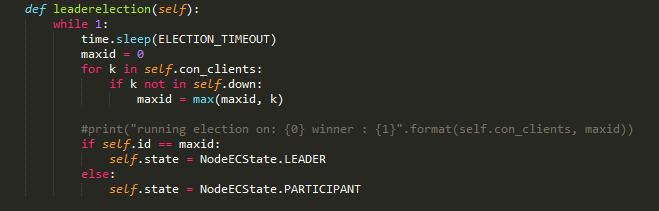
Design

The design can be sectioned into three main components

* ***Fault-tolerance:*** I implement a **primary-backup replication protocol** with remote write where the primary is responsible for all writes and reads can be responded from any replica. If a write is requested at a replica which is not the primary, the write request is first forwarded to the primary. The primary then propagates this write request to all the replicas, if all replicas successfully write and send a positive ACK then the primary wraps a positive response to the write and it travels back to the client. In this way the client is blocked until all replicas are updated.

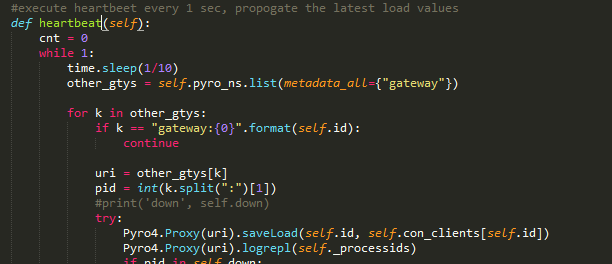
Choosing the primary

The primary is chosen by a simple leader election algorithm. We follow the design of RAFT and below show the implementation. As per RAFT we have an election timeout wherein all nodes vote for the leader, and the leader is chosen as the process which has the highest PID. If in a given time interval, the system shows to have two leaders which do not match (possible in crashes) then the client cannot use the system in that microsecond interval, although this gets auto resolved in the next election timeout



Heartbeat

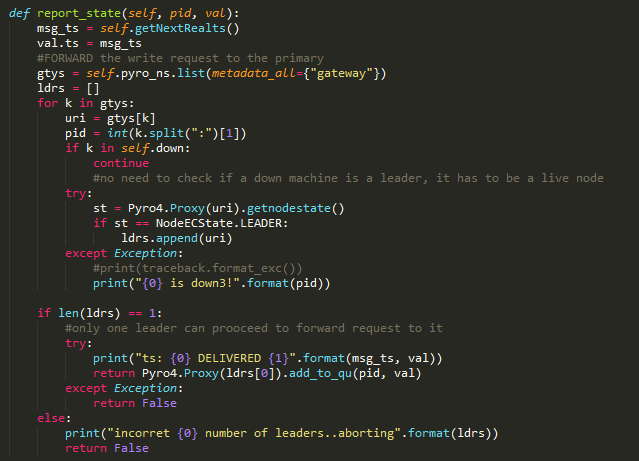
Central to the communication between the replicas is heartbeat.



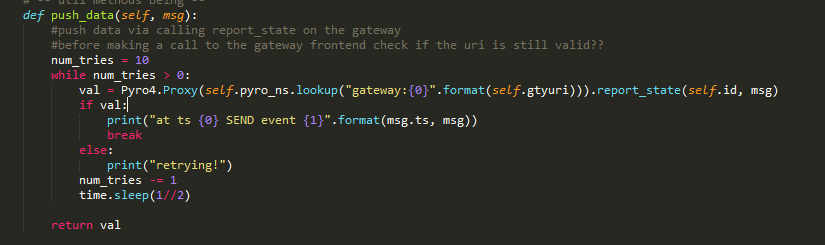
Heartbeat is basically an RPC call made from one gateway replica to every other gateway replica to send information such as LIVE status, LOAD information, and any new EVENT such as registration information of sensors/devices which needs to be replicated. These are contained in the function saveLoad (function for passing load such as number of clients/sensors currently connected to replica X) and logrepl (function for log replication)

Request Forwarding

As described above all writes are forwarded the primary replica. Since replicas may crash, join and exit the group dynamically we need a consensus algorithm, we choose RAFT. We implement RAFTs algorithm to forward all writes to the primary. As seen in the code below if a replica wants to forward a new data record, it first fetches all gateways (since this is dynamic) it then queries them to find who the LEADER is. If there are more than one LEADER (inconsistency) or no leader the replica replies to the client that it is aborting this transaction and the client should retry



* ***Consistency:*** A logical design decision with the above fault tolerance protocol was to implement **sequential consistency**. With this design choice, the system achieves same ordering of events across all replicas, making it correct to reason about events and in a situation where the write fails, the client is notified of the failure. In terms of the CAP theorem we choose **consistency** over **availability.** With this contract, the client handles this appropriately by using timeouts and retries as can be seen in the push data client method below. This method is part of every device/sensor.



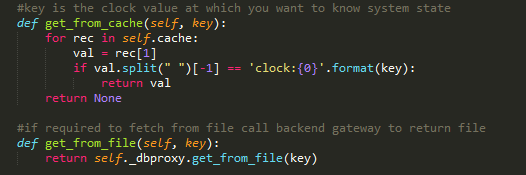
The primary replica maintains a local centralized queue where all the forward messages are deposited, this ensures that messages are consumed sequentially and another message is processed only when the first is fully completed, replicated and processed. Since sequential consistency doesn’t demand that the events follow a particular order, this paradigm goes well with our implementation. Further program order is followed since we depend on TCP which ensures that messages from the same sender come in the order of send

* ***Caching***: As a part of each replica frontend gateway we have an in-memory local to that replica cache of a fixed size (defined by the user at input time). All events that are received by the replica are cached at it (subject to the size) and an LRU cache eviction policy is used

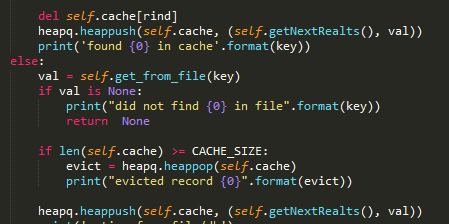
Design

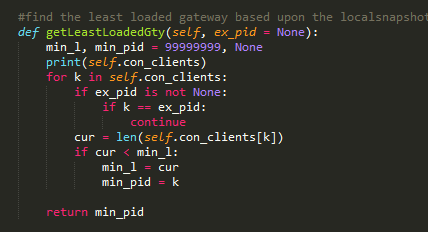
I have implemented an **LRU cache** (least recently used) in each of the backend gateway replica. This cache works to speed up reads by storing recently accessed data in-memory instead of going to the DB/file each time for a request. Further, I implement a **write-around** caching policy where the writes are directly written to permanent storage bypassing the cache, this reduces the cache being flooded by write IO requests and makes the cache consistent. To implement this I used a min-heap where each event (state information) is saved in the heap with the clock value of the time of access as the key. The cache size is a tunable user parameter which is saved in the variable CACHE\_SIZE.

The main caching methods are—



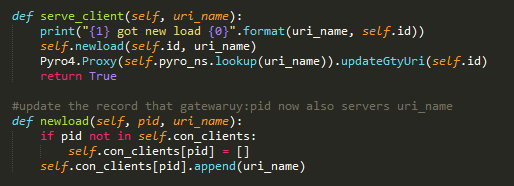
And for LRU cache implementation I use the heap data structure to push and pop elements where the key or priority measurement is done based on the access time



* ***Load balancing:*** This is implemented based on load statistic which is the number of clients currently being served by a given server or process. This load information is exchanged by piggy backing the heart beat messages. Shown below is how we get a least loaded machine
* ***Client handover incase of failure***

Here I explain the design of the actions the primary performs incase of a failure of any of the slave replicas

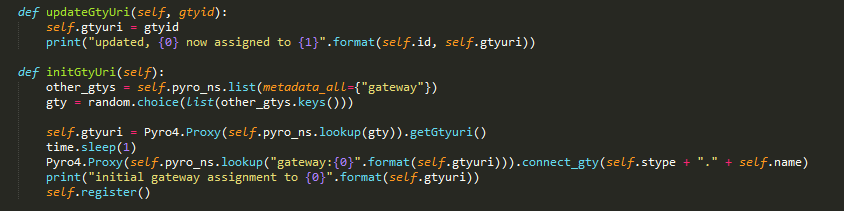
1. During each heartbeat message, which the replicas exchange with each other, they also exchange the load/sensors registry information. Thus at any point in time each replica has information about the registered clients at other replicas. This information is useful incase of a replica failure
2. Once a replica fails, since it cannot exchange heartbeats this is immediately caught by the replicas and they mark that node down In their local list. The primary replica in addition has the responsibility to redistribute the load that was present on the failed replica. This is shown in the test examples below
3. To assign the new clients to a specific replica the below code is run which does
   1. Notifying the replica of the new load
   2. Subsequently inform the client of the new replica assignment



In case the primary itself dies

1. Raft leader election which is running once every ELECTION\_TIMEOUT detects that there is no leader and starts an election
2. Once the primary is elected the steps are same as the previous

Each client (sensor/device) stores the gateway replica it is contained in a variable *self.gtyuri* which can be dynamically updated as the state of system changes. Below is a snippet of the sensor code which shows how this variable is updated and how it is initialized.

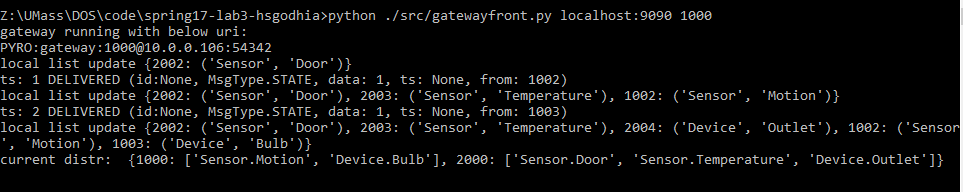


Test

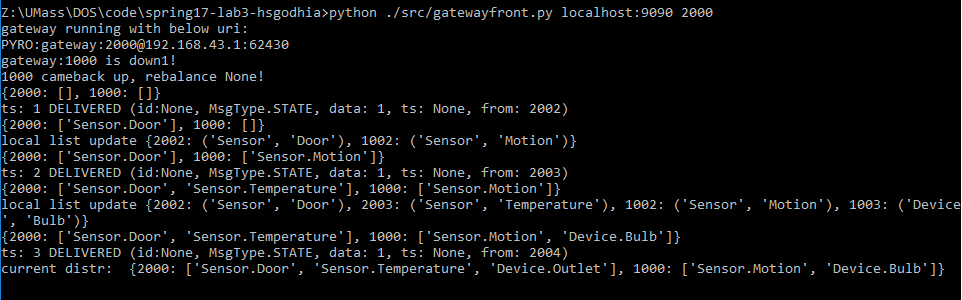
**Test : Replication and fault-tolerance 1 - Load balance at startup**

In this test we show during the startup of the devices and sensors, they are assigned gateways such that the load is almost equally distributed amongst the gateways. We assume two gateways with ID 1000 and 2000 respectively and we start 2 devices and 3 sensors

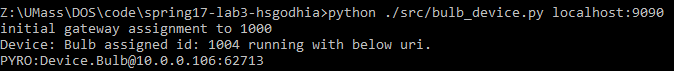
**Gateway replica 1**

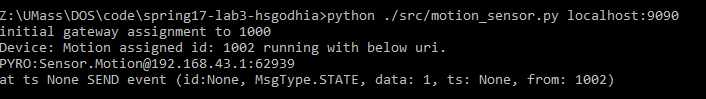


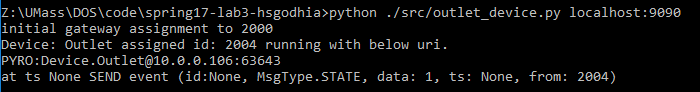
**Gateway replica 2**

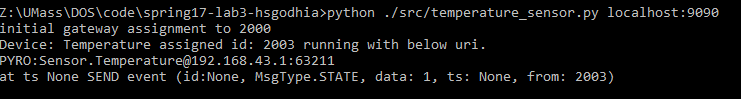


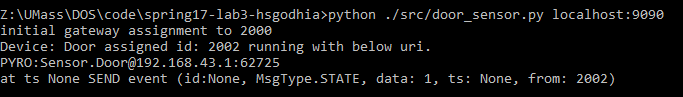
**Devices and sensors**











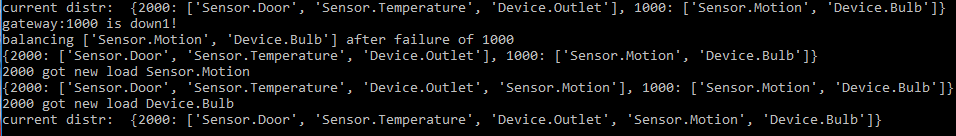
Observations—

1. As expected the load is equally distributed 2x for PID 1000 and 3x for PID 2000, this is shown in the gateway log as a print statement of the current load distribution
2. The gateways data store implement a **strict sequential consistency** and this is used when sharing registration information. Since each sensor/device registers with the gateway it connects to, some sensor/device register with gateway 1 while others register with gateway 2. As we can see in the gateway logs the list is updated so that all registration information is replicated and consistent

**Test : Replication and fault-tolerance 2 - Load balance on replica failure**

Here we simply stop one of the gateways by issuing CTRL + C (Note: we stop both the frontend and backend process of one of the gateway). In this test case we stopped the gateway PID 2000 and its load ['Sensor.Door', 'Sensor.Temperature', 'Device.Outlet'] is equally distributed between the remaining live replicas.

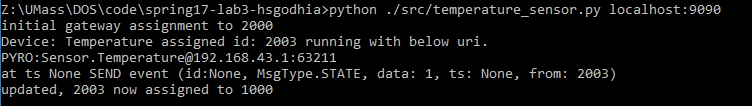
**Gateway replica 1**



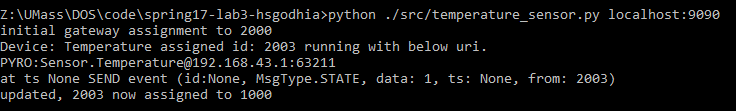
**Gateway replica 2**



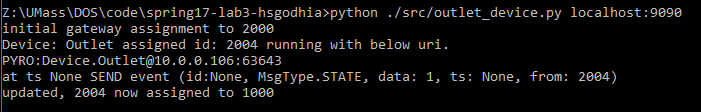
**Door sensor**



**Temperature sensor**



**Device Outlet**



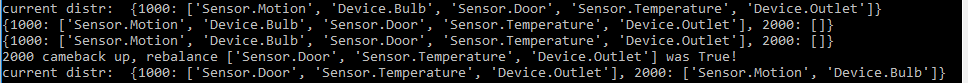
Observations—

1. As we can see the last statement in the log of the sensors who were earlier assigned to gateway PID 2000 are now reassigned to PID 1000 due gateway 2000 crashing
2. As can be seen from the gateway logs of replica 1, in the final statement once the sensor and devices are balanced, the current distribution is printed. Since, in this case we have two replicas, after the crash of one all the sensor/devices originally connected to the crashed gateway now fall over to this gateway. Hence, it shows all devices/sensors as its client

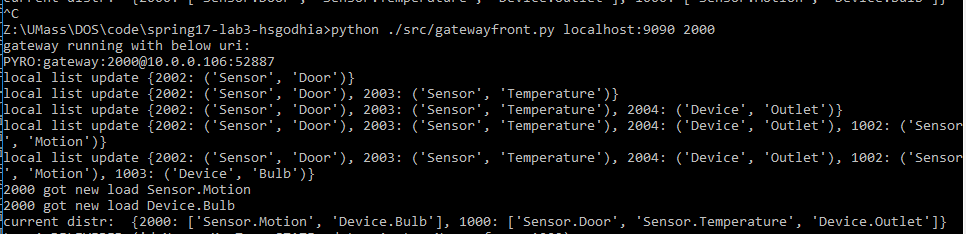
**Test : Replication and fault-tolerance 3 - recover from crash and rebalance**

Now we restart the gateway replica we killed that is restart gateway PID 2000. We notice that this change is quickly picked up by replica PID 1000 and the clients/loads is again re-distributed between the two replicas. Looking at the logs of the senor/device this was confirmed

**Gateway PID 1000**

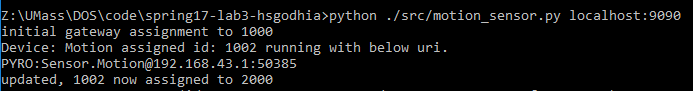


**Gateway PID 2000**

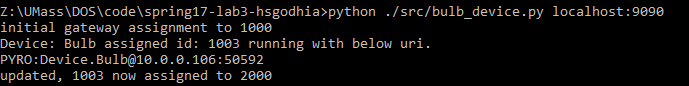


As we notice it gets new load, that is, it is assigned clients Motion and Bulb and serves them now

**Motion sensor**



**Bulb device**

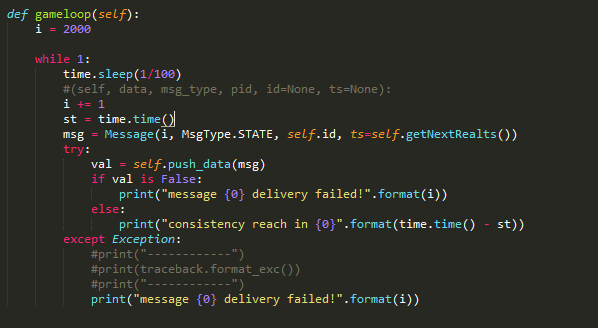


Observations—

1. First due to crash of gateway replica PID 2000 we see that the entire client load, sensors and devices are served by PID 1000 replica, this is reflected in the log of PID 1000 showing the distribution of load
2. Once, we restore the replica PID 2000, a re-distribution of the load happens and Bulb and Motion which were originally assigned to PID 1000 replica is moved and is now served by replica PID 2000, this is observed by their log above and in the gateway log of PID 1000 which shows an updated distribution
3. PID 2000 which was the crashed replica that restored shows receiving new load: motion and bulb

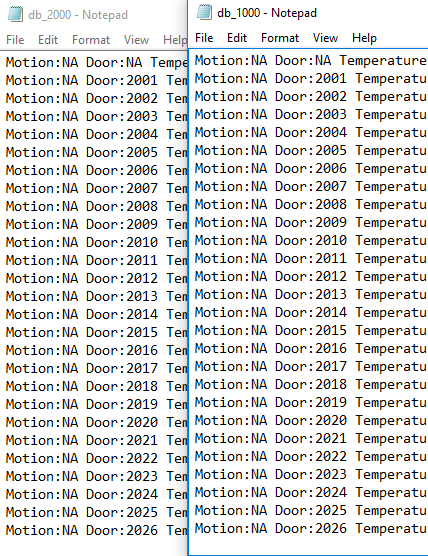
**Test : sequential consistency-1**

To test that the sequential consistency model implemented works correctly, I wrote a function in the door sensor that continuously sends events every 1/100th of a second, this is a hypothetical case for a door sensor but we can assume it to be like a gyroscope or speedo meter that is reporting very real time data. What we are verifying is that this event stream is consistently replicated across the gateway backend DB replicas.

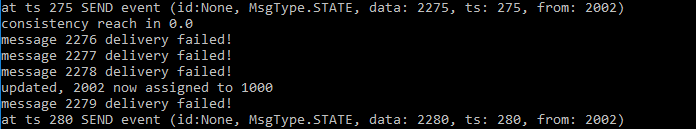


Observations—

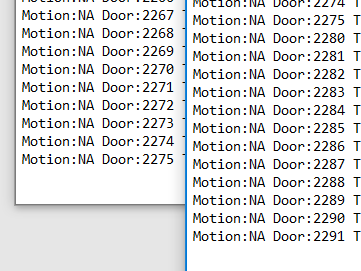
* On the next page, are the snapshot of the DB files for each replica. Db\_1000 for gateway replica PID 1000 and db\_2000 for PID 2000. Each door event is uniquely identified by the data payload (for simplicity it’s an incrementing number). We notice and confirm that the sequence of events received at both the replicas are **sequential consistent** (eventsin same order)



**Test : sequential consistency-2**

Keeping the setup the same as the previous test case, we now in between of the events being sent and written to disk we crash one of the replica by CNTRL + C. In this test case we want to measure how many events are missed due to this crash. Due to the crash of a replica to which the sensor was assigned it can no longer send events, so there is a loss of events. This loss of events continues until a new replica is assigned to the sensor. In my experiment I noticed a loss of 4 events, which is **approximately 0.04 seconds** until the sensor was automatically assigned another replica to which it can continue to stream information. Loss of message SEND from the door sensor

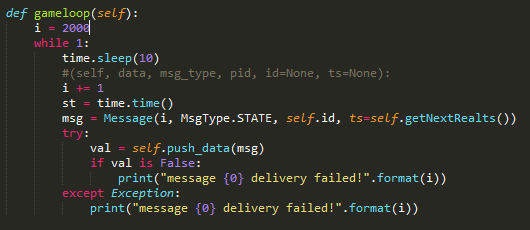
Similar gap in message stream at the gateway backend DB



Note: the left column is of the gateway PID 2000 to which the door sensor was assigned initially. Once PID 2000 is made to crash using CNTRL + C it stops receiving updates and the event with data 2275 is the last event that it sees. Now the sensor falls over to the other replica PID 1000 which is shown on the right in the above image. As we see there is a gap between events 2275 and 2280, the 4 missing events arises due to the time taken to detect a failure or crash of a replica and re-assigning the clients being served by that replica to another replica. Although this is data loss, the client is notified of such loss by returning a False response and the client does a retry. As we can see in the sensor/client log above the SEND events fail message is printed

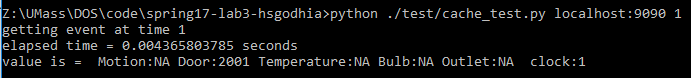
**Test : caching**

To simulate a highly loaded system with lot of events instead of making the sensors publish YES, NO or OPEN, CLOSE events I tweaked the message payload to contain numbers as the data (only for this test case). Such as the door sensor is made to publish messages every 10 seconds, and each message contains a unique number. This is also useful for us to check the cache working and calculating cache miss/hit speeds. This is contained in the game loop in one of the sensor

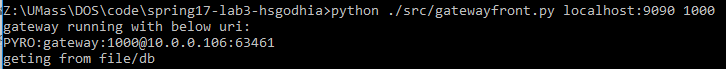


**Part-1 (get from DB/file)**

When the cache is empty, the first get request results in a cache miss and the system fetches the record from the database file. This is simulated in the test case as below



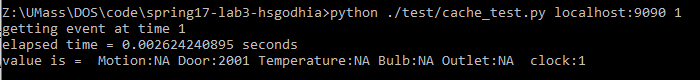
Where the request is fetching from the system at time state 1 (clock value). Here is how the gateway sees this request



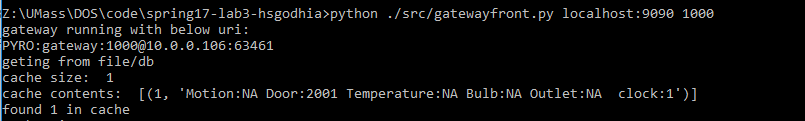
Observations—

1. As expected when the cache is empty the record is fetched from the database file which the backend gateway can access, the time taken is about 4 milliseconds

**Part-2 (get from cache)**

Now when the cache is loaded and we request the same record again, we would see a major speedup as this time the record would be fetched from the cache instead of the backend. 

Here is how the gateway sees the request with the cache serving the record this time

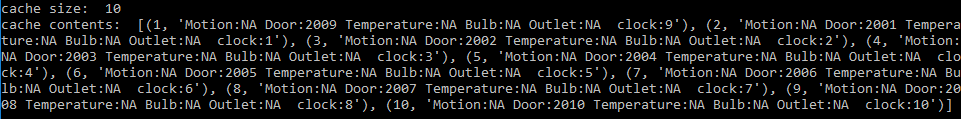


Observations—

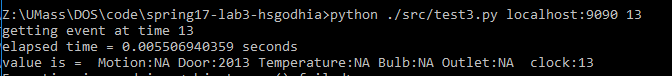
1. As we see above the speed up is almost 2x with the record being fetched in 2 milliseconds

**Part-3 (LRU cache eviction)**

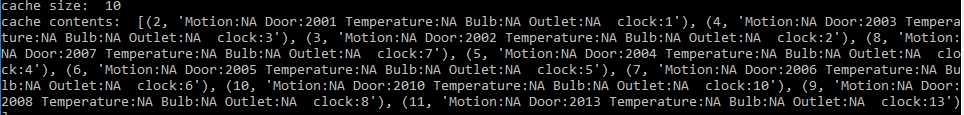
To show cache eviction we will load the cache fully till its CACHE\_SIZE currently set as 10. So after continuously fetching records assume the cache reaches the state



Now fetching more records will result in a cache eviction as capacity has been reached. In this case the least recently accessed record is the one with key value 1 in the above cache contents which corresponds to the data for Door 2009. It would be evicted which we simulated below. Note: this took considerably more time as well because of cache miss, cache evict, DB/file fetch and cache store it had to perform one addition operation as compared to a usual cache miss.



And now the cache contents as displayed by the gateway log below contains the new record corresponding to time clock value 13 but it evicted the record which was previously there



Observation—

1. Confirming the working of LRU cache eviction policy

**Consensus Design**

Assume that now the system has *k* replicas. The problem here is–

1. There could be a delay in state replication from one replica to another and an overall delay for all the replicas to reach a consistent state
2. Since each sensor/device registers with a different replica and the load of registration is as evenly as possible divided between the replicas different update/state change requests goes to different replicas
3. When a client sends a request the value should be the one that is agreed upon by the replicas or the replicas are in consensus of the response of that request

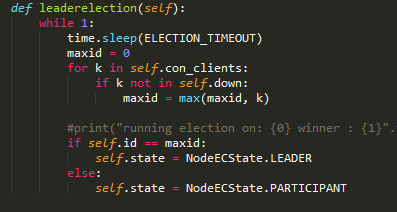
In such a system we need consensus on two things—

1. The order in which the change requests are executed
2. The value of each change requests and response to the client

I use the RAFT protocol to implement this. RAFT is a primary centered protocol, what this means is that all system change request go through the primary first, this is in-line with the primary-backup replication protocol design choice made earlier. This ensures that order of change events followed is consistent and that a majority of replicas reach are in consensus. In accordance with the RAFT protocol, the two main components that I implemented are *(\*****I am doing the extra credit implementation and testing****, note: I am not copying pasting the definitions of RAFT protocol such as term, leader, backup, election\_timeout, append\_entries and the algorithm for leader election or log replication, as they are present in the reference papers found here* <http://raft.github.io/> *. I describe below how I modify and apply these algorithms to our IoT system/problem )*—

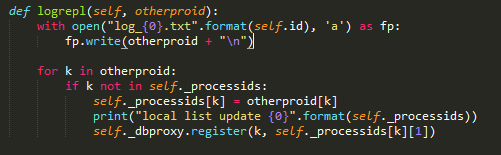
1. Leader election

I simplified the leader election to reduce the number of messages exchanged by assuming that all replicas are assigned a unique process PID and the algorithm chooses the maximum PID as the leader. Since the system is dynamic and replicas may crash or restart this leader election is run once every ELECTION\_TIMEOUT



1. Log replication

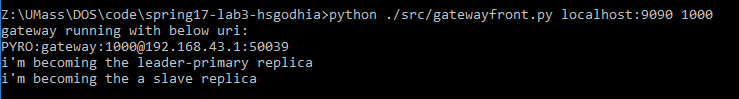
Another key component of the design is log replication, which ensures the consensus of a majority of the system components. Just as in the RAFT protocol my implementation exchanges APPEND\_ENTRIES during every heartbeat which is used to update the local logs of each replica. Once a majority of replica replies with a positive ACK the primary issues a COMMIT message which instructs all the replicas to make the log event permanent and store it into the stable storage. For ease of implementation the log is not maintained on the file system (as that would be the same as the DB/file) instead an in-memory map is used

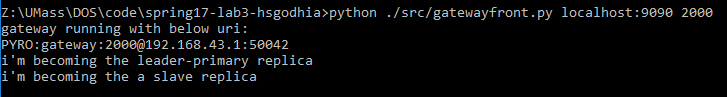


**Test RAFT - 1 auto-leader election**

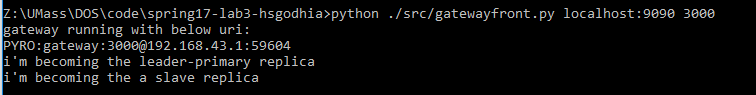
Here, I simulate and test the RAFT implementation I wrote for leader election. We start 4 replicas and

notice how the leader is updated each time a higher process PID node starts-up

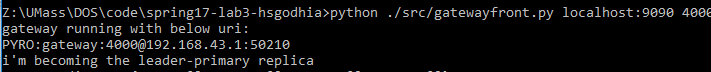
Initially we start the replica with PID 1000

Then we start the one with PID 2000

Then we start the replica with PID 3000



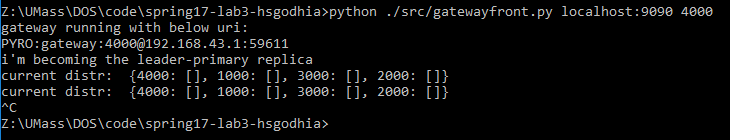
Finally we start the replica PID 4000



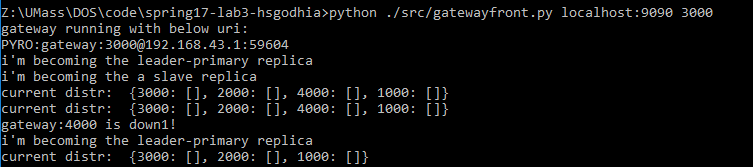
Test-2

Now I kill one of the replicas, lets say PID 4000 and we notice how the leader is automatically chosen

next



We notice that instantly the PID 3000 is made a primary replica



Note: the above snippets of the log show the client current distribution as empty since we are not

loaded/registered any sensors/devices but are independently testing RAFT

**Test RAFT - 2 log replication**

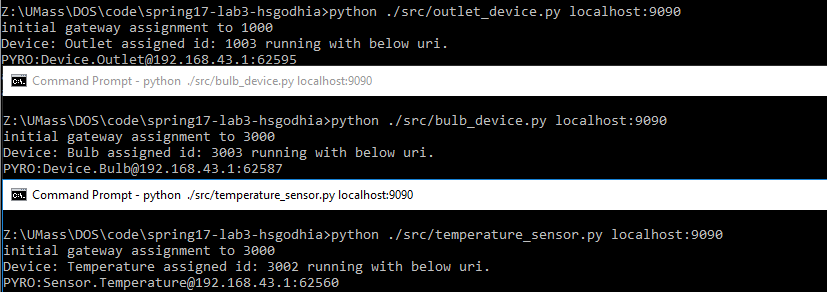
As per the code mentioned above in the design, we show how the log files are updated and replicated

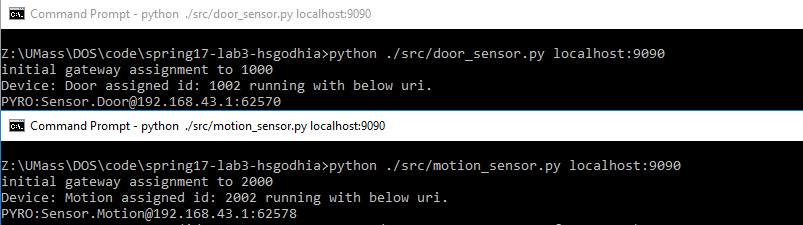
across all replicas. Here the setup is: we have three replicas running and several sensors/clients, we

show how the registration information is exchange and how log replication is used to keep replicas

updated and including screenshots of the append only write-ahead log

We started a total of 5 (3 sensors and 2 devices)





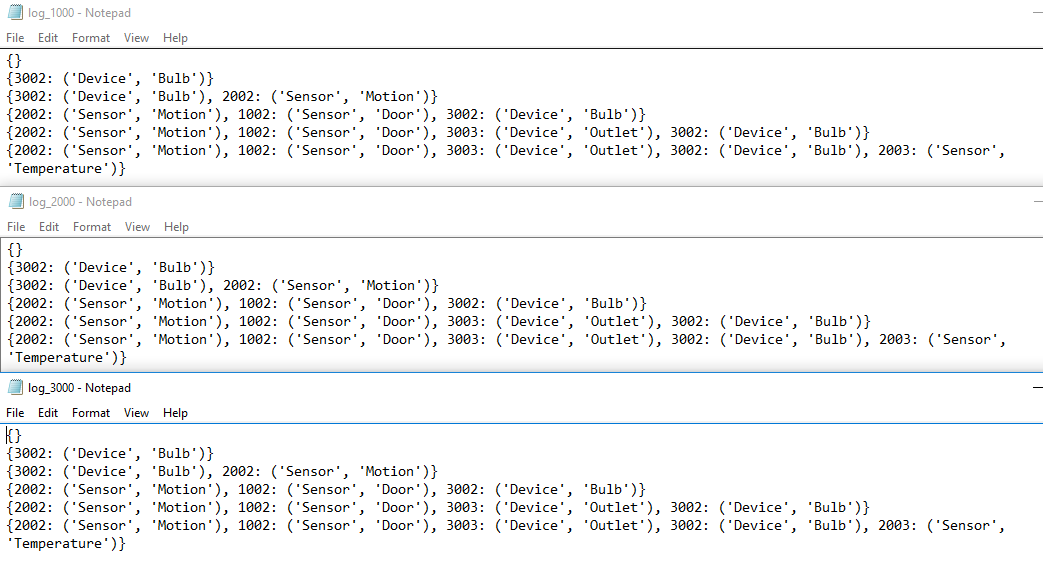
Since each sensor/device registers to different gateway replicas we need this registration information

to be exchanged and this happens according to RAFT log replication. Below is a snippet of the messages

exchanged via logs.

Observation—

* This way we are able to make the system work in *K* replicas and effectively using raft consensus protocol to orchestrate the process
* The raft is built into the gateway frontend, and work by spawning daemon background threads which perform the above functionality



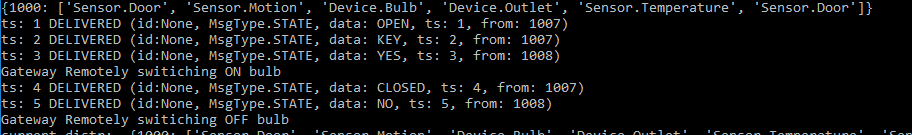
**User enter test case**

The previous test cases of Lab 2 apply here as well since the behavior of the sensors and devices are

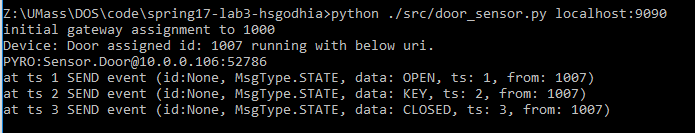
Intact. For brevity I only include the case of user enter exit and the other test cases can be read upon

from the Lab 2 report

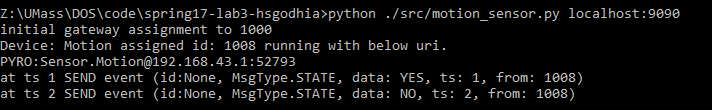
**Frontend gateway**



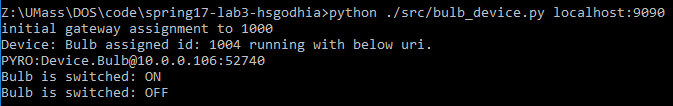
**Door sensor**



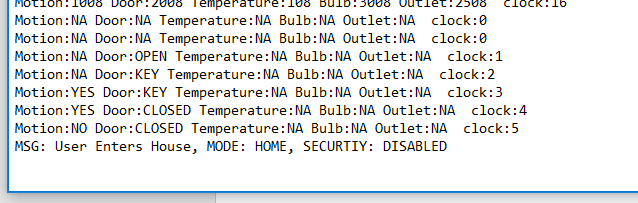
**Motion senor**



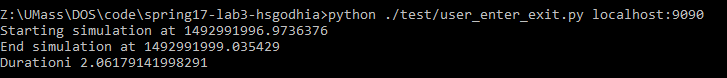
**Bulb device**



And backend DB file



**Test file**



Harshal Godhia

[hgodhia@umass.edu](mailto:hgodhia@umass.edu)