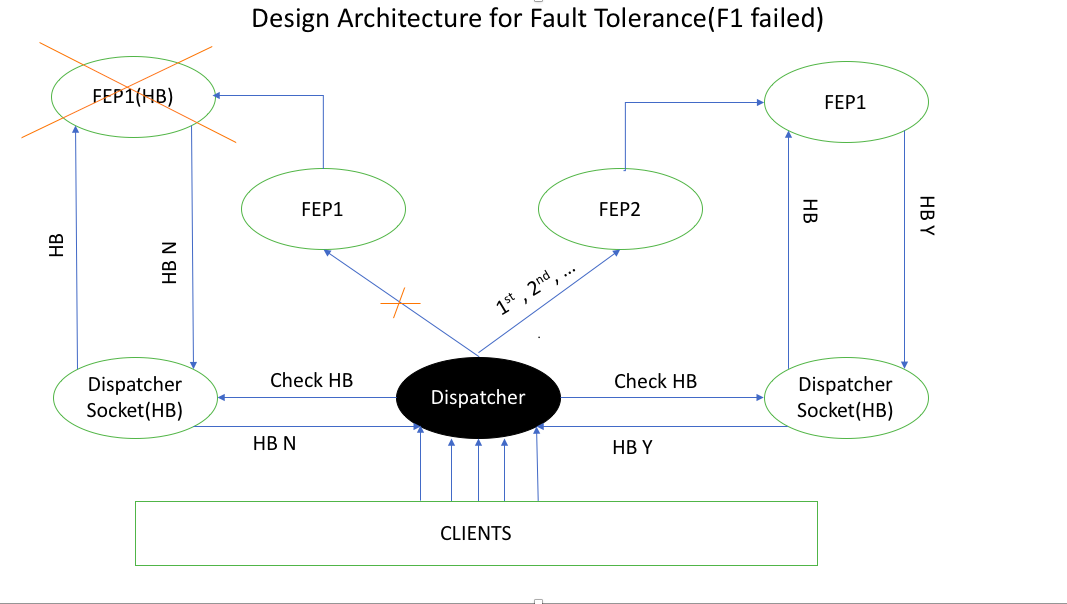
Step 2: Dispatcher before dispatching clients to the FEPs, check whether there is heartbeat on both the FEPs.

Step 3: If Dispatcher doesn’t receive heartbeat from the FEP1 HB(Heartbeat) sockets, it will not forward clients to that socket. Instead the client will be forwarded to the other FEP.

Step 4: Dispatcher repeats ‘Step 2’ again. If it finds the heartbeat from crashed server, it will assume that the FEP is restored and it will start dispatching the clients normally(even-odd distribution).



In Fig above , dispatcher is getting heartbeat from both FEPs so its sending 1st , 3rd , 5th … Clients to FEP 1 and 2nd , 4th , 6th … Clients to FEP2.



The figure above shows that the FEP1 has crashed and Dispatcher is dispatching clients to FEP2 only.

**EMULATING A CRASH SERVER:**

We have **designed FEP2 as a faulty server** which crashes after a fixed amount of time. The time is not clock time instead it’s a counter tick which counts from 0 to 100. As soon as the count gets to 100 , FEP2 HB socket sleeps for 60 seconds hence doesn’t respond back to dispatcher’s heartbeat checks. Once this cycle is complete normal functioning resumes.

# Cache Implementation (Shared Cache)

**Server Push Method(Mode=1)**

We have implemented cache with server push mode in following way:

1. We have used the URL sent by client and hashed it so that we can use it as unique key to our dictionary. The dictionary holds <Key, Value> pair in which value will be the result received from end server when we sent that URL and KEY is hashed requested URL itself. The value also contains valid bit which tells whether an entry has been modified or not.
2. We have excluded updates from Cacofonix since it is just an update hence we need not to store the response in our cache.
3. When we receive any update request the URL is split and our code figures out what cache entry will be impacted if we update a value in database.
4. Once it figures out, our model will invalidate the entry in the cache by setting the valid bit in the cache dictionary to 1.
5. Setting a valid bit means even if there is a cache hit (entry found in our cache for a URL) we will ignore that entry and fetch result from End Server. In the meantime, that entry from the cache will be removed.
6. Before sending request to End Server the cache entry is checked and if there is a hit and valid bit is zero the value is sent straight to the client.
7. If there is a cache miss, we will fetch data from end server. After receiving data, we remove the first entry from our cache and add this new <key, value> pair in our dictionary. (Note that entry will only be removed if the size of our dictionary is equal to the size limit).The first enty in the dictionary will be oldest entry. **We have maintained our cache size to be 5.**

**Server Poll Method (Mode =0):**

We have implemented server poll method in following ways:

1. Just like server push all URL will be hashed into the key and value from end server will be pushed in.
2. Cache hit is handled in the same way as in case of server push.
3. Cache miss on the other hand will be handled a little differently. In case of cache miss data is fetched from End Server and sent back to the client. The data entry in cache is not removed.
4. We constantly poll the FEP’s cache asking them to delete any entry in cache with valid bit set to 1.
5. A cache entry is invalidated in the same way as explained above.

**Polling frequency:** We have implemented cache polling by starting a tick whenever a cache is accessed; if the tick modulo 3 will give 0 it will start to poll and remove all element with dirty bit(valid bit set).This means that after every 3rd access to cache we will start polling and removing dirty entries.

# PAXOS (EXTRA CREDIT)

We have implemented PAXOS in following steps:

Step 1: Initially we set up everything, we have 4 FEPs (mirror FEPs) which will receive same client.

Step 2: The FEPs when initiated will set up a their respective PAXOS socket thread.

Step 3: We are using FEP1 as fixed proposer. Only FEP1 is responsible for sending response to the client.

Step 4: All FEPs will first fetch data from end server.

Step 5: After FEP1 has received the data PAXOS will be initiated.

Step 6: After an agreement is made, the agreed value is sent back to all FEPs.

Step 7: A failing server or a hacked server will never send the output back to client. Data will be sent through FEP1 only

# Algorithm For PAXOS:

**LEGENDS:**

n\_a: The local proposal number for all FEPs

v\_a: The value received from End server.

my\_n: Proposal number of our proposer FEP.

V: Final value that will be send from Proposer.

1. FEP1 starts PAXOS by sending *“FEP1:<prepare, FEP1.my\_n>”* to all FEPs using socket connection.
2. FEP2, FEP3 and FEP4 receives the prepare request. They split the request takes out my\_n and check whether their local proposal number(n\_a) is less than my\_n or not.
3. If the proposal number(FEP1.my\_n) is less than what FEPs already have they will send out reject message to FEP1 (e.g. *FEP2<prepare-reject>)* otherwise *“FEP2<prepare-ok, n\_a, v\_a>”.*
4. If Proposal receives majority of prepare ok message from all FEPs, which in our case will be more than 1(greater or equal to 2). It will take out n\_a from all the received message, take the value(v\_a) from the receiver which has largest n\_a value. Stores that value to V and update its own local n\_a value to the largest n\_a value received. Please note, in general a majority among 4 is taken as 3 but in our case if one server is byzantine server then taking its vote into account will result in system never reaching a majority hence majority is greater than equal to 2.
5. Proposer will now send *“FEP1:<accept, FEP1.my\_n, FEP1.V>”.*
6. All FEP receiver will repeat step 2 again. Sends accept only if my\_n is greater than the local n\_a they have.
7. If FEP1 will again see a majority accept from the FEPs, it will finally think that an agreement is made and the final message *“FEP1:<decide, V>”* will be sent from FEP1 to all server.
8. When FEP1 doesn’t receives majority in step4 or step7. The PAXOS is delayed and restarted from Step1.

# 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.
* A better polling mechanism will decrease the response time.
* We have implemented PAXOS with an assumption that FEP1 which is always a proposer will never crash. We can handle proposer crash by using our fault tolerance mechanism. If a proposer fails, we can start a new PAXOS using different server.
* 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(before running this change cache architecture mode in Sportserver file(line # 1520) by changing value of variable mode = 1/0 (Push/Pull).
2. Run cclienttab.py (5 seconds after SportServer; this is setup time for our system)
3. Configure how many clients you want to run at a same time
4. **Run them in different terminal to observer parallel activities of each Clients/Server.**

**EXTRA CREDIT**

1.) Run SportsServer.py

2.) Run cclienttab.py(5 seconds after SportServer; this is setup time for our system)

3.) **Run them in different terminal to observer parallel activities of each Clients/Server.**

**Please Note: When you will Run SportServer.py(PAXOS IMPLEMENTATION) , you will see a lot of lags in between logs at server side. These lags are inserted to see the response on Server side.**