Operating Systems : Lab 2

Sean Noran

April 3, 2017

1 Instructions to Run

See the README.md file on instructions to run the tests.

2 Design Choices

The system is written in Python. I use the asynchat module to allow the server node to communicate asynchronously with multiple clients to ensure that one client does not block communication with another.

One important design choice is that I did not use a second communication layer which would allow clients to communicate with one another for the purpose of registry, leader election algorithm and clock synchronization. The reason is because I initially did not know that we were permitted to do that, but also that in a real IoT system, clients likely would not have the possibility of communicating with one another - only through the gateway. This doesn't mean that my system does not allow clients to communicate to each other; they can via the gateway, just not directly through a socket. This is important to mention because it means that the round-trip time between two clients used for the Berkeley synchronization algorithm is somewhat larger than otherwise. In general, this also means that the synchronization error is also somewhat larger, however still reasonable.

The base system comprises of a server and client, which can be found in the server.py and client.py files respectively. Each other node then subclasses the appropriate role, e.g. the Gateway class in gateway.py subclasses Server and the Device class in device.py subclasses Client. The Backend class also subclasses Client, since it need only communicate with the front-end server tier, making it in effect a client node.

As another design choice, I do NOT use concrete subclasses of Device such as TemperatureSensor or SmartLightBulb to represent those entities. Instead, when creating a temperature sensor, this can be done by specifying the attributes of the Device instance appropriately, e.g. type='sensor' and name='temperature'. Additionally, the user process should specify the values of these sensors using the Device._set_state() method. This could for instance set the temperature reading to 70 degrees. Of course, in a real IoT system, the sensor would regularly make a measurement with some analog device, convert it to a digital reading and update its state appropriately. We have to have some other process to intervene since we are working with virtual devices.

Also, you will not see the query_state functionality available in the gateway used, because for the purpose of this home management system, it is not necessary. In reality, this may be very valuable especially if the connection is not always reliable or another entity had the ability to query certain sensors, e.g. a user interfacing through their desktop computer.

3 Tests

The documentation at the top of each test file is fairly detailed. See that. In short, test_sync.py tests the leader election and Berkeley clock synchronization algorithms. Then test_vector_clock.py tests that the vector clock multicasting correctly gets the event ordering; for the purpose of this test, no Berkeley synchronization is done, because all nodes are actually running on the same machine and therefore are perfectly synchronized. Lastly, test_home_system.py tests to make sure that the security system works appropriately after simulating the user leaving and re-entering with the beacon, then the user leaving and a burglar breaking in.

In the second test, the alarm should be on (1), then go off (0), then go back on (1), then the alarm should sound (-1).

4 Future Improvements

This system is far from robust. One of the details I wasn't able to address was using a dynamic vector clock, which can update as clients are added or removed. Currently, I assume there will be no more than 7 devices with vector clocks in the IoT network (including the gateway, excluding the back-end tier). If you try connecting more devices, it will raise an exception when reporting one of the device's state.

In general, I don't handle the disconnecting of nodes gracefully. I make no use of an unregistering mechanism, meaning because of the issue just described about the limit on the number of clients, you can only run each test script once, then the server needs to be restarted. The good thing is that we can make assumptions about the IoT network we have, and if none of the devices ever fail, then this isn't really a problem. In reality, devices will disconnect and this should be able to be handled on the server (whether the disconnection is graceful or not!).

It would also be useful to cache latest entries in the database for faster lookup. Preferably, the database could also be queried with a reduced latency for more recent events, since they are more important for the purpose of this system.

5 Latency and Performance

The latency at each of the clients appears to be relatively consistent even when more clients are connected, around 25 ms. However, this was not tested with a very large number of nodes. When the number of clients is at 7, then the gateway's latency is about 50% larger.

I did not have much time to do comprehensive performance tests of the system. If I had more time to do this, then I would plot the latency as a function of the number of clients connected to the gateway and connect many clients and see whether the performance degrades significantly.