IOT Assignment #2 COP5614

Haibin Guan, Pedro Soto

February 2019

1 Overall Program Design

The overall design of this network is the same as assignment 1 with some added features: the back-end/database, a door detector, a leader election algorithm, and two clock synchronization algorithms (Berkeley and Lamport). Most of

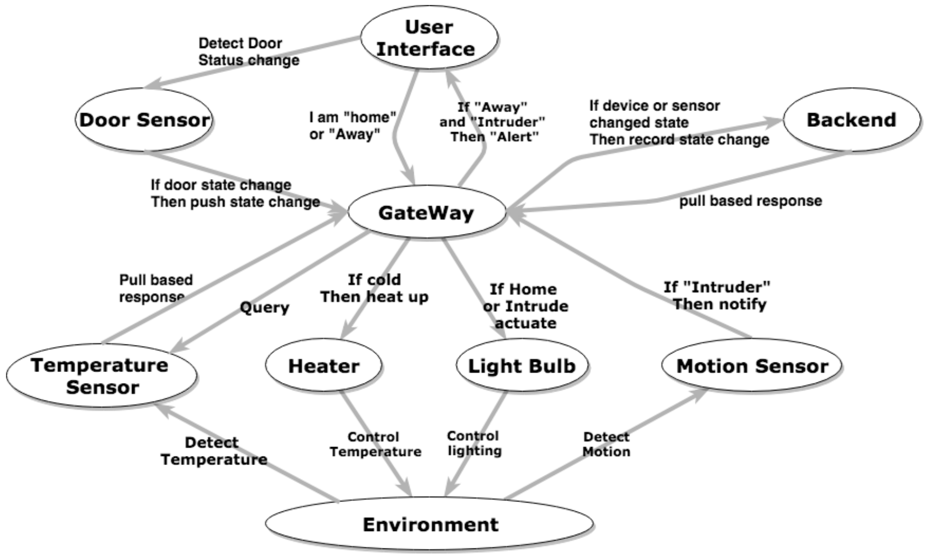


Figure 1: The overall design. Circles indicate separate processes and the arrows indicate the communication between the processes. The process with the tail of an arrow indicates the sender and the head indicates the receiver.

the difficulty of the implementation of this project revolved around the leader election algorithm (i.e. the Berkeley clocks algorithm) and the other additions were quite straightforward. In stark contrast the Lamport logical clocks implementation, despite being an ingenious solution to the synchronization problem, involves very little effort in its implementation. All that was needed was a simple alteration to all of lines of code involving recieving messages.

The backend is a process which imitates a database of all of the state changes of all of the devices and sensors. The backend device is simulated by a process which constantly receives messages from the gate and checks if the message indicates a change or not; if there is a change the database both stores the message and the time of the change of the state and also updates the new status in a registry containing the up-to date states of all of the devices/sensors. The door sensor is simply a sensor that detects movement of the door and is similar to the implementations of sensors in assignment 1.

2 Leader Election Algorithm

The leader election algorithm uses the famous algorithm suggested Gallagher by et al. in [GHS83] for general undirected graphs (which itself has had a strong impact on the design of distributed algorithms in general, and won the Dijkstra Prize for an influential paper in distributed computing) to create a minimum spanning tree with the root as a leader. We would like to emphasize that this algorithm produces the **optimal** solution to leader election algorithm for any general graph (i.e. the graph can have any topology as long as it is connected).

Given the complex nature of the algorithm we present the pseudo-code of the implementation we performed (which differs from the original algorithm in a couple of ways) and present figures (Fig. 2a-Fig. 6d) of the execution/test-run of the code in the a particular instance. The figures (Fig. 2a-Fig. 6d) make the operations more intuitive and easy to understand.

Technically before the algorithm itself starts the node/processes send each other test messages in order to test the communication latency of each edge they then label each edge with the worst case communication latency and use those edge weights to perform the leader election algorithm. This step is called create_edges(name, seedy,neighbors =[]) in leaderelection.py.

Essentially the leader election algorithm works as follows: Every node/process begins by choosing the minimum edge weight and waiting for a response as shown in Alg. 2 (which is the first routine called in Alg. 1).

Algorithm 1 Find_MST(edges = [], my_name)

```
1: my_rank, children, parent, status = Initialization_Step(edges, my_name)
2: while edges != ∅ or children != ∅ do
3:     if status == "follower" then
4:         my_rank, children, parent, status = Follower_Task (edges , my_name)
5:     if status == "leader" then
6:         my_rank, children, parent, status = Leader_Task (edges , my_name)
7: if status == "follower" then
8:     parent.send("I am finished")
```

The entire algorithm is essentially performing this operation over and over again but the difficulty lies in coordinating a large city of nodes to behave as one large unified node. This is achieved by a constant question/answer protocol between the leaders and its followers or more generally (i.e. the recursive equivalent) a question and answer protocol between a parent and his children. The first questions are asked in Alg. 3, where the parent constantly queries its children asking them for their minimum edge and they reply with either their min or a declaration that they have run out of territory to explore, in the latter case the parent marks the child as finished and no longer queries them.

During this process (line 11 in Alg. 3 or lines 107 and 224 of leaderelection.py) the parent can discover a cycle. Handling this logical case is one of the most subtle difficulties encountered in the implementation.

Algorithm 2 Initialization_Step(edges = [], my_name)

```
my_rank == 0, children = []
min_edge = find_min(edges)
min_edge.send_message("I choose you")
mail = min_edge.wait_on_message()
if mail.data == my_rank and mail.from < my_name then
    children.append(min_edge)
    edges.remove(min_edge)
    status = "leader"
    my_rank = my_rank + 1
else if mail.data > my_rank or mail.from < my_name then
    parent = min_edge
    edges.remove(min_edge)
    status = "follower"
return my_rank , children, parent, status
```

Algorithm 3 City_wide_search(edges, my_name)

```
min_edge = find_min(edges)
for child in children do
    if did_i_find_find_cycle == FALSE then
        child.send_message("search")
        child_mail = child.wait_on_mail()
        if child_mail.command == "my_min_is" then
            if child_mail.data < my_min then
                old_min = my_min
                my_min = child
                did_i_find_min = FALSE
            else if child_mail.data == my_min then
                did_i_find_find_cycle = TRUE
                if did_i_find_min == FALSE then
                    my_min.send_message("delete cycle")
                    child.send_message("delete cycle")
                    my_min = old_min
                else if did_i_find_min == TRUE then
                    edges.remove(min_edge)
                    child.send_message("delete cycle")
                if old_min ∈ edges then
                    did_i_find_min = TRUE
        else if child_mail.command == "i_am_finished" then
            finished_children.append(child)
            children.remove(child)
```

The parent can discover an cycle in one of two ways: a child can have an edge weight equal to his current min or two of his children can have edge weights equal to each others mins. In the former case the parent deletes the cycle edge and alerts his child to the presence of a cycle and in the latter case the parent alerts the two children to the presence of the cycle cases. After the parent must return to the penultimate (second-least min) edge and restart the search. In actuality (i.e. in the actual Python code we wrote) the while city wide search routine (we must restart the whole routine because of concurrency issues, only one decision can be made per search round) restarts but in order to simplify the explanation we have left out little house keeping issues (or else the pseudo-code would be as long and complicated as the actual code which ended up being about 350 lines of code). The parent then sends his minimum edge weight to his parent if he is a follower he then proceeds to perform the rest of Alg. 5. If the node is actually the leader he can then make the final global decision, which is depicted in Alg. 4.

It follows by a simple recursive argument that the leader ends up with the global min of all of the nodes in his city. He then either sends a message to the min edge declaring the intent to merge ("I choose you" line 4 of Alg. 4) or sends a message down to his min-child commanding him or her to merge with the min edge ("You are min" line 6 of Alg. 4) which is recursively passed down to the correct edge who then declares the intent to merge. This essentially a repeat of the very first "Initialization step" but now the entire city works as a large unified node. If the replying city has a larger rank then the smaller city is absorbed, if the replying city has a larger rank then the node's resident city is absorbed and if they are equal then the tie is broken by lexicographic ordering of the names.

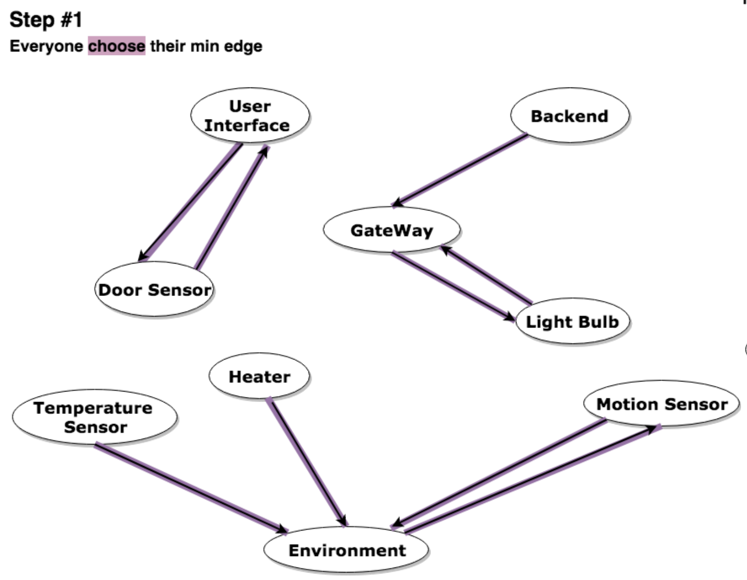
If the ranks are equal then the new city increases its rank by one (as depicted in line 8 of Alg. 4). One intuitive (yet non-rigorous) reason why this works is that if you add 16 to 64 then the resulting log is barely affected but if you add 64 to 64 then the log is increased, i.e. $\log(64+16) \approx \log(64)$ but $\log(64+64) \approx \log(64)+1$. This turns out to be a very cost efficient way of approximating the "size" of any two cities (although it is not actually that because one leader could actually be a giant ring topology, it actually counts the number of "non-min mergers" the city has made and therefore is more like a combined measure of size and latency).

Algorithm 4 Leader_Task (edges = [], my_name)

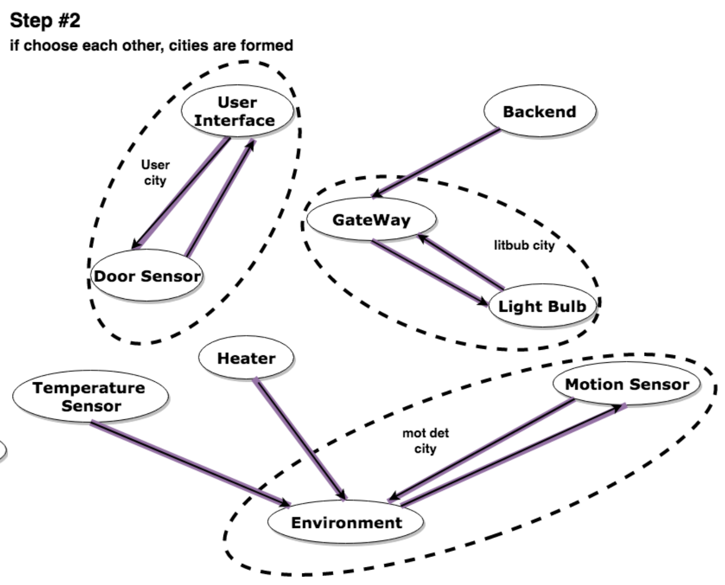
```
my_min = City_wide_search(edges, my_name)
if edges != ∅ and children != ∅ then
    if did_i_find_min == TRUE then
        my_min.send_message("I choose you")
    else if did_i_find_min == FALSE  then
        my_min.send_message("You are min")
    response = my_min.wait_on_mail()
    if response.rank == my_rank then
        my_rank = response.rank + 1
        if response.name < my_name then
            if did_i_find_min == TRUE then
                children.append(my_min)
                edges.remove(my_min)
            else if did_i_find_min == FALSE  then
                my_min.send_message("We won")
        else if response.name > my_name then
            parent = my_min
            my_name = response.new_city_name
            status = "folllower"
            if did_i_find_min == TRUE then
                edges.remove(my_min)
            else if did_i_find_min == FALSE  then
                my_min.send_message("We lost")
                children.remove(my_min)
    else if response.rank > my_rank then
        parent = my_min
        my_name = response.new_city_name
        status = "folllower"
        if did_i_find_min == TRUE then
            edges.remove(my_min)
        else if did_i_find_min == FALSE  then
            my_min.send_message("We lost")
            children.remove(my_min)
    else if response.rank < my_rank then
        if did_i_find_min == TRUE then
            children.append(my_min)
            edges.remove(my_min)
        else if did_i_find_min == FALSE  then
            my_min.send_message("We won")
```

Algorithm 5 Follower_Task (edges = [], my_name)

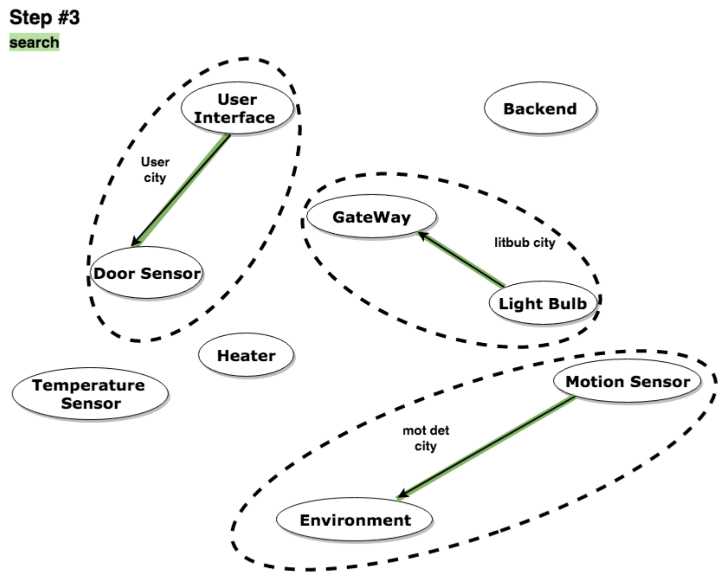
```
orders = parent.wait_on_mail()
my_rank = orders.rank
my_name = orders.new_city_name
**if** orders.command == “search” **then**
    my_min = City_wide_search(edges, my_name)
**if** edges != $\emptyset$ **and** children != $\emptyset$ **then**
    parent.send(“My min is”, my_min)
    response = parent.wait_on_mail()
    my_rank = response.rank
    my_name = response.new_city_name
    **if** response.command ==“You are min” **then**
        **if** did_i_find_min == *TRUE* **then**
            my_min.send_message(“I choose you”)
        **else if** did_i_find_min == *FALSE* **then**
            my_min.send_message(“You are min”)
    response = my_min.wait_on_mail()
    parent.send(response)
    orders = parent.wait_on_mail()
    my_rank = orders.rank
    my_name = orders.new_city_name
    **if** orders.command == “We lost” **then**
        **if** did_i_find_min == *TRUE* **then**
            parent = my_min
            edges.remove(my_min)
        **else if** did_i_find_min == *FALSE* **then**
            my_min.send_message(“We lost”)
            parent = my_min
            children.remove(my_min)
    **else if** orders.command == “We won” **then**
        **if** did_i_find_min == *TRUE* **then**
            children.append(my_min)
            edges.remove(my_min)
        **else if** did_i_find_min == *FALSE* **then**
            my_min.send_message(“We won”)
    **else if** orders.command == “delete cycle” **then**
        **if** did_i_find_min == *TRUE* **then**
            edges.remove(my_min)
        **else if** did_i_find_min == *FALSE* **then**
            my_min.send_message(“delete cycle”)
    **else if** orders.command == “You are not min” **then**
        **if** did_i_find_min == *FALSE* **then**
            my_min.send_message(“You are not min”)
```



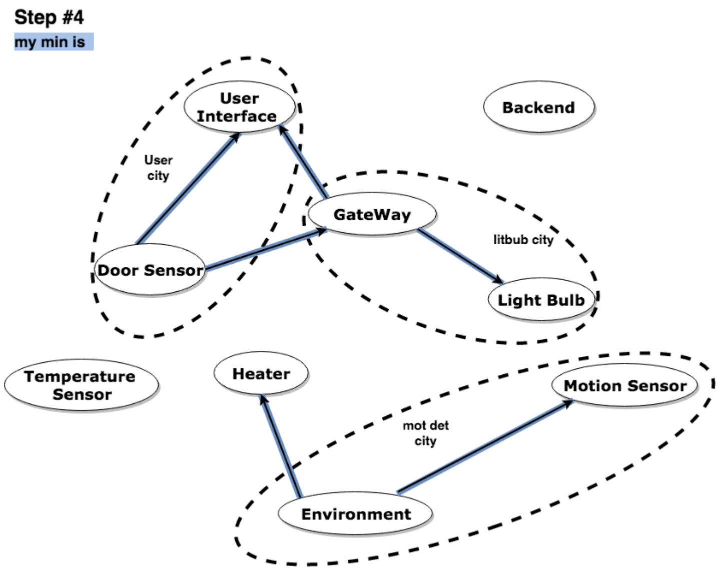
(a)



(b)

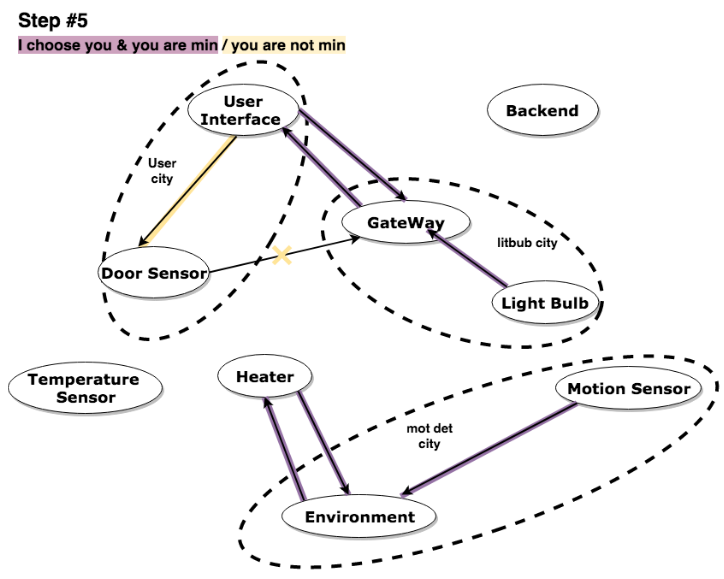


(c)

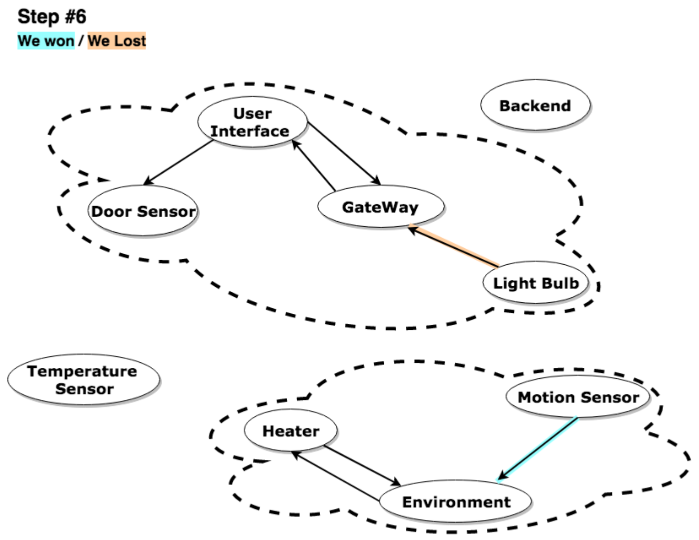


(d)

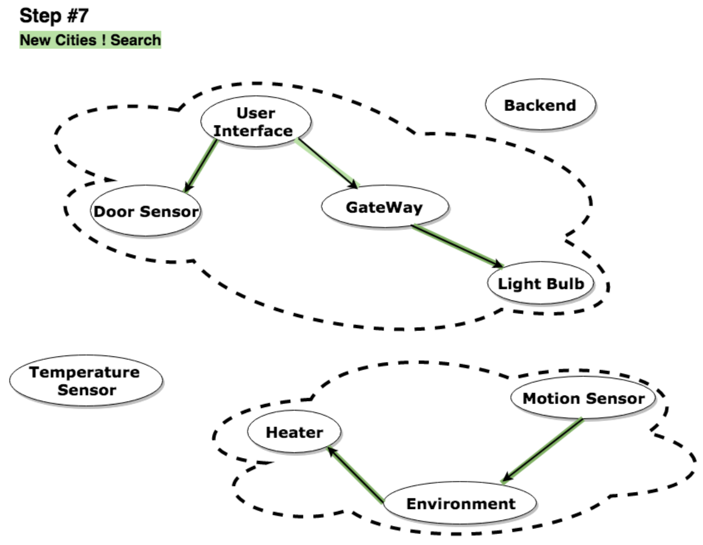
Figure 2: Leader Election Procedures: Step 1 - Step 4



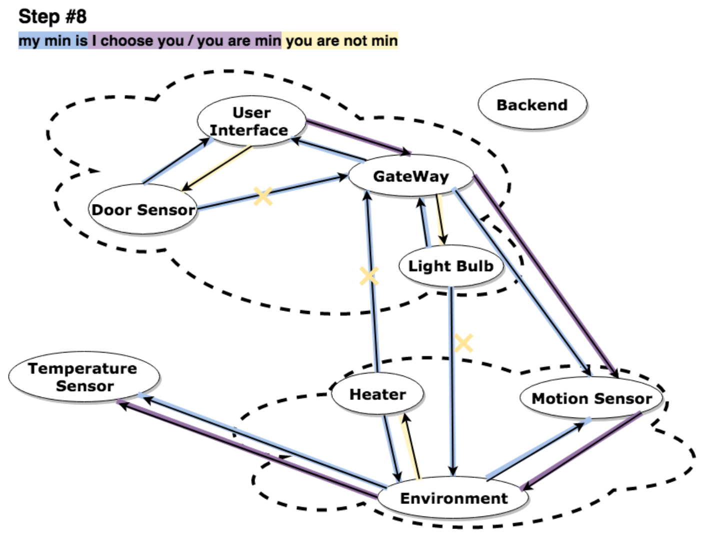
(a)



(b)

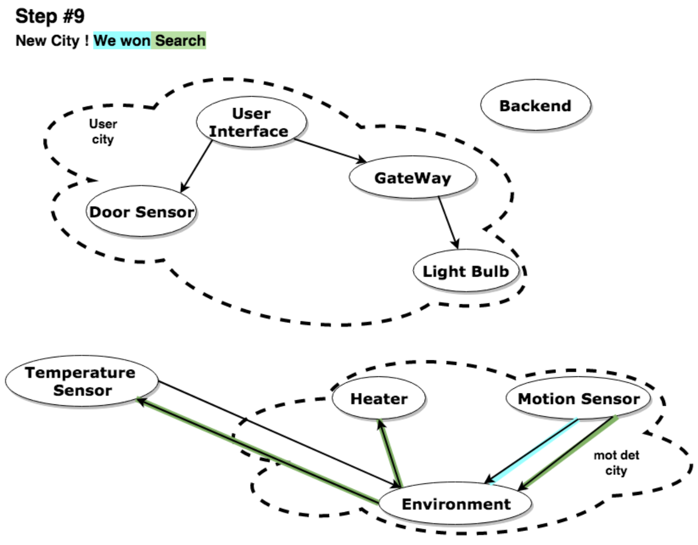


(c)

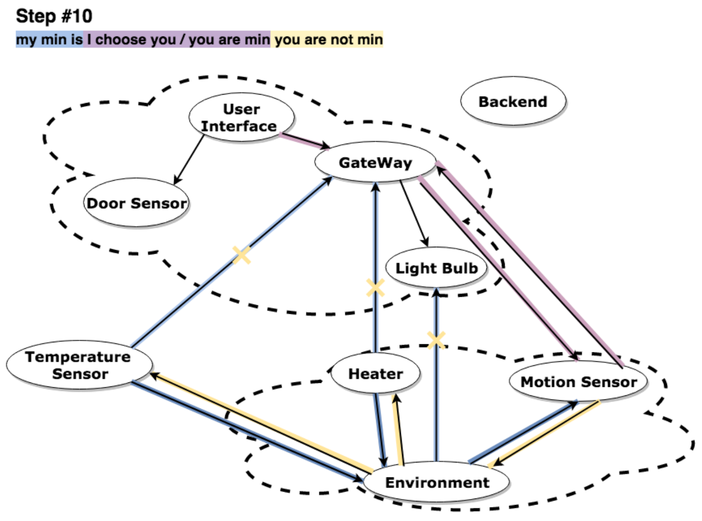


(d)

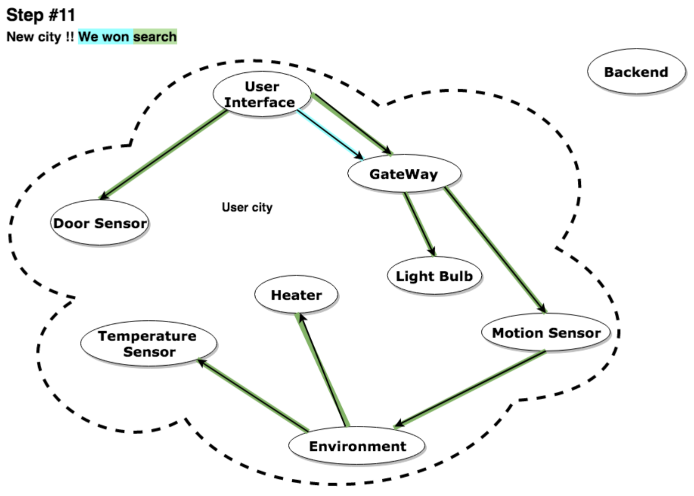
Figure 3: Leader Election Procedures: Step 5 - Step 8



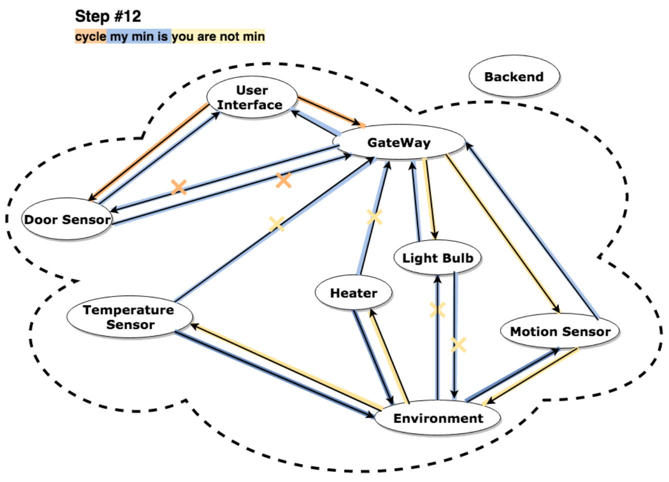
(a)



(b)

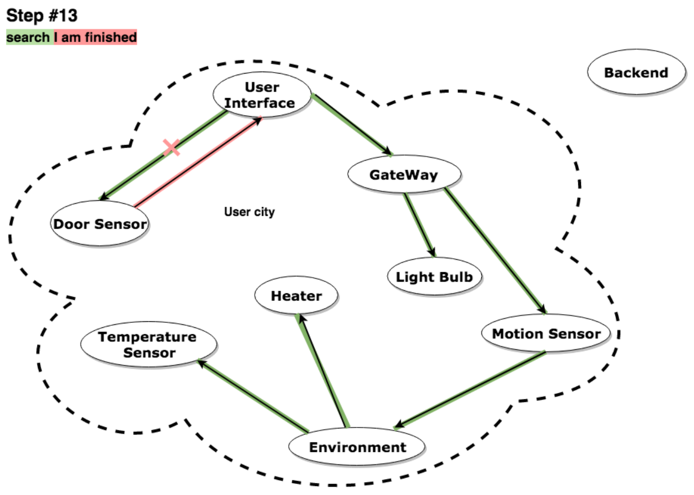


(c)

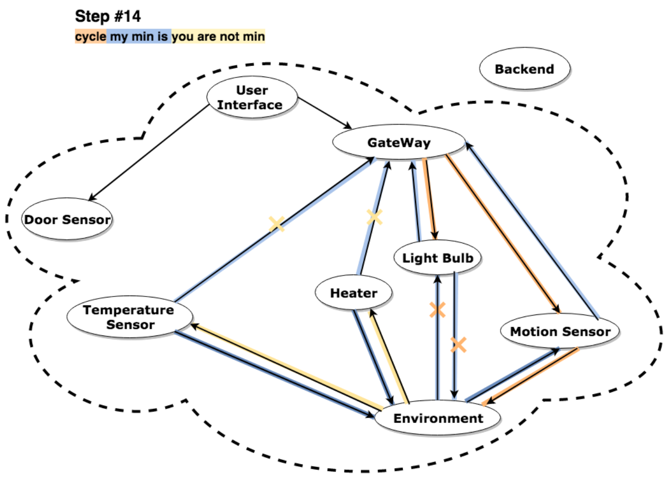


(d)

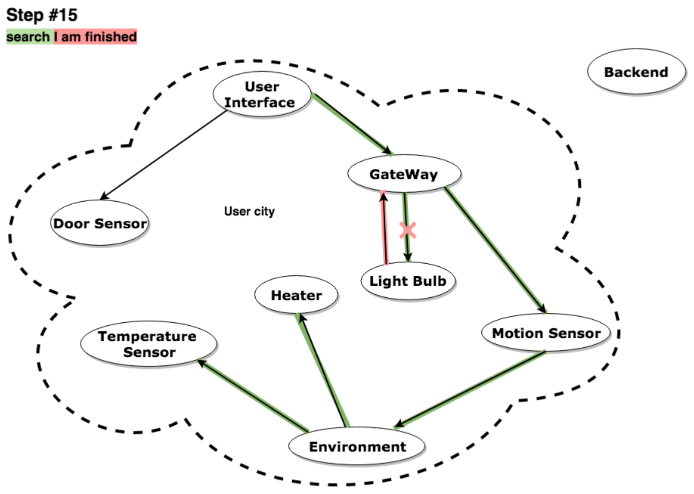
Figure 4: Leader Election Procedures: Step 9 - Step 12



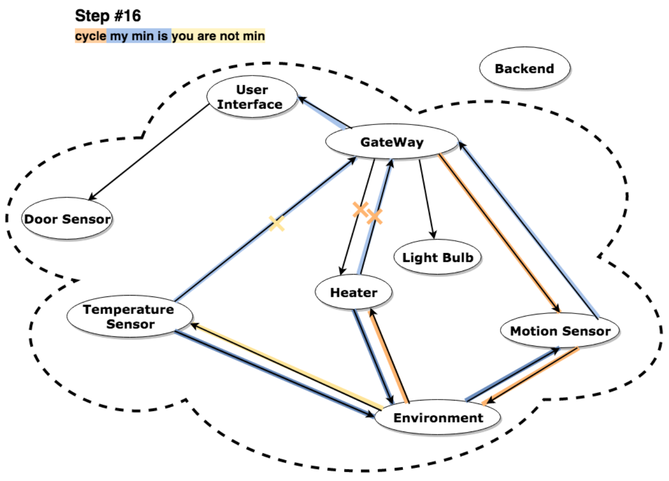
(a)



(b)

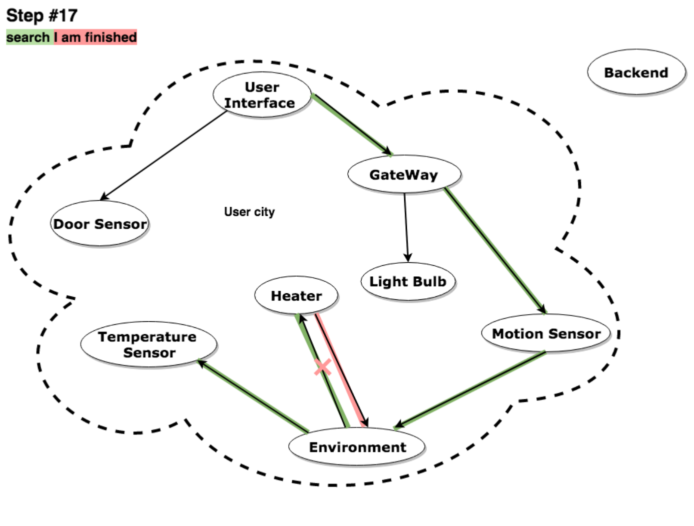


(c)

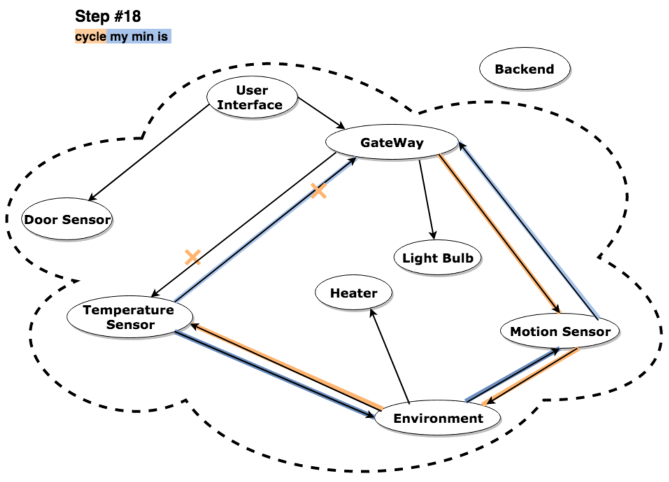


(d)

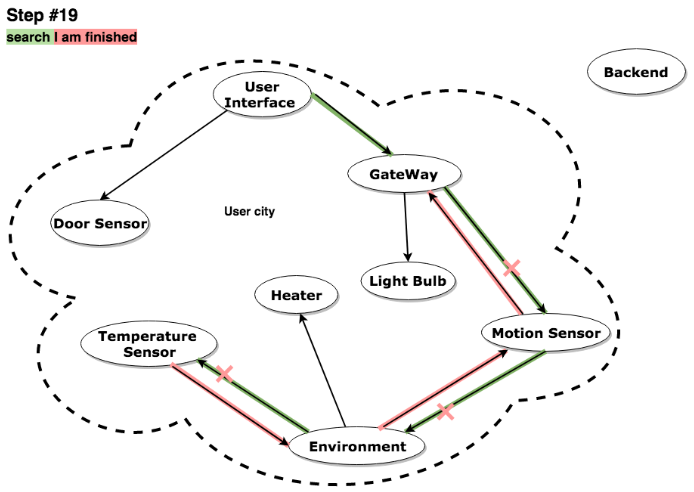
Figure 5: Leader Election Procedures: Step 13 - Step 16



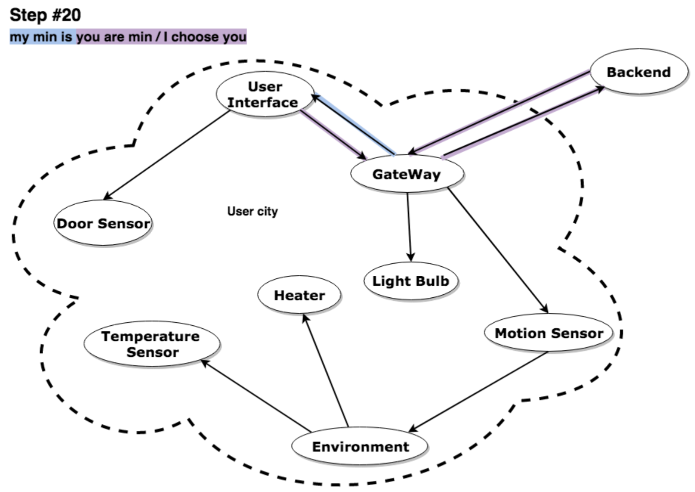
(a)



(b)

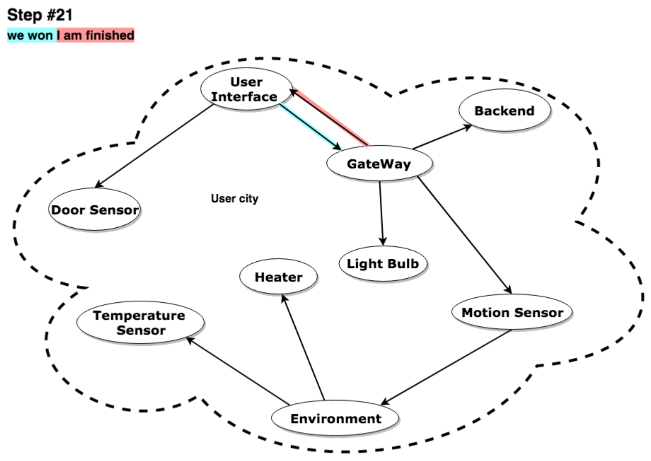


(c)

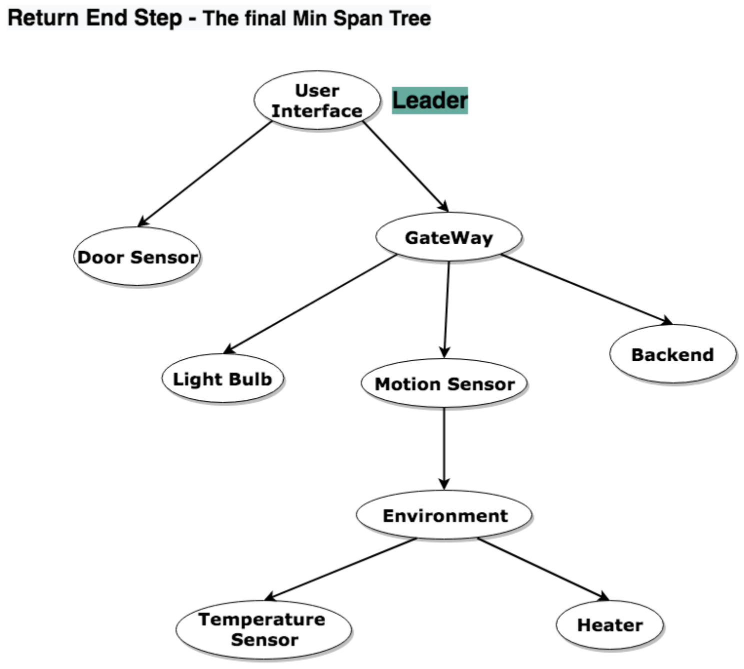


(d)

Figure 6: Leader Election Procedures: Step 17 - Step 20



(a)



(b)

Figure 7: Leader Election Procedures: Step 21, Final Step

3 Berkeley Algorithm

Because we have created a minimum spanning tree in our leader election algorithm we can now exploit the topology in order to optimize the Berkeley algorithm! The key idea is that the leader does not have to compute the average of all of the times of the nodes or compute all of the offsets of all of the nodes. As a matter of fact he only needs to do so for all of his children nodes. if the maximum branch out factor, call it b, of the tree is small enough then each parent node will only perform $O(b)$ many operations as opposed to $O(n)$ many operations. For a small enough b this reduces the complexity of the algorithm from $O(n)$ to $O(b\log_b(n) + w\log_b(n))$ where w is the maximum communication delay (which itself has been optimized because we choose the minimum spanning tree). This distributes all of the tasks out evenly among the nodes of the graph.

To see how this works lets consider the simple case where we have a leader with 2 children and $n-3$ grandchildren. Let $a_1, ..., a_k$ be the left grandchildren, $b_1, ..., b_j$ be the right grandchildren, c be the left child's time, d the right child's time, and t be the leaders time. The leader could have taken the average in the following way

$$\text{Average time} = \frac{a_1 + ... + a_k + b_1, ..., b_j + c + d + t}{n}$$

which would be highly inefficient. Suppose that instead the left child had already computed

$$\text{Average}_{\text{left}} = \frac{a_1 + ... + a_k}{k}$$

and that the right child had already computed

$$\text{Average}_{\text{right}} = \frac{b_1, ..., b_j}{j}$$

then the leader only has to compute

$$\text{Average}' = \frac{k * \text{Average}_{\text{left}} + j * \text{Average}_{\text{right}} + c + d + t}{n} =$$

which equals

$$= \frac{a_1 + ... + a_k + b_1, ..., b_j + c + d + t}{n} = \text{Average}$$

i.e. the correct average. A similar argument shows that the leader does not need to compute all of the latency because

$$\text{Comm. Delay}(\text{leader}, \text{grandchild}) \approx$$

$$\approx \text{Comm. Delay}(\text{leader}, \text{child}) + \text{Comm. Delay}(\text{child}, \text{grandchild})$$

This exactly what was performed in the implementation found in berkeley.py.

4 Lamport

As was stated earlier the Lamport logical clocks was a relatively straightforward implementation. All that was done was a simple alteration to all of lines of code involving receiving messages. What is gained in amount of work is lost in terms of results. The Berkeley algorithm actually syncs the clocks, whereas the Lamport algorithm only *partially* orders events that need to be logically ordered.

5 Design Tradeoffs

There were many issues encountered with the leader election algorithm. How to handle discovering cycles was one of the biggest issues. eventually we were forced to make the edge weights unique using calls to random.random() to resolve the issue. Another issue was that the queue/broadcast style of communication had to be abandoned for a endpoint-to-endpoint non-blocking communication style[1]. The non-blocking style was a necessity so as to avoid any synchronization issues. As explained earlier every city must make one decision per call to Alg 3 (city_wide_search()) so as to avoid any issues. That is because the algorithm has a basic question answer protocol that recursively travels up and down from parent to child. If this protocol is broken then undefined behavior could occur.

Another issue was flushing out old messages from the communication pipes which could also create undefined behavior. This is why we have seemingly redundant/unnecessary Boolean checks such as

```
if orders.command == "search":
```

in line 87 of leaderelection.py and

```
if orders.command == "lets_see_how_fast_you_are":
```

in line 33 of berkeley.py. Also the byzantine nature of the large number of if elif statements in the code made for many logical errors.

5.1 Testing

The code itself was tested extensively. There were times were if the code was tested, it would only cause errors 1 out of 30 times. The code was hand-tested by drawing out the steps performed by each of the nodes and resulted in something between 50-100 hand-drawn trees (complete with color coded messages) to find exactly were each error occurred.

Further one of the authors of the algorithm went as far to write a proof of program correctness. The code itself has been tested well over a 1000 times and is very unlikely to have any bugs[2].

[1]This essentially amounted to having to rewrite many of the previous communication protocols from the previous assignment.

[2]With the exception of the potential of undefined behavior if two edges have the same

6 Possible improvements and extensions

One possible improvement would be to remove the unique edge weights assumption. Doing so would increase the amount of code by a large amount (leaderelection.py is already 350 lines long).

One way to remove this assumption efficiently would be to introduce another improvement which would be to allow for blocking communication to occur. Doing so would be extremely difficult if not impossible given the synchronization issues stated earlier; basically if we allow blocking communication then it would break the question answer protocol structure of the algorithm, forcing one to bascally implement a whole new version of the algorithm.

Therefore the most likely form of doing away with the unique edge weights assumption would potentially involve many more lines of code and larege amounts of elbow grease (because of the potential necessity of non-blocking communication.

7 How to run it:

Enter the "IOT_project_multiprocessing_style" in your directory and type in

- python3 main.py

(you must run on python 3.6 or later, python 3.7 may give you warning about time.clock that you can safely ignore).

References

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983.

weight, which is as likely as being stricken down by lighting 100 times in a row because the weights w are randomized by performing the transformation $w \to 2^{rw}$ which creates a exponential difference and furthermore each process is given a unique random seed to plug into r forcing uniqueness to be statistically guaranteed.