Programming Assignment 3: Programming Assignment 3: Internet of Things - Fault tolerance, Replication, and Consistency

DESIGN DOCUMENT

Submitted by: - **Rahul Raj and Olenka Dey**

1. Introduction

The main aim of this assignment is to understand the concept of replication, consistency, fault tolerance and caching. These concepts are then implemented on 'IoT for smart home', created in assignment-2. For this assignment, we have kept the the same structure of the distributed network with an addition, i.e. three sensor nodes: motion sensor, door sensor and temperature sensor, two device nodes: smart bulb and smart outlet and a multi-tiered, central gateway and its replica. Both the replicated Gateway servers different sensors and the devices balancing the load between them. Caching technique and strict consistency methods are implemented in the gateway to improve the performance and to ensure the data across the servers are persistent respectively. Whenever one gateway fails, the other gateway takes over the charge and addresses all the sensors and devices connected to the failed gateway on top of its own thus, implementing Fault Tolerance. We have used Remote Procedure Calls (RPCs) for communicating between processes and have used events, locks and threads for achieving consistency and fault tolerance. Finally, we implemented a PAXOS for consensus agreement among the gateways. The paxos is implemented using six replicated gateway and few sensors/devices such as temperatures sensor, motion sensor and smart bulb devices.

Following is the file structure which we have implemented.

--“**eventLog**” – This is a basic text file which contains all the events that a user or a gateway will perform during the run. It contains two tags: ”User” and “Gateway” for each event to redirect them to their respective processes for initiation. It also indicates the presence sensor, ”PS” with the door activities for security system.

-- “**myParser.py**” – This file is to read the eventLog file.

--“**run_ me.sh**” – Top most module. Used to start all the processes and also to kill one of the gateway after a specific interval of time.

Every node has one process. Suppose the node name is “motionSensor”, it has one file:
|---“**motionSensor.py**” – It initiates the application of the sensor that responds to user activities like enter or exit and to the gateway when it queries something. Similarly, “User” process have this file. Therefore, it contains 6 “nodeName.py” files. The replicated Gateways are the multi-tiered servers which contains two files: each frontend_1(2).py and backend_1(2).py and each gateway replica has its own database stored inside 'DB' directory. The database files are “current_status.txt” and “history.csv”.

2. How to Run

Running the application involves 2 steps:

1. eventLog File – Look into this file to have an idea of how events are mentioned and check the possible general and testing events. Modify them accordingly for any random sequences and save.

2. running run_me.sh in the terminal. This will start all the nodes and no need for the user to open in n different terminals, as all the messages will be printed in the single terminal for the user convenience. Use the following command to run it. It also kills one of the gateways after certain time, to demonstrate failure and fault tolerance. You can change the gateway to kill and its lifetime in this file. It takes cache size as arguments for both the gateways.

./run_me.sh cach_size_G1 cache_size_G2

Eg, “./run_me.sh 2 1”. This command should work, if it doesn’t work you have some permissions issue, use **chmod 775 run_me.sh** to give the desired permission to this shell script. Please look into the Output/Results document for the different results.

3. Design and Implementation

We choose Python as the programming language for this lab. We have used simpleXMLRPC for the all nodes in IoT implementation. We have written a shell script which starts all the nodes together and kill one of the gateway after certain time. Following are few of our design considerations.

- Random events can be given to the system using the eventLog File. It can be either the user or gateway initiated activities. We have added tags with each activity to differentiate between them.
- **Replication**: The two gateways are replicas of each other. The replicas associates and serves different sensors and devices based on load. They divide the loads almost equally among themselves.
- **Load Balancing**: Each sensor/device dynamically connects to the available gateways which exhibits low load
- **Consistency**: we have implemented **strict consistency** to synchronize the database. Whenever there is a write in any of the gateway's database, its front-end connects to other gateway's front-end and ask them to update their database so as to maintain consistency. So, there wont be any data loss in case one of the replica crashes in between as both the replicas are responsible for writing to the database. We though that strict consistency would work in this scenario as the network delay is very less in this case and the system has only 5 nodes to deal with.
- **Caching**: we have maintained a cache of user-specified size that each gateway's front-end maintain and update it whenever there is any query or update in states of the sensors/devices. The gateway first looks into its cache if the latest data is available, if so then it reads from it otherwise it query the sensor/device node over the network or the database based on the need.
- **Cache Consistency**: In order to have persistent data all the time, we have implemented “write-through” policy. So, whenever there is a write to the cache, the database is also written and there in no inconsistency between the cache and the database. One of the advantage of using write-through cache is that there will be “no data loss” if the replica crashes as we are writing to the database at the time of the cache write itself, in the case of write-back cache there was a chance of data loss or inconsistency because there is a possibility that the updated data might be in the cache but the data was not written to the disk and in the meanwhile the replica crashes so the data in the cache is also lost without updating the database and hence inconsistency or data loss will be there in this case.
- **Cache Replacement Policy**: The replacement policy we have implemented when cache is full is Least Recently Used (LRU). We are storing the access time stamp in the cache and by looking at the time stamp we can easily figure out which entry was least recent. Next, that entry can be thrown out from the cache and new entry can be cached.
- Replicas keep track of cache hit and cache miss and execution time to determine performance.
- **Fault Tolerance**: The gateways keep the other replicas informed about its connection whenever it establishes any new connection. So, basically the replicas know what address the other one is servicing. Both the gateway's front-end continuously checks for a heartbeat message from the other replicas about their health. Whenever one replica crashes (which is done using .run_me.sh), the other doesn't receive the heartbeat and thus, detects failure. It then it sends message to all the nodes, associated with the already collected addresses from the failed replica, that their gateway failed and it will be servicing them. All the nodes update the server address and connects to the running gateway. Once the connection establishes, the gateway resume its work and servicing all the nodes, taking up the responsibility of the other failed replica along with its own.
- Because of strict consistency and write-through policy, there will be no data loss in case of failures.
- We manually check the consistency between two database based on the event ordering.

As an output, the system also generates two files for each gateway replicas, which are always consistent:

- current_status.txt: it tells you about the latest state of each of the sensors/device.
  history.csv: It keeps track of all the state changes in that happened in the sensors and devices during the activities.

PAXOS Design and Implementation:

Paxos is used to maintain strong consistency among all the gateway replicas which basically ensures if any event is logged in one of the replica, that event data propagates to all other replicas and they too update it . Using paxos helps to maintain fault resilient database. Suppose, one of the gateway fails and other available replicas continues to log data, if the failed server comes back again then using paxos avoids conflicts in the log data. Thus, It provides safety during/after failures and in terms of data loss.

Considering 6 gateway replicas, we have implemented paxos consensus method for the system. Whenever the front-end receives a request from the sensors or devices like bulb on, motion active, door close etc., the gateway addressing the sensor starts the paxos. Once any consensus is achieved among the gateway replicas, the response to the request is sent to the particular sensor/device.

The paxos implemented for this IoT network basically have the following steps:

1. Leader Election: continuously run the leader election processes to keep all replicas updated about the leader. In the implementation the leader is the one with highest node_id, i.e., Gateway-N is the leader always, where N is the highest number.
2. Whenever any gateway receives a request from a sensor/device, it forwards that to the leader who is continuously watching for a signal, and starts the paxos.
3. When the leader receives a request, it sends “prepare(N)” to all other replicas (acceptor) connected to it. Where N is the index at which the event data need to be logged (the next empty slot). It can also be thought as the next event sequence number.
4. The participating replicas responds by accepting the request or denying it using “NAK”. The gateways accepts by sending back their last logged index, n', and if it hasn't logged anything before than it simply sends 'Empty'. This step is to synchronize all the database to same sequence number/index.
5. Once the leader receives majority of the acceptance from other replicas, it sets the value that need to be logged and give a reply back to the sensor/device.
6. After setting the value, it sends request to add consensus to all the acceptor replicas which contains the index/sequence number and the event information using “acceptRequest(N, data)”.
7. Therefore, all the gateway at the end of paxos, registers a sensor/device activity in its database with the same sequence number and all of them acknowledge the leader that they updated their database by sending “accepted”.

As an implementation, we showed how the temperature sensor, motion sensor and smart bulb device, all connected to different gateway's (we considered few remaining gateways who are not connected to any sensor/devices), query the gateway(read) and requests for change in their state (write). We could achieve consistency in all the write requests. We have compared the eventlogs of all the 6 gateways, and found that using paxos we can achieve consistency even in case when some gateway fails.

Trade-off and Limitations

- We considered one leader for the paxos implementation. We can account for multiple leaders by just adding leader control block that keeps track of them and give chance one by one.
- We considered write-through cache. We could have improved performance using write-back cache but there would have been data loss with that during any gateway failure.