Overall Program Design:

Our implementation uses Python and the python xmlrpclib, SimpleXMLRPCServer and socket library. The system consists of a two-tier gateway (a front end and a database), four sensors (temperature, motion, door, beacon), two smart devices (bulb, outlet), and a user process. The communication between gateway and distributed clients are mainly done by remote procedure calls (RPC). There are five important RPCs in the system: *register(type,name,localadd), report_state(devid, state)* and *change_mode(mode)* are provided by the gateway*, query_state(devid), change_state(devid, state)* are provided by distributed sensors and smart devices, *text_massage(string)* is provided by the user process. Besides, every process has a *update_vector_clock(vector)* for vector clock update. For leader election and clock synchronization algorithm, we use sockets to exchange messages.

Design Details (Lab 2):

Leader election and clock synchronization:

We use a token-ring algorithm to do the leader election. First, we use the gateway as the coordinator to build the ring topology. At the beginning, the gateway listens on a broadcast port for other devices to report their address. Processes who want to participate the election broadcast their address to this port. The gateway then collects these addresses and build a ring structure with them. It sends every candidate its successor in the ring. After that, the gateway generates a token and circulate it. The election is based on a random generated election id. The one with the largest id would be the leader. After one round, the gateway gets the result and notifies everyone the result.

The elected leader starts a Berkeley clock synchronization algorithm and acts as master. Communications between master and slaves is done through TCP. The leader will tell slaves its current time and the slaves return the offsets. The leader then return the average offset and everyone calculate its synchronized offset according to it. After synchronization, clients start to register to gateway as in Lab 1.

Vector clocks:

Each party participating has a unique index in the vector clocks. We use the device id assigned during registration as the index and set the id of gateway to be 0. Every time a message is sent, sender initiates a multicast to all other parties of the current time vector. On receiving a message, the receiver updates bits of all other parties and add one to its own value. In this way, every message has got a time vector, and the system can provide causal ordering.

Event ordering:

we timestamp every event with a vector clock and synchronized physical time and save them in the database. When certain events are triggered (for example, door sensor pull a state), we look in the database for relative events and use timestamp to order time. If we

use the physical clock to reason the ordering, we only consider events in the database that are within a certain range of the timestamp. In this way, we can avoid searching the entire database.

Back-end database:

The database process works as a back-end tier, and it interacts with the front-end tier with through three interfaces: write (cid, state, timestamp,vector) writes into the database, recording the state, timestamp and clock vector. read(cid, timestamp) reads from the database. When timestamp == 0, returns current state; when timestamp >0, return state of time indicated by timestamp; when timestamp < 0, returns all the state history of the device. read_offset(cid, timestamp, offset) returns the state of a device from (timestamp – offset, timestamp). The gateway and the database interact through RPC calls.

How to run the program:

Deployment:

1. The input file name is "test-*input.csv*"
2. Run the bash script file "*code_single/run-all.sh*". Output will be in separate files. Or run "*code_single/run-test-case.sh*", output to “test-output.txt”.

Note: The clients and gateway uses port number 10000 to 10008 to communicate. To adjust these ports, modify *localadd* values in "*setting.py*" file.

Possible extension:

Now, we assume the topology of the network is known ahead. We could extend this to an unknown network topology where the number of devices are not known ahead. This situation is more common in real world. Also our front-end and back end use synchronized RPC communication, we can extend this to asynchronize to improve concurrency.

Group member:

Tengyu Sun
Dan Zhang