Internet of Things - Fault tolerance, Replication, and Consistency

Noman Bashir, Shubham Mukherjee

April 23, 2017

The goal of this project was to implement replication, load-balancing, consistency, caching, and fault tolerance. The project builds upon the previous lab exercise to make it more practical by enhancing reliability of the overall system. We next describe the design decisions, trade-offs, working of the system, and possible future improvements.

1 Design Decisions

There were five core requirements for the project: replication, load-balancing, consistency, caching, and fault tolerance. The design decisions taken to achieve each objectives are described next.

1.1 Replication

The first task of the project was to implement replication where the two-tiers of the gateway server are replicated. In an ideal scenario, both of these servers will be running on a separate machine having different IP addresses. As during the implementation of the project, it is not convenient to continuously test our code on two remote machine, we decided to emulate the scenario one a single local machine. We start two name servers at different ports: 9090 and 9091 with "localhost" as server for both.

- **Gateway Server 1:** Server Address: localhost, Server Port: 9090
- **Gateway Server 1:** Server Address: localhost, Server Port: 9091

Having implemented this for the sake of convenience, we argue that Pyro facilitates the use of any other IPs as server addresses as long as they belong to the same network. Hence, this functionality can be tested on two different machines by simply changing the server address for both gateways.

1.2 Load-balancing

The next logical part of the assignment was to dynamically balance the load. We implement this in three steps:

- In first step, we assign the devices an incremental ID that starts from 1. For example if four devices are to be registered, they will be assigned IDs from 1 to 4.
- After ID assignment, the register function is called on the devices and assigned ID is passed to them as a parameter. Upon receiving the ID, the devices register themselves with a particular gateway based on their number. For example, devices with even IDs register themselves with gateway 1 and devices with odd IDs are assigned to gateway 2. In this way no matter how many devices are to be registered, they get divided evenly across the gateways.
- During the ID assignment these IDs are stored on an "id" file. The config file containing all the initial constants parses this file to store IDs for all devices, which are then used throughout the rest of the code.

We believe that this mechanism will work even if the number of devices is large. Also, in case of n replicated gateways, we can simply take modulo n instead of modulo 2 to divide the load evenly across gateways.

1.3 Consistency

The next part of the project was to implement the consistency mechanism for the replicated gateways. We implemented a mechanism that ensured strict consistency. The design decisions to enable it are following:

- whenever a message arrives at one gateway, it stores the message and sends it to the other gateway as well. Upon receipt of the message, the second gateway stores it into its database as well using the **recv_write_info()** function.
- although the two gateways are inconsistent only for the time taken for the message to travel from one gateway to the other. Another event may still occur at that gateway before it stores the message. In order to tackle that, we check the timestamp of the last message stored in the gateway. If it is lower than the message received, then we simply store the message in the database. However if it is higher, we swap the two messages.

1.4 Caching

We implement caching at the gateway level and the rationale behind them is explained next.

- **w**e implement the caching at gateways using the FIFO mechanism. We argue that this mechanism is better suited for our application than the least recently used (LRU) mechanism. It's because our event order logic to translate events into actions just considers the last event. This scheme is implemented using **cache_handler()** function.
- also, the cache size greater than 2 will always result in cache hits. It's because our event order logic just uses the last value from the database. The cache size of 2 will store the current event and the one before that resulting in 100% cache hits.

1.5 Fault tolerance

The last important part of the assignment was to implement a fault tolerance mechanism. The devices should be able to sense if a gateway goes down and register themselves with the other gateway. We took the following design decisions to implement the fault tolerance.

- Every gateway sends a periodic heartbeat to the devices connected to it as well as the other gateway. All of the processes receiving the heartbeat wait for "heartbeat period + extra time" to check if the heartbeat is received. Extra time is added to account for the communication delays. The gateway sends the heartbeat using **sendPulse()** function and all the processes check the pulse using **checkPulse()** function.
- We also decided to stop sending the heartbeat message once one gateway crashes. We argue that this is a reasonable design decision in the given scenario as there is no use of heartbeat messages once the backup gateway has failed. In case of more than 2 gateways, the checks can be removed easily to continue the heartbeat mechanism on the live servers.
- After the devices have sensed that gateway has gone down, they re-register themselves with the backup gateway with the same ID as before. This won't cause any problem as all the devices in the network have unique IDs.

2 How it works?

The process to run a test case for this project is explained in the README file. However, the overall flow of actions can be described as follows:

1 Two Pyro name servers start at localhost with ports 9090 and 9091. They can also be started at any given IP addresses, but that would have to be added to the config file as well.

2 registerProcess.py script is used to register the processes. It asks user to input the no. of devices (n) to be registered between 0 and 6. It selects the first n devices from the list containing all devices and registers them.

3 Registration process is dynamic. The devices are assigned incremental IDs and they evenly divide themselves between gateways based on their evenness.

The previous steps described how replication and dynamic load-balancing are implemented. We next describe the flow of actions for consistency and caching mechanism.

4 whenever an event occurs at a gateway, it sends the event information to the other gateway. Upon receiving the message, the receiving gateway stores the value into the database.

5 The value is also stored in the cache as well. When event order logic is to be implemented, the program fetches the values from the cache if possible.

We next describe the sequence of actions that take place to implement fault tolerance.

6 As soon as the gateway servers start, they begin sending heartbeat to the devices connected to it and the other gateway as well.

7 All of these processes implement a listener mechanism as well which checks if the last heartbeat has been received or not.

8 If a server crashes, all of the devices connected to it and backup gateway miss the heartbeat and conclude that the gateway has gone down.

9 The backup gateway stops sending the heartbeat and also informs the devices connected to it to not expect a heartbeat.

10 The devices connected to crashed gateway register themselves with new gateway and start communicating with it.

3 Design Trade-offs and Assumptions

There are some trade-offs and assumptions that we made during the design of the project. The details of these trade-offs is explained below:

- **Fault Tolerance:** In fault tolerance, we stop sending the heartbeat once one gateway has crashed. Our rationale behind this decision is that there is no use of such information when there is no other gateway available. We further argue that in a typical home scenario, we don't expect to have more than two gateways and this design decision suits the application scenario. It is beneficial to do that as it reduces communication overhead.

- **Consistency:** When a gateway receives an update from the other one, it checks if it is the latest entry. If an event has happened on the receiving gateway and stored in the database, we swap the two values so that the two databases are consistent. We make a decision to check the time stamp of only last event saved to avoid complexity. We believe that our decision is justified given the second gateway receives the message only after t seconds delay which is very low (less than 10ms as computed in last lab), since both gateways are on same network. It is highly improbable that even one event will occur during this time.

- **Fault Tolerance:** When one gateway crashes, it will take some time to notice that crash and get registered with the other gateway. There may be some events that occur for a particular device during this time interval. In actual scenario, such events will happen but won't be notified to the alive gateway.

  One way to tackle it can be that the device informs the alive gateway after registering if an event has happened and gateway takes an appropriate action depending on the nature of the event. For example, it may be okay to ignore a temperature update or bulb changing its state.

However, in the current implementation, we emulate the gateway server going down by shutting down the nameserver connected to it. Hence, our user process is not even able to access any device and control its state. Therefore, when user makes such a call, we print a message that event has occurred but the gateway cannot be accesses, and the home should move into a safety state. In actual, the event will happen on the device, could be sent upon registering, and appropriate action can be taken afterwards.

4 Possible Improvements and Extensions

Our project has a strong core which implemented all of the core requirements for the project. We have outlined some of the assumption that we made during this project as outlined in the previous sections. However, some of the further possible improvements in the core system as well as application are envisioned below:

- **Implementation of RAFT algorithm:** Although we described in detail the RAFT algorithm in the context of this system, we ran out of time to implement this into our system. We would like to implement this algorithm in future.
- **Data loss in case of name server crash:** In current implementation, the user is not able to do any task on corresponding devices after name server goes down. We didn't do it as it would have had consequences on our event order logic, which we argue that is better off with out this knowledge considering that these events may be ordinary one that don't matter. It won't be possible to do with Pyro, but we can atleast store these states inside the user process and send this information to the new gateway to store in database as well as change the state of the devices in their respective processes to be consistent.

5 RAFT Consensus Algorithm

The accurate working of our system depends upon the ordering of the events. As we retrieve values from the database for event order logic, it becomes crucial for us to store the values in correct order. In our algorithm, one event occurring at a single gateway should be propagated to all the gateways and stored on the respective databases. We plan to use RAFT consensus algorithm for this purpose. We first present the pseudo algorithm for the leader election process and then move onto describing how the events are committed to the database.

Algorithm 1 presents the leader election part of the RAFT consensus algorithm. We base this on the description provided on the website by the developers of this algorithm. All of the nodes start as followers and we assume that election timeout occurs for one or more of the gateways. Upon timeout the followers turn into candidates and start election process by asking for votes by sending *vote_for_me*. If another gateway receives vote message, it replies with a *yes* vote if it neither have not replied, nor started the election with a higher term number. Otherwise, it replies with a *no* vote. The candidate gateway counts the vote and becomes leaders if successful, otherwise process is restarted. Upon winning the election, it calls the heartbeat function to start sending the heartbeats. The heartbeat function (Algorithm 2) is described below:

- In the context of this system, we assume each gateway as a user who wants to commit a value to all databases when an event occurs at it.
- Whenever it has something to commit, it sends that value to the leader which in turn sends it to all the gateways and asks them to store the value.
- The receiving gateways stores the value and reply with a *yes_stored* message.
- If majority of the gateways respond positively, the leader proceeds with commit changes locally and also sends a *commit* message to all the gateways.

This scheme works fine even in the presence of faults and missing gateways. However this algorithm suffers one drawback. If the leader receives a newer value before an older value which is received at the leader with some delay, the newer value will be committed before the older value. This serves the consensus purpose as all of the databases are in the same state, but it may not work in the context of this system which works based on the exact event ordering.

In order to solve this potential issue we use both **RAFT + Paxos**. We assume that whenever an event occurs at any gateway, it informs all the other gateways about it rather than only the leader. The algorithm is described below:

- All of the gateways maintain a temporary queue in which they save the event information before commit.
- Leader sends all of the gateways a message purposing a certain commit event.
- If a gateway also has that event on top of the queue, it sends *okay* message. However if it has some other value, it suggests that value to the gateway and replies *no*.
- Leader counts *okay* votes. If it gets the majority votes, it commits the changes locally and asks all of the other gateways to commit that change.
- If it doesn't get strict majority, it checks the response of the gateways to get the most suggested value. It proceeds the commit with this value, if it is suggested by more than half members, and asks all other gateways to commit it.

This algorithm assumes that the event queue at most of the gateways will be in correct order. We argue that it is a realistic assumption as conflict only occurs when an event happens on a gateway before a message about previous event on a any other gateway is received. As the delay is only less than 10ms, we argue that this issue would rarely happen.

Algorithm 1 RAFT Consensus Algorithm

```
Require: $list\_of\_ids, total\_number\ (k)$
 1: $start\ as\ follower$
 2: $start\ listening$
 3: if ($election\ timeout$) then
 4:     $become\ candidate$
 5:     $start\ election\ process$
 6:     $vote\_count\ ++$
 7: else
 8:     if ($recv\_msg == vote\_for\_me$) then
 9:         if $already\_replied$ then
10:             $vote\ no$
11:         else if ($haven't\ replied$) then
12:             if ($started\ election\ with\ higher\ term$) then
13:                 $vote\ no$
14:             else if ($started\ election\ with\ lower\ term$) then
15:                 $cancel\ election\ \&\ vote\ yes$
16:             else if ($started\ election\ with\ equal\ term$) then
17:                 $vote\ yes$
18:             else
19:                 $vote\ yes$
20:     else if ($recv\_msg == vote\_reply$) then
21:         $count\ yes\ votes,\ count++$
22:         $at\ the\ end\ of\ counting$
23:         if ($count > k/2$) then
24:             $election\ won$
25:             $call\ heartbeat()$
26:         else
27:             $wait\ \&\ restart\ election$
28:     else if ($recv\_msg == heartbeat$) then
29:         $reset\ election\ timeout$
30:         if ($recv\_msg == set\ x$) then
31:             $store\ x$
32:             $reply\ yes\_stored$
33:         else if ($recv\_msg == commit\ x$) then
34:             $commit\ x$
35:             $reply\ yes$
36:         else
37:             $reply\ anything$
```

Algorithm 2 HEARTBEAT()

```
function HEARTBEAT( )
    $while(true)$
    if ($input\ from\ user:\ commits\ x$) then
        $send\ x\ as\ heartbeat$
    else if ($reply\ received\ from\ followers$) then
        if ($recv\_msg == yes\_stored$) then
            $count\ yes\ votes,\ count++$
            $at\ the\ end\ of\ counting$
            if ($count > k/2$) then
                $commit\ changes\ locally$
                $send\ commit\ msg$
            else
                $send\ discard\ msg$
            $respond\ user$
        $sleep(heartbeat\ time)$
    else
        $send\ empty\ heartbeat$
```