1 How to Run

- Go to `src` directory and Execute *$ python server.py –n_servers 3*. One can separately test each functionality by setting the flags –is_leader_election and –is_clock_sync and –is_raffle. See the optional flags.

```
usage: server.py [-h] [--disp_ip DISP_IP] [--disp_port DISP_PORT]
                 [--fes_port FES_PORT] [--db_ip DB_IP] [--db_port DB_PORT]
                 [--n_servers N_SERVERS]
                 [--is_leader_election IS_LEADER_ELECTION]
                 [--is_clock_sync IS_CLOCK_SYNC] [--is_raffle IS_RAFFLE]

MuliThreadedFrontEndServer

optional arguments:
  -h, --help            show this help message and exit
  --disp_ip DISP_IP     Dispatcher IP
  --disp_port DISP_PORT
                        Dispatcher Port number
  --fes_port FES_PORT   Front-end servers starting port number
  --db_ip DB_IP         Database IP Addr
  --db_port DB_PORT     Database Port number
  --n_servers N_SERVERS
                        Number of Front End Servers
  --is_leader_election IS_LEADER_ELECTION
                        Leader Election Enabled?
  --is_clock_sync IS_CLOCK_SYNC
                        Clock Synchronization Enabled?
  --is_raffle IS_RAFFLE
                        Raffle Enabled?
```

- One also needs to run database server separately. Go to `src` directory and Execute*$ python database_server.py*.

2 Design

We are using Python 2.

2.1 Multi Threaded Server

This implements the multithreaing support for all APIs. This is the base class which all the servers inherit and hence database server, front-end server, dispatcher etc. all are multithreaded.

2.2 Database

We have exposed Database Server using REST APIs. It is implemented in 'multitier/database.py'. Following are the endpoints that it exposes:

- query_score_by_game
- query_medal_tally_by_team
- update_score_by_game
- increment_medal_tally

Database Schema Since clients need to know the time when the score was last updated we decided to keep a json file as our database table. The json file also keeps UNIX Epoch time (see /src/multitier/team.py) . In order to get maximum throughput we save all the game scores in separate files and also the medal tally are in separate file for both the teams. Hence, unless there is a query for same game or same team tally the databaseserver would process them parallely in separate thereads and they require separate locks.

2.3 Database Server

This is different from Database which implements/supports all the queries and the job of a server. This handles all the REST API requests and spawns a thread for each. [Design choice] The database server is only one and we have optimzed it to return read queries from in-memory rather than from the json file. Write requests updates the in-memorty data structure and the database file too.

2.4 Dispatcher

This is implemented in 'src/multitier/dispatcher.py' Dispatcher has all the load information which is the number of clients registered with each front-end server and it selects the front-end server which has minimum number of clients registered to it i.e. we do load balancing. [Design Choice] The Dispatcher starts the front-end server (see create_front_end_servers) and the database server. It assigns sequential port numbers to all the front-end servers. It exposes APIs required by the distributed front-end server like /getAllServers, /getAllFrontEndServers, /getLeaderElectionLock, /releaseLeaderElectionLock for various purposes like leader election and clock synchronization. It is also responsible for the raffle see start_raffle_thread.

2.5 Leader Election

Implemented in 'src/clocksync/leader_election.py'. All the servers (front-end, database) have a thread running (see perpetual_election), which implements the ring algorithm to elect leader. It happens every t seconds , specified in config file. A server would have to get the centralized lock (implemented in dispatcher having a rest endpoint) to start the election (endpoint /newElection) and it passes the election in ring topology. Every server in the ring topology implements the API end-point /passElection. When the system which started the election sees that the cycle is complete it releases the lock and sends the /coordinatorMessage to every system in the ring which the leader server address. After electing the leader it release the lock over the API.

2.6 Clock Synchronization

Implemented in 'src/clocksync/clock_sync.py'. Clock Synchronization happens every Delta/(2*rho) seconds perpetually in a thread(see def perform_clock_sync_func and def perform_clock_sync), the constants are specified in config file. The master/leader implements Berkley's Algorithm. Every clock exposes an API /getClock which returns the offset based on the UNIX Epoch time. The Master clock is the leader among the servers (front-end, database). As the leader is elected based on load, master is the most resourceful server.

2.7 Total Ordering

We implement the totally ordered multicasting. The test file confirms the order of elements that are popped from the queue are same for all the servers. Confirming that the order is same across servers.

2.8 Front End Server

The Front End Server implements all the APIs that are from the previous assignment. It also implements registerClient, unregisterClient which basically are for dynamically registering/unregistering clients including cacofonix.

2.9 Combined Server

This server is the culmination of all 5 servers: Multi Threaded Server, Front End Server, Leader Election, Clock Synchronization, Total Ordering. Uses Multiple Inheritance to achieve this.

2.10 Client

We have designed the Client to do both forceful client-pull whenever required and periodic client-pull perpetually in the background.

3 Test

It is explained in another file Test.tex and Test.pdf.

4 Improvements

- There is no fault tolerance. There are potential failure cases where if the dispatcher crashes the leader election will fail because we use centralized locking.
- Dispatcher starts all the front-end servers and database server on the same machine. Although the code is generic in the sense that we can start front-end servers on separate machines manually and they will connect with dispatcher.
- For raffle one should ideally return the request to client and do the raffle process post that on a separate thread. But we run the raffle on the same thread and so the response will be a little delayed compared to the case when the raffle is conducted post the request on a different thread.
- We have done load balancing at front-end servers level but now the real bottleneck becomes the database server.