We need to talk!

Ishupreet Singh (SBU ID: 111424300)
Chinmaya Dehury (SBU ID:112077747)
Toby Babu (SBU ID: 111463549)

Contents

Introduction

In recent years the use of wireless sensor networks to solve common problems has been exponential. From structural health monitoring (Yuan et al. 2017), underwater marine life monitoring (Xu et al. 2014), military and security surveillance, health-care, smart homes, automotive industry. A typical sensor network would consist of hundreds of sensor nodes deployed over a geographical locality. A single sensor node will have some obvious limitation in terms of memory, battery power, computation, and communication capability. On the other hand, a group of sensor nodes working together can perform bigger tasks easily.

Parking is a huge issue in any metropolitan area. And with the increasing growth of the automotive industry, the demand for intelligent parking service is expected to grow rapidly in the near future. A driver, on average, spends 17 hours a year to find a parking spot in the US and 107 hours a year to find parking in NYC (Anon n.d.). This obviously is a huge problem. Our proposal is to have a mechanism which would enable parking lots to monitor the vacant spaces in the lot and update them once they are filled in real time. The solution should be feasible and cheap enough so that it can be implemented in any parking lot. Currently, parking lots are either manned where a driver needs to inquire on the possible vacant lots or unmanned where the driver will have to drive around the lot to find a parking space. Another common approach to solve this problem is to use video cameras but video cameras based solutions are energy inefficient and the amount of data becomes difficult to process and transfer over multiple hops in a network.

In our project, we try to solve this problem using proximity sensor coupled with WiFi enabled microcontroller. We implement a network with nodes which represent parking spots. The central system and the nodes would form a tree structure. When a node is vacant/filled, it will contact its parent with an update until the update reaches the central system. The central system pushes this information to the internet. And in this way, our system would enable drivers to access the spaces within each parking lot. This makes our method better than traditional methods being used. Moreover, information gathered by each node can be collaboratively processed in a distributed or centralized way to evaluate other meaningful metrics such as duration of parking, automatic billing, and payment, etc., to the benefit of users and administrators.

Protocol

ESP8266 is a device which can connect to a WiFi network and broadcast its own WiFi network as well. The system will have a root device which will connect to a WiFi network that can connect to the internet. This root device will send the status of it's nodes to an HTTP endpoint which will be hosted in the network. As soon as the system is turned on, the root device will connect to the WiFi network and once connected, it will broadcast its own WiFi network. The SSID of the broadcasted WiFi network will have a predefined nomenclature so that the other devices, which would be child nodes, can recognize it. The child nodes will be continuously scanning for a WiFi network matching the SSID nomenclature, and once found will initiate a connection to it and start broadcasting their own WiFi network which helps its children to connect to it.

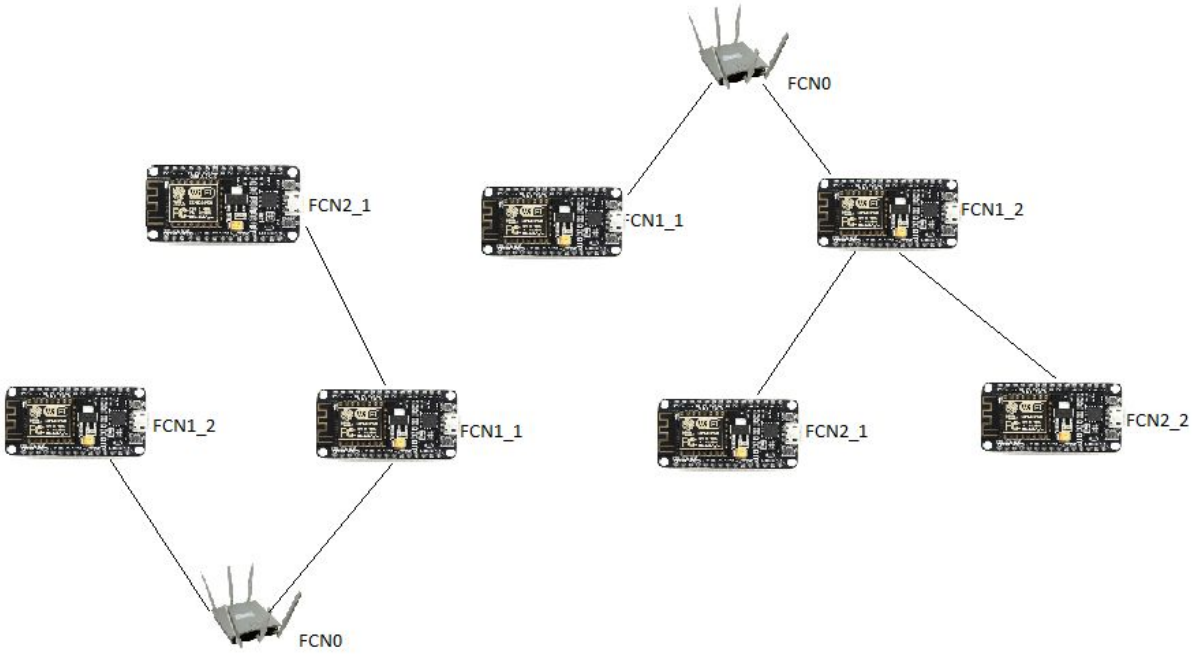

Figure 1:Example topology of a possible network

Let's take the example of figure 1. Suppose we have two access points available and the nodes are sprinkled around in the area. Suppose, 4 nodes are in the range of these access points and they detect the WiFi network, they will connect to the strongest network available. Once they connect, they themselves will start broadcasting a WiFi network. All those nodes that are in the range of those WiFi networks, would connect to them and start broadcasting their own networks and this effect would keep on percolating down till all the nodes are connected. While connecting, these nodes will negotiate the names of their SSIDs depending on their level in the tree.

We use a common prefix for all the SSIDs, after the prefix we have the level number and after the underscore, we have the number of the child connected to the child's parent.

Fault Tolerance

The benefit of such a tree-based design is the flexibility of having many nodes with a minimal configuration which would recognize its relationship with other nodes and create a network of its own. But such a design would also need to consider situations where a node is brought down either due to faulty chips, battery failure or just manual breakdown.
When a parent malfunctions, the device will stop broadcasting it's WiFi which will result in its entire child tree to be cut off from the main network. During this scenario, the child will break off from its previous network and scan for other networks. If a suitable network is found, then the child will connect to it and initiate the part 2 all over again. There are various issues to be considered here. For example, if a node goes down we need to make sure that it does not connect to any of its own children. This could cause a cycle to be created and would result in an infinite request/response. To handle this problem our initial idea is to establish a strict hierarchical based nomenclature of SSID. In this hierarchy suppose the parent has an SSID of "1" then its child nodes will have SSID of "<parent_prefix><levelnumber>_1", "<parent_prefix><level_number>_2" and so on. So a third level node will have an SSID like "<parent_prefix>3_1". To facilitate this we need to maintain a count of the child nodes connected to it. If the parent prefix is x, the child count is 3 and the level of the parent is 1 and a new node gets connected its name will be x2_4. When a parent goes down, all its children will first get disconnected from it's parent's WIFI. The device will now start searching for other parents whose levels are higher than that of the device. On finding a new parent, the level and the id of the device will be re-negotiated and the device will have a new id.

API Design

The ESP8266 acts as an HTTP server and an HTTP client. Each action between a parent and child will be HTTP request-response. We created multiple APIs to work with the different actions of each device.

1. Heartbeat: It is imperative for a child to know if its parent is up and running and if it is connected to the main network. If this doesn't happen, any status changes or actions from any of the children will be lost. To rectify this, we have created a heartbeat API which is an HTTP request sent from the child to the parent every 5 seconds. The time of 5 seconds was measured using the clock frequency of the device and measuring the clock ticks to reach 5 seconds. A heartbeat request would be served by the parent would respond with a "Good" value. If the child gets back this value, the parent is

assumed to be up and running. If the child does not get a response back, it senses that something is wrong. We retry this 5 more times and if no response comes back even then, the child disconnects from the parent's WiFi and goes to a rebalancing mode.

2. getChild: A tree network will work without issues only if a child can identify which of the nodes' are its parents and which are its siblings and which are the children. We use a naming convention to get through this. On connecting to a parent, the getChild API will give you a token which signifies the level of the node and its sequence with respect to its siblings. This token will be used to create the SSID of the child so that the child can serve other devices.
3. sendToRoot: The core part of this network is to have a mechanism through which status updates can be communicated from any device to the outside world. This was resolved using a sendToRoot API which would be triggered by a child whose status has changed to its parent. The API is recursive and would make the parent trigger it to its parent and so on till it reaches the root. The root is connected with the internet and has access to the outside world.

Evaluation Setup and Testing

The Test plan for this project revolves around data reliability, speed, fault tolerance, and load testing.

Data reliability will also be measured from the perspective of the root node. If a leaf node changes its status, then does that result in a notification at the root node.

This scenario is handled by the sendToRoot API discussed in the previous section. The API endpoint would be recursive and sends the status of the leaf up the tree and to the root.

Fault tolerance will be measured by removing random parent nodes and making sure the root node still shows the correct status of the network.

This was tested as part of the convergence testing of the network. We set up a root node having 2 children and one of its child having 2 grandchildren. Now the node having the grandchildren is disconnected from the network. This would make the network go into a rebalancing state which would be indicated through the heartbeat API calls. The grandchildren would start looking for WiFi APs to connect to. It will not connect to each other realizing that they are at the same level. It will either connect to the child having no children or the root node. This decision would be made using the signal strength. We found that the entire network rebalances itself with healthy heartbeats from all its available nodes in 30 seconds. But this time does not create too much of an impact because the stability of the network is being checked every 5 seconds. So the network will quickly realize if something has gone wrong and corrects itself.

Load testing can be tested by generating large amounts of data and sending it to the root through one of the children.

Due to unavailability of a large number of devices, load testing was simulated by creating a program which will connect to the WiFi of a parent and spawn multiple threads and invoking the sendToRoot API request from each of the threads. Using this we can measure the capacity of a node to handle a large number of requests. The testing started with 10 parallel requests and ended with 150 parallel requests. We found that requests were getting dropped or getting responded really late during the range of 100 - 150 requests. Till this limit, all the requests were responded in a timely manner. In a real-time scenario, this limit is more than enough because 100 parallel requests to a node mean that 100 parking lots are being free at one instant of time which is pretty unlikely.

The creation of the evaluation setup of 5 nodes with 5 sensors connected to each of the nodes cost 20$.

Energy Consumption

We used the available online resources to find out the amount of energy used by the nodes. We found the following energy table from this source. (Anon n.d.)

Parameter	Typical	Unit
Tx 802.11b, CCK 11Mbps, P_{OUT}=+17dBm	170	mA
Tx 802.11g, OFDM 54Mbps, P_{OUT} =+15dBm	140	mA
Tx 802.11n, MCS7, P_{OUT} =+13dBm	120	mA
Rx 802.11b, 1024 bytes packet length , -80dBm	50	mA
Rx 802.11g, 1024 bytes packet length, -70dBm	56	mA
Rx 802.11n, 1024 bytes packet length, -65dBm	56	mA
Modem-Sleep	15	mA
Light-Sleep	0.5	mA
Power save mode DTIM 1	1.2	mA
Power save mode DTIM 3	0.9	mA
Deep-Sleep	10	uA
Power OFF	0.5	uA

We used the ESPs in IEEE 802.11n mode so that the power consumption is low. As shown in the table above, the current used in the Transmission mode is 120mA and in Receiving mode is 56mA. Since the ESP works on 5V supply the power usage is 5*120 = 600mW and 5*56 = 280mW.

There are two ways in which we can implement the data transfer protocol:

1. Periodic Pings: We have configured the nodes to send pings at every 5 secs. In this mode, the power consumption would 600mW and 280mW for transmission and reception as shown above.
2. Triggered Data Transfer: In this mode, the data is sent only when there is a state change in any of the sensors. In between state changes we can put the WiFi circuitry to sleep using the Deep-Sleep mode. In this mode, the power usage is 5*10 =50 uW, which is a significant energy saving.

Future Work

The experiments we have conducted shows that such a design is feasible, cheap and scalable. But there is still scope for improvements.

1. Security: Security was not a prime consideration during the design of the network. The key for the WiFi was pre-shared and is stored in the HTTP server as plain text. A better way to go

about this would be something similar to home routers which have a default password associated with it. It will also have a web page where we can update the password of a device.
2. Contention with other WiFi networks: We are creating a large WiFi network where each device will be broadcasting its own WIFI. But what would happen if another device comes in range having the same SSID as one of the nodes. A certain degree of randomization would be needed and maybe the use of a pre-shared GUID prefix rather than a constant prefix.
3. Optimization of status change requests: The status change requests are currently sent as part of the heartbeat as well to maintain the stability of the network. But this could be optimized so that these requests are sent only during a change of the status of the parking lot.
4. Failure of root node: One of the issues we haven't addressed is the scenario where the root node goes down. In such a case we will have multiple children of the same level and one of these nodes need to be promoted to the parent. A design decision needs to be made where the lowest id of these will be promoted to the parent and that will cause all the other children to get connected to this new root.

Code

The code is available at this location.
We have taken help from the following links:

1. https://github.com/esp8266/Arduino/blob/master/doc/esp8266wifi/udp-examples.rst
2. https://randomnerdtutorials.com/esp8266-web-server-with-arduino-ide/
3. https://stackoverflow.com/