Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 20-001

Constellation Plan B Report

Aravind Alagiri Ramkumar, Rohit Sindhu, Jon Weissman

January 3, 2020

Constellation Plan B Report

Abstract →

Data explosion has been exponential in the last decade. A new and major addition to this phenomenon is the Internet of Things (IoT). With the technology becoming cheap and affordable, there has been a boom in the number of smart devices which generates huge amounts of data. All of these devices may have varying connectivity and are made of diparte designs. IoT is one platform which can help us make use of this huge amount of data being generated by thesis heterogeneous devices. Since, these devices are very different in design and architecture, we propose a system which provides a unified view to this heterogeneous world. We propose a dynamic P2P and edge based system named Constellation. This work presents the work pertaining to creating this network, the inherent dynamism involved in such a system and ways to utilize the system to provide useful analysis by executing global and local tasks.

Introduction →

There has been a huge increase in data being generated globally and most of it is being generated at edge systems or low power IoT devices. Majority of these devices are either mobile, have low bandwidth connectivity and transient connections. Also, since these devices have been developed over a period of time and from various organizations / vendors / individuals, the designs and architectures is quite differing. The only way to counter such a disparity and dynamism is to ensure a system has such concepts built in to be dealt with. Internet of Things(IoT) is one of the latest attempts to counter this problem. An IoT system aims to optimally discover, use and manage numerous disparate computing devices for accomplishing global tasks. Though IoT systems are aimed at resolving the disparities of computing world, they themselves face similar issues. Up until now all traditional IoT systems have been made, designed and operated specific to an enterprise/ organization or vendor. This leaves very little leeway for extendibility and support from the community. Also since, a single organization can not (and should not) design, sell and operate all types of devices, it is very unlikely that any of these systems can support the whole ecosystem. In addition, all the major IoT systems (AWS IoT, Google Brillo, Azure IoT, etc) have central control systems which is not scalable when the number of devices increases exponentially to billions in recent future. Also, not all the devices and areas have good connectivity to the cloud or may only have local connectivity. Some of these examples are farms, forests, Bluetooth, Zigbee, etc. There has been a lot of work by using constrained devices in remote areas to service them or increasing efficiency. For example, FamBeats system from Microsoft Research help increase and optimize yield of remote and large farms. Though it uses an edge based approach to solve dynamics and connectivity issues of network, it still rely on central cloud control. IoT in itself is a dynamic and booming field of research and development, with new protocols and standards are being published very rapidly. This calls for a revolution one like the internet which can help connect all the devices this time. We need a generic and dynamic IoT system. Two most advanced ways of global and wide area distributed computing are P2P and edge computing. So a peer to peer edge based IoT system does better fits the requirement of a generic and dynamic IoT system. Constellation is designed to be the same and made up of two major components - Edge devices and IoT devices. Edge nodes make a P2P network and each one manages a disjoint set of IoT devices and communicate with its neighbors to accomplish any global task.

System Components →

Edge Node

An edge node is any machine that helps in the managing and querying the IoT devices. It can be a desktop, laptop, mobile phones, Raspberry PI etc. Multiple edge nodes connect with each other to form the P2P constellation network. When an edge node is queried, it collaborates with the other edges in the peer-to-peer constellation network to perform a global or local task.

IOT Device

IoT devices are light weight machines or devices which listens for messages and performs an action based on the message. In our setting, only WiFi based IoT devices were used. It can be easily extended to bluetooth, NFC, etc.

Client

Client is someone who uses Constellation for performing tasks using data being generated. Client fire queries based on custom CQL language which then compiled, interpreted and sent to the nearest edge node for execution. The final result is collected and relayed back to Client.

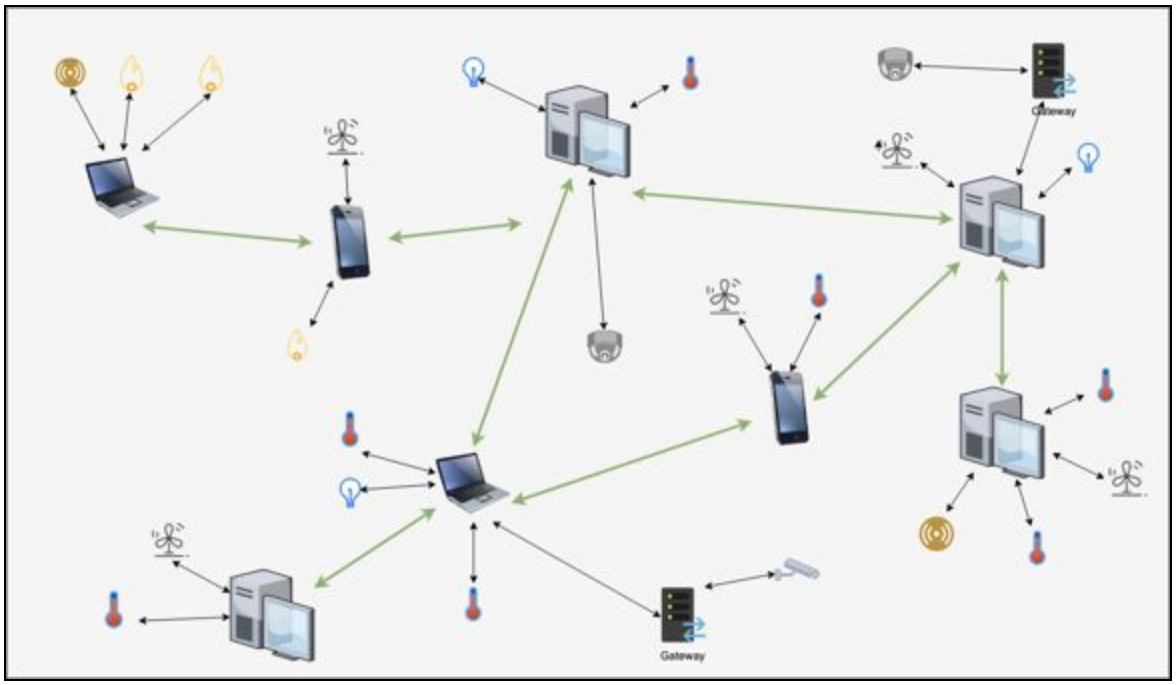


Figure 1 -- Constellation Network

Figure 1 shows a Constellation network. Edge Nodes make a peer to peer network as shown by green line connections. Each edge node owns the control of a disjoint set of IoT devices as shown by black line connections.

Constellation Network Creation →

In this section, we focus on some of the research questions in creating and managing the constellation network such as -

a. How to configure the network given a set of edge nodes and IoT devices?
b. How to add edge nodes and IoT devices dynamically?
c. How to handle failures of edge node/IoT devices?

Discovery Mechanisms

The process of adding a new edge node or an IoT device to an existing constellation network is called discovery. In the following sections, we assume that all the edge nodes and IoT devices are in the same network and the messages/requests from an edge node or an IoT device will reach at-least some of the other edge nodes or IoT devices in the network.

Edge Discovery

Whenever a new edge node wants to join an existing constellation network, it performs a sequence of actions to find the neighbors which are as follows:

1. New edge node broadcasts UDP request and waits for a certain time interval.
2. The existing edge nodes receiving the broadcast message responds with its metadata including number of neighbors, current load, location etc.,
3. After the wait time interval, the new edge node processes all the received metadata and selects the appropriate neighbors based on the selection policy such as random, location based or load based.
4. While selecting the neighbors, the new edge node sends a connection request to join as a neighbors to the selected edge node.
5. On receiving the connection request, the existing edge node evaluates if it can take more neighbors based on its neighbor selection policy, load, etc. and then responds with success / failed response.

Refer to the Appendix for detailed pictorial representation of the above steps. This process is done repeatedly by the new edge node until it is part of the constellation network. Even existing edge nodes will try to discover new neighbors using this process, if they have neighbors less than a threshold. This helps make sure there are no network partitions.

Bootstrap Nodes and Network Partition Scenario

A special case that needs to be addressed during the edge discovery is to avoid the formation of partitioned networks. Let's take a scenario in which 12 new edge nodes are trying to form/join the constellation network and every edge node has threshold of maximum number of neighbors as 2. In this scenario, every node will be running the discovery sequence and there is a good chance that the new nodes might communicate to each other and form single/multiple islands of constellation network within themselves instead of joining the existing network. Figure 2 is a pictorial representation of the above scenario.

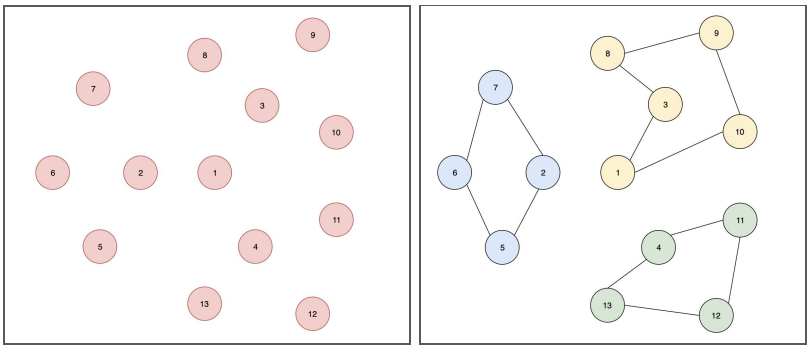


Figure 2 -- P2P network partition

In order to avoid such a scenario, we must restrict that only nodes that already part of constellation should respond to UDP broadcasts. But initially, there will be no nodes as part of the constellation to respond to the new nodes. Hence, we introduced the concept of **bootstrap nodes**. These are the nodes which are always running and already formed a constellation within themselves. In this setup, the bootstrap nodes will respond to the new nodes and add them to the constellation network. Once those nodes are part of the constellation, they can also start responding to the other new nodes and the network can gradually grow. Figure 3 shows how the bootstrap nodes helps to prevent the partition scenario where **node - 1** is the bootstrap node.

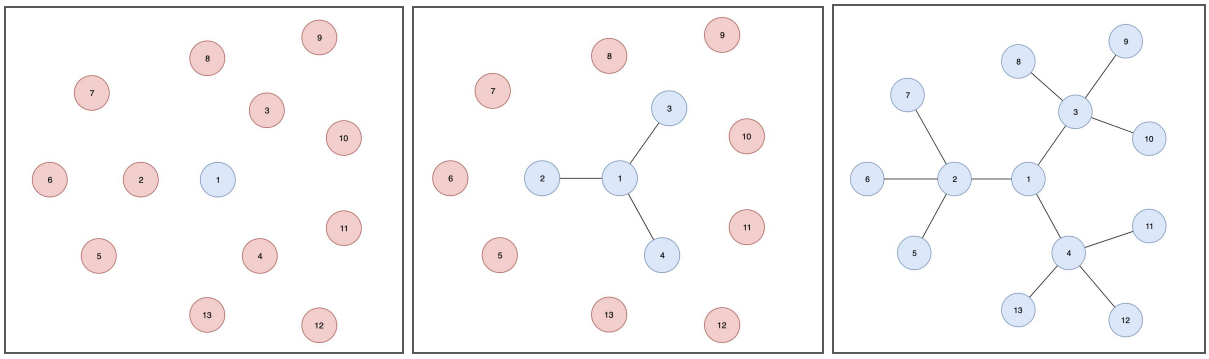


Figure 3 -- Bootstrap Nodes

IoT Discovery

Constellation has a set of requirements in case of IoT devices. One IoT device should only be managed by only one edge node in the constellation network and the IoT devices should be agnostic of which edge node is managing it. Similar to the edge discovery, whenever a new IoT device wants to join the constellation, it broadcasts the UDP message to the whole network. But in this case, the IoT device cannot decide which edge node manages it as it is does not have much compute and also IoT device should not be aware of the managing edge node as per the requirements. The edge nodes will have make the decision of which edge node is going to manage the new IoT device and it needs to be global decision amongst all the edge nodes. Hence there is a need for a global consensus mechanism. In this work, we followed a leader based consensus mechanism but it has been implemented in such a way so that it can be easily replaced with a distributed consensus mechanism like paxos or raft.

The sequence of actions for an IOT device to connect to constellation is as follows:

1. The new IoT device performs a UDP broadcast to the whole network
2. Once every time interval, the edge nodes sends the list of discovered IoT devices along with the list of IoT devices that it already manages to the leader.
3. The leader listens to all these messages from all the edge nodes and runs an IoT device to Edge assignment policy such as random, location based or load based policies once every time interval and then generates the IoT device to edge node mapping.
4. Once the leader generates the mapping, it sends a list of IoT devices to be managed to the corresponding edge nodes.

Refer to the Appendix for the detailed pictorial representation of the above steps. As the IoT device is agnostic of whether an edge node is assigned to it or not, it always sends the UDP broadcast to the network. The leader is intelligent enough to avoid conflicts or to not assign the same device to multiple edge nodes.

Constellation Execution Interface →

Execution in Constellation network has query semantics based on IoT devices. Each IoT device can have properties by which it contributes the data being generated and/or allowed actions/functionalities which can change the device's state. Constellation acts a transparent middleware running on edge devices, managing IoT devices of multiple types, architectures and from different vendors. The custom Constellation query language (CQL) is agnostic of presence of edge devices.

Device to edge communication is done using a driver program running on edge node which also exposes the properties and functionalities of the device to the network. When a new device is discovered, it is registered on the parent edge (node managing the IoT device) against a driver corresponding to its type which defines its properties and functions. Following figure 4 shows an example of an IoT device properties and functionalities/actions. All the sense entries are properties which can be used to get data from IoT devices like Temperature, Energy, etc. Actuate entries correspond to the functionalities or actions allowed on IoT device based on the data read from device. For example we turn led on/off or change color based on temperature readings from device.

```
<sense vid="Temperature" fun="getTemp"/>
<sense vid="Energy" fun="getEnergyLevel"/>
<sense vid="BigData" fun="getBigData"/>
<actuate vid="changeLED" fun="alterLED">
    <param vid="r" type="int"></param>
    <param vid="g" type="int"></param>
    <param vid="b" type="int"></param>
</actuate>
<actuate vid="ledOn" fun="activateLED"></actuate>
<actuate vid="ledOff" fun="deactivateLED"></actuate>
```

Figure 4 -- IoT Device properties and functions

Query Interface

Constellation query language is based on IoT devices and interacts with them via a query interface from client side. It currently supports following type of queries.

1. **FIND** -- This type of query is used to find all or a specific set of IoT device set. It sets a contextual alias representing the device set which can be used in next queries.
    - FIND DEVICES WITH **Temperature** AS **temps**
    - FIND DEVICE WITH **FanOn** AS **fansOn**
    - FIND DEVICE WITH **FanOff** AS **fansOff**

    Here we find device sets which has properties or actions named Temperature, FanOn, FanOff and fansOff respectively. We can sense temperature and based on the readings we can take action of switching on or off.

2. **SENSE** -- The data collection from device sets is done using SENSE statements/queries which returns the actual data read from IoT devices.
    - SENSE **Temperature** FROM **temps** PERIOD 10 SEC

    This SENSE query periodically pulls data from device set using the device set alias 'temps' contextually set by the FIND query previously.

3. **ACTUATE** -- The functionalities or actions of IoT devices are accessed using ACTUATE statements/queries. For example, after getting a periodic reading from SENSE we may decide to turn on or off fans.
    - ACTUATE **FanOn** ON **fansOn**
    - ACTUATE **FanOff** ON **fansOff**

Query Execution →

Figure 5 describes the software architecture for Constellation. Client submits query to constellation network by submitting the query to an origin node who then relays it to the whole network. Client can find an origin node to send the query in a way just as edge node discovery works. It can then select a node as origin node and send the query.

Client as two major parts -

1. **Query Compiler** - It transforms string query to Statement object to be used in software stack.
2. **Query Interpreter** - It sends the statement to origin node and stores the returned results, metadata if required as in case of FIND. Client communicates using Google ProtoBuf.

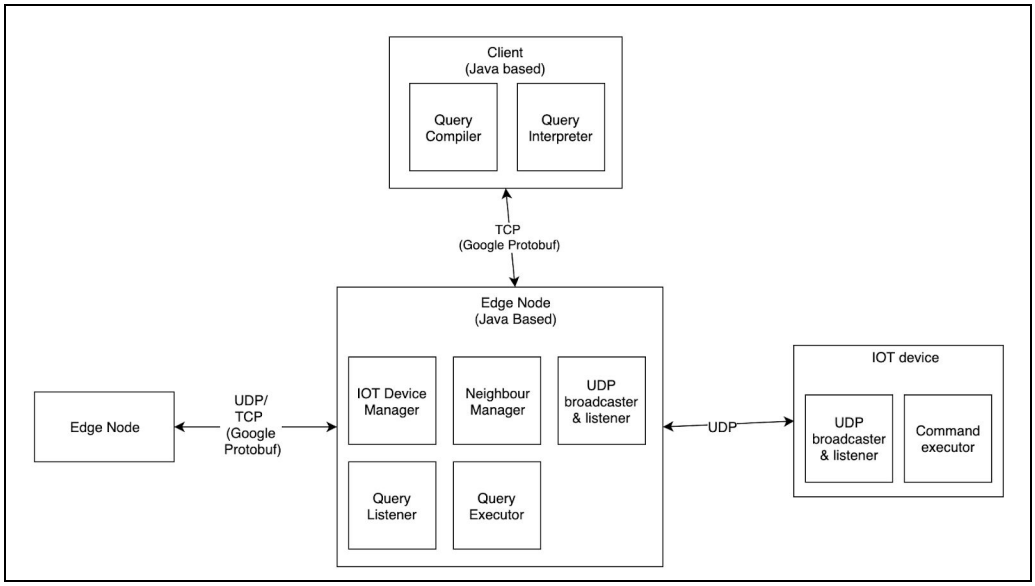


Figure 5 -- Software Architecture

IoT Device has two parts namely '***UDP broadcaster and listener***' which is used for IoT discovery and '***Command Executor***' which runs query actions received from managing edge node. Communication to an IoT device happens via UDP.

Edge Node has multiple components and it can communicate with other edge nodes via UDP, TCP and Google ProtoBuf -

1. '**UDP broadcaster and listener**' is used for edge node and IoT device discovery
2. '**IoTDevice Manager**' holds the metadata about the discovered IoT devices managed by the node and provides additional utilities over it.
3. '**Neighbour Manager**' which keeps track of current discovered neighbors of an edge node along with performing health check via heartbeats.
4. '**Query Listener**' is the entry point of all requests to the edge node. The requests are then relayed to appropriate component for execution.
5. '**Query Executor**' is responsible for running query using the IoT devices managed by the node, relaying / forwarding the query to neighbor nodes, fetching the results and sending them back.

Query Interpreter

Let's see how the Query Interpreter is used on client side. Query Interpreter is also used by Query Executor in Edge node to run queries on neighbour nodes. Essentially Query Interpreter works as shared hashmap between FIND, SENSE and ACTUATE queries. FIND query interpreter process runs and stores the results in hashmap which is then used by corresponding SENSE and ACTUATE queries. The shared hashmap has FIND alias as its key so that SENSE and ACUATE can quickly get the device set metadata. Value field is made of **Cardinality Metadata** and list of **Node Metadata**. First node in Node Metadata list is always the origin node where the corresponding SENSE / ACTuate query is sent to. We can also send queries parallel to all the nodes but all of them might not be reachable directly from client. **Node Metadata** consist of IP, port of edge node and system generated unique alias for FIND query.

Cardinality Metadata is made of -

1. Number of IoT devices found
2. Number of edge nodes having matching IoT devices
3. Number of routers (edge nodes not having matching IoT devices)
4. Height of FIND tree

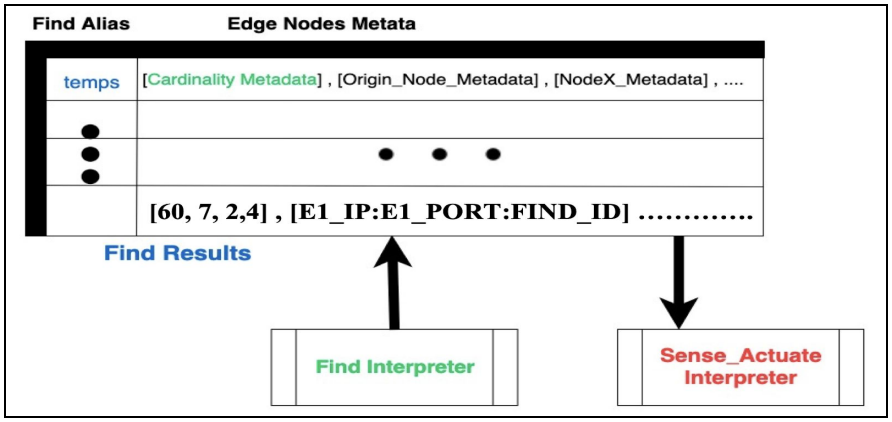


Figure 6 -- Query Interpreter

Query Deduplication

When relaying the query forward to it neighbors, the neighbour node might get a duplicate query. For example Node C in figure 7 will get duplicate queries when Node B and Node D will relay the query to Node C as part of flooding. We solve this by keeping track of all the find aliases which arrived on the edge node and using it for de-duplication and query rejection so that we do not send back duplicate data and the query runs to a completion instead getting stuck in an infinite loop.

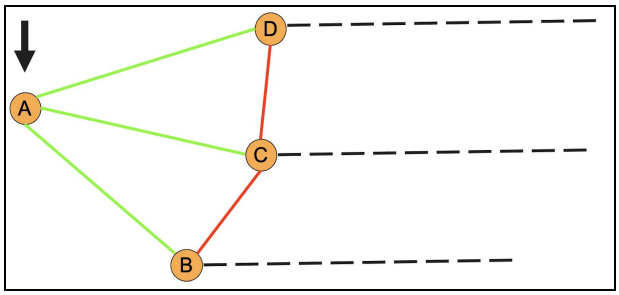


Figure 7 -- Query Deduplication

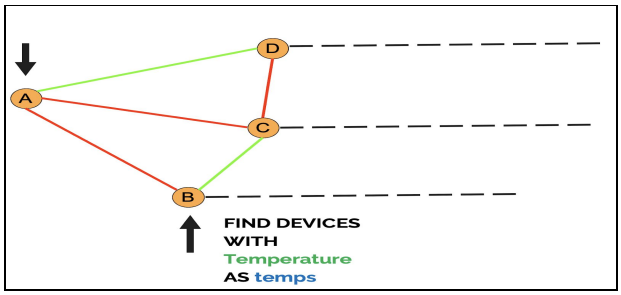


Figure 8 -- Multiple clients with same alias

Multiple Clients

Another issue arises when two clients use the same alias for same or different queries. Since query executor uses query interpreter which works on find alias as key, it is necessary to differentiate between these two aliases from different clients on a system level. If not done, it would result in a situation shown in Figure 8 where the same query comes at two origin nodes Node A and Node B. Node A relays to Node D and Node B relays to Node C which gets accepted but all other relayes gets rejected resulting in incorrect results. We solve this by generating a system wide unique alias per statement on origin node in the format ***Alias_OriginNodeInfo_UUID*** where Alias is client provided alias, Origin node info is IP and port of origin node; and UUID is a randomly generated number. This id is generated locally on origin node but is unique across the whole system. Now this new generated alias is used to send and store query metadata and results instead of original alias. The unique alias is also returned to client in case of FIND and is stored in the value part of hashmap as described previously.

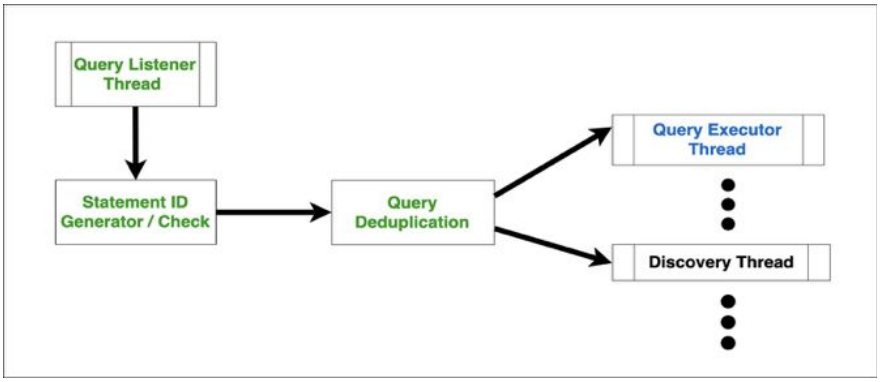


Figure 9 -- Query Deduplication and Unique ID generation

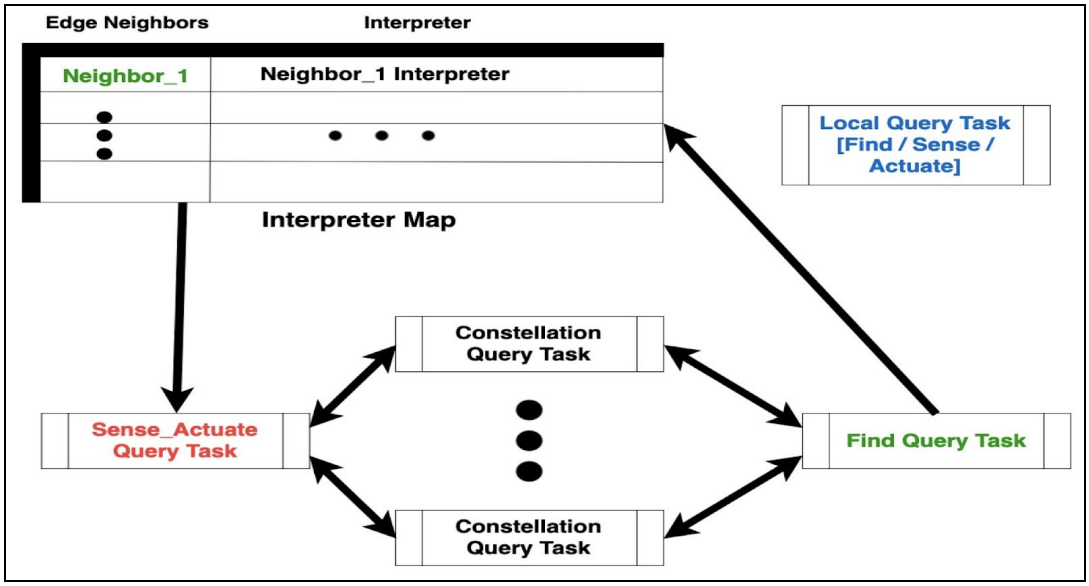


Figure 10 -- Query Executor

Query Executor

Query Executor uses a two level data structure to run queries. It keeps an interpreter hashmap as shown in figure 10 where key is neighbour edge node info and value is a query interpreter previously described in figure 6.

When a FIND query comes to an edge node, it starts a FIND Query Task. FIND Query Task then starts a local FIND task to run query using IoT devices managed by the node. It also starts a Constellation Query Task, one of each neighbour to relay the query to. Once all the tasks are completed, the metadata collected from Constellation tasks is stored in respective interpreter of the neighbour and the combined data from all the tasks are returned back. This is how we make the logical FIND tree.

When a SENSE / ACUATE query comes in, it creates a local task just as FIND query. It then uses interpreter hashmap data to check out which all neighbors have seen the corresponding FIND query and forward the SENSE / ACTUATE query only to those nodes using Constellation Query Task. It then waits for completion and sends back the combined data.

Fault Tolerance →

Constellation network handles edge node or IoT device failures using the health check mechanisms.

Handling Edge Node failures

The edge nodes continuously exchange TCP health check messages between its neighbors once every time interval. If any of the edge node is not reachable during the health checks, it removed from the set of neighbors and queries are stopped being routed to that node. After a neighbor failure, incase if needed, the edge node can run the discovery process to find new neighbors.

Handling IoT Device failures

The UDP broadcasts from IoT device is used as health checks. The edge node will interpret the UDP broadcasts as health checks if the IoT device is already assigned to it or it will just treat it as regular new device join request. If the edge node doesn't receive the UDP broadcast for more than 2 cycles of time intervals, then the device is removed from the set of managed devices.

The above mechanism is useful even in the case of moving edge nodes or IoT devices in addition to the failure scenarios. In both cases, failures/movement leads to incorrect/stale routing information stored for each query on each of the edge nodes. In order for the query routing to work properly, the query state needs to be fixed. It can be done in two ways as follows :-

1. **Method 1 - Re-running FIND query :** Re-run the FIND queries of all the affected queries. It helps identify the new devices, edge nodes or any changes in the constellation network in addition to fixing the query state. But it might dramatically increase the load on the network depending on the number of affected queries.

2. **Method 2 - Locally fixing the query state :** Update the FIND trees of the affected queries with the new information recursively upstream starting from the affected edge node. It will generate very less number of messages compared to the method-1 above but it will not discover any new devices or edge nodes recently added in the network. It will just fix the existing state.

Network Reconfiguration →

The discovery mechanisms discussed above have no knowledge on the querying patterns while forming the network. But what if the network created is not optimal for the queries getting generated. How to reconfigure the network structure in order to optimize the queries without global state of the whole network? This problem can be addressed in a lot of interesting ways which is a research project on its own. In this work, we implemented a systems based approach to identify and perform the network reconfiguration. Each and every edge node monitors the number of the communications between each of its neighbors. If it finds the communication between a pair of neighbor exceeding the threshold, it will suggest them to connect to each other directly as neighbors. Along with the suggestion, it will also send a set of queries that can be routed directly. This reconfiguration will not only improve the performance of the re-routed queries but also improve the runtime of future queries as we assume that the past query patterns are the best indicator of the future queries.

Experiments & Evaluations →

Setup: All the experiments were performed on a simulation platform running on a single machine. Each IoT device runs on a seperate single thread for concurrency. Each edge node is represented with seperate set of threads, one for each parallel component in edge device software stack. Both IoT devices and edge nodes have geo-locations associated with them, which is used for discovery, configuration, execution and re-configuration, etc. To make it more realistic, the geo-locations of edge devices correspond to locations of 50 Halls and buildings from UMN-TC east bank campus. The IoT devices are randomly generated over the span of campus buildings. The following figure 11 shows a simulated setting where red dots shows edge devices in halls and black dots corresponds to IoT devices.

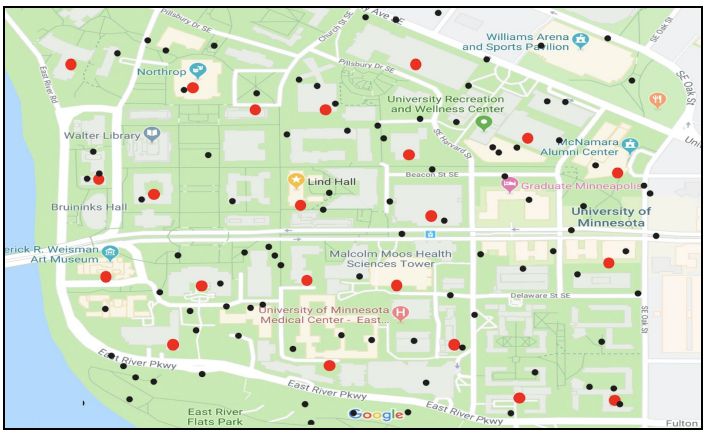


Figure 11 -- Edge Nodes and IoT devices

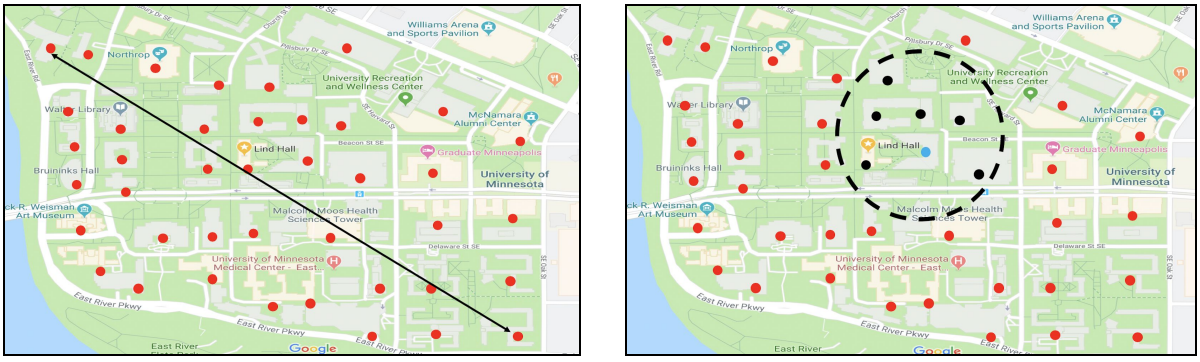


Figure 12 -- Reachability

Another concept which is central to simulation platform, experiments and configuration of network is Reachability. **Reachability** of 'r %' of an edge node or IoT device is defined as the number of edge nodes whose geo-location lies within 'r % of maximum span edge node distance from the geo-location of said edge node or IoT device. For example left part Figure 12 shows the maximum span distance between edge nodes and right part shows 25% reachability (black nodes) of an edge node (blue node). To remove any bias and outliers in execution, we run experiments multiple times with random settings and take an average of all readings. Following figure 13 shows the Simulation Platform testing logic.

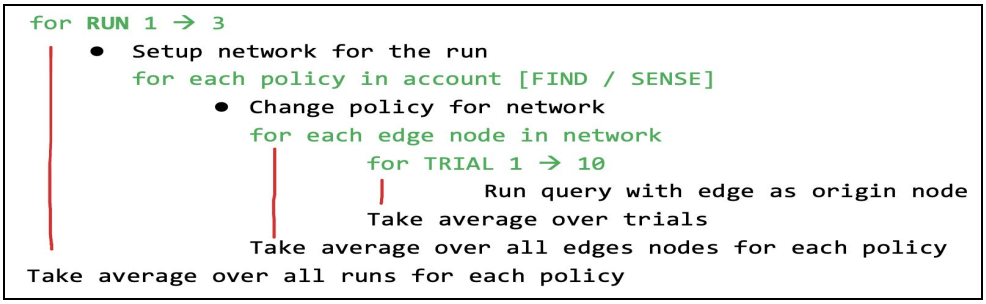


Figure 13 -- Simulation Platform Testing

- ***Discovery and Configuration***

➢ Time taken to form the constellation:

Varying max neighbors: Figure 14 shows the time taken to form the whole constellation network given a set of new edge nodes joining at the same time. The x-axis indicates the number of new edge nodes that want to join the network and y-axis indicates the time taken in milliseconds for all the nodes to join the constellation network. The three different lines indicates the time taken to form the network by varying the maximum number of neighbors threshold each edge node can accept. From the graph, we can infer that the time taken to form the network increases as the number of edge nodes increases. Also, we can see that for a particular set of nodes say 30, the time taken decreases as the number of neighbors increases. This is because when the maximum number of neighbors are less, the bootstrap nodes will max out on the neighbors count and the edge nodes will have to wait for the nodes that joined newly to the network to respond to their discovery requests. So as the maximum neighbors count increases, there will be less wait time to join the network. These simulations will be helpful in determining the neighbors count according the time requirements.

Varying edge node reachability: Figure 15 shows a similar experiment as above. Here the bars indicate the time taken to form the network by varying the reachability of the edge nodes. From the graph, we can infer that as the reachability between the nodes decreases, the time taken to form the network increases. This might be because the nodes will have to wait for its reachable nodes to be part of the constellation. But it also shows an interesting pattern, when the number of nodes are increased, varying/decreasing the reachability doesn't change the time taken. This is because, we increase the number of edge nodes with the same geographical area. So there will be a lot of nodes that are reachable which might be part of the constellation - leading to lower wait time to join the network.

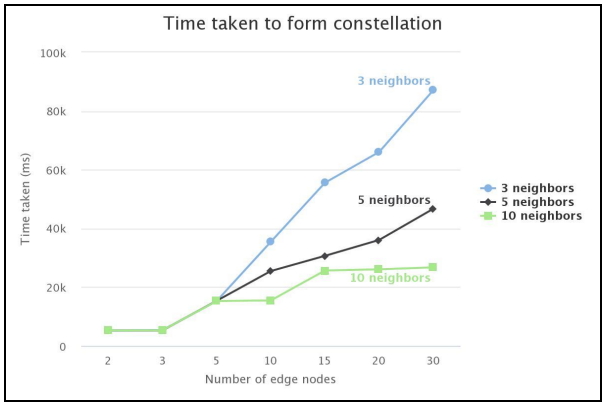


Figure 14 -- Time to build VS edge nodes

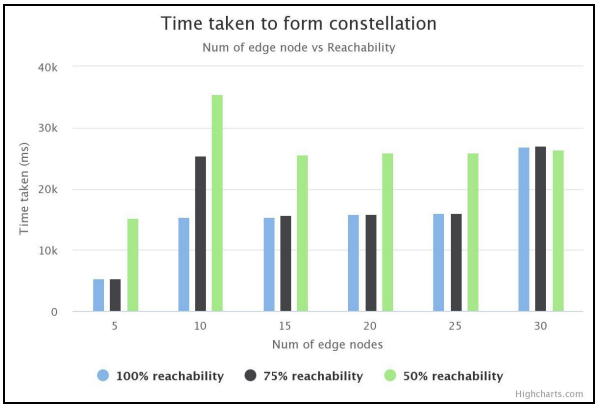


Figure 15 -- Time to build VS reachability

➢ IoT Device Discovery

IoT device discovery decides which edge node is actually going to manage the new IoT device. There are 3 different policies which are as follows:

1. **Random Policy:** The IoT devices randomly assigned to one of the edge nodes who discovered it.
2. **Distributing Device Types:** The IoT devices are distributed according to their device types across all the edge nodes who discovered it. This will result in each edge node having at-least one of all the different device types.
3. **Clustering Device Types:** The IoT devices are clustered in a same edge node based on their device type. This will result in all the devices of the same type are managed by the same edge node.

Figure16 gives a pictorial representation of the above three policies.

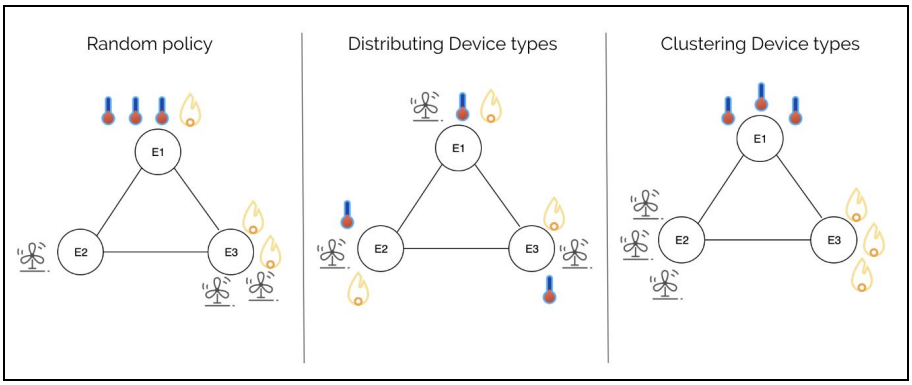


Figure 16 -- IoT Device Discovery Policies

The experiment setup involved 5 edge nodes, 25 IoT devices and 5 different device types (5 devices of each type). The time taken to SENSE the AIR type sensor values under different policies was measured which basically indicates the TCP connection cost and number of parallel open connections during the query. The x axis indicates how many sensor values are requested in the SENSE query and y-axis indicates the time taken to finish executing the query. From Figure 17, we can infer that the time taken to get 1 sensor value is the same across all the policies. But if we increase the number of sensor values requested, the clustering policy always yields a similar values and is less as compared to the other two policies.

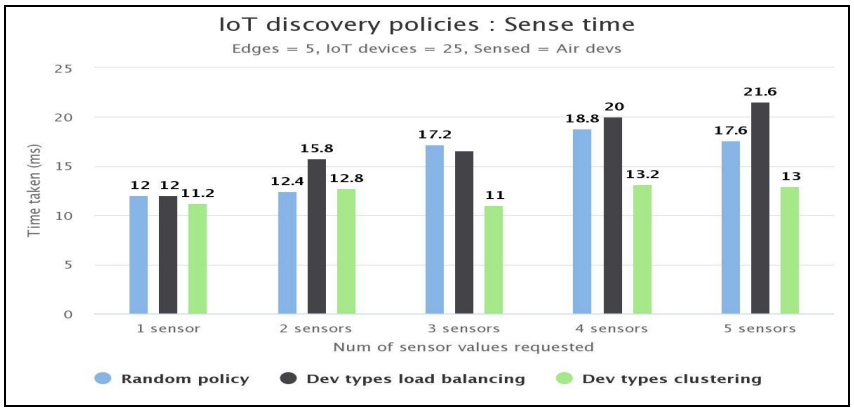


Figure 17 -- Time to SENSE vs IoT Device Discovery Policies

➢ Performance comparison between failure handling methods

The experimental setup involved 12 edge nodes and 12 IoT devices. And the 12th node is failed, which is the last node in the FIND tree for the queries ran. The x-axis indicates the number of queries affected in the failure and y-axis indicates the time taken to fix the query routing state of those affected queries. The bars indicates the different methods in fixing the state. From Figure 18 , we can infer that the time taken to fix the state by using the reconfig is always lesser than FIND re-run. The time taken indicates the number of TCP messages sent and the number of parallel open socket connections during the state fixing. The larger time taken for re-running the FIND query is cause of the whole network flooding. Running multiple FIND queries at the same time will dramatically increased the load of the network.

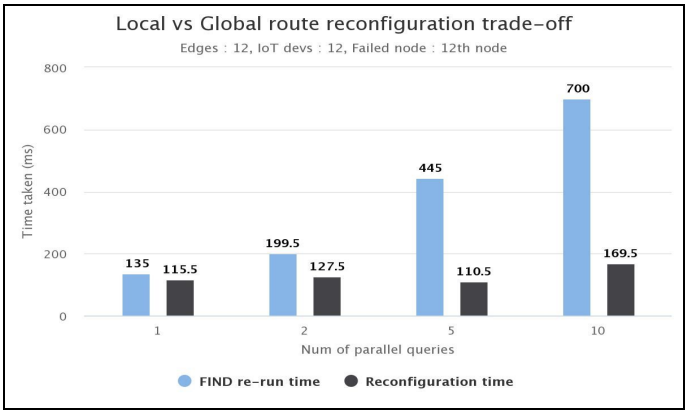


Figure 18 -- Time to reconfigure vs re-run FIND

➢ Network re-configuration

The experiment setup involved a simple constellation network consisting of 12 edge nodes and 5 IoT devices. The re-configuration is manually triggered to test the query performance before and after reconfiguration. Figure 19 indicates the time taken for a SENSE query before and after re-configuration along with the time taken for the re-configuration. From the graph, we can infer that the time taken to complete the query is decreasing after each re-configuration. It may seem that the time taken to perform the reconfiguration is larger than the efficiency gained but our argument is that this re-configuration will make this query faster in the future periodic runs and also all the new queries in the future as they are expected to have the similar pattern.

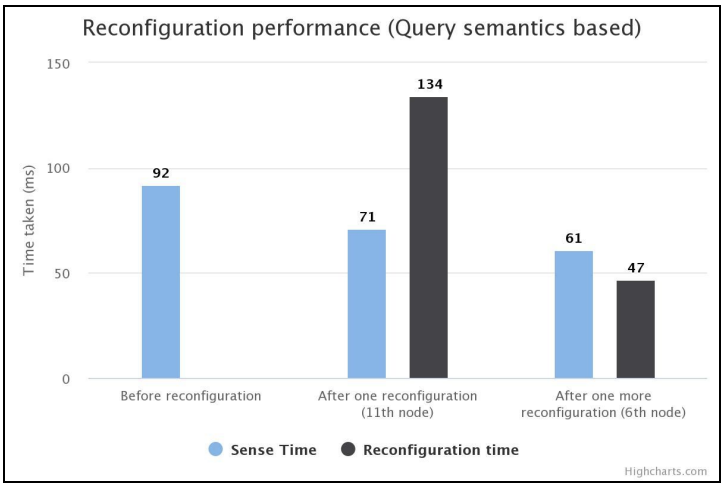


Figure 19 -- Query Semantics based reconfiguration

• *Query and Execution*

Before we discuss different policies for executing queries, let us understand the topology and network configurations on which these experiments were run. The network configuration depend on Edge Node discovery and IoT discovery. For Edge Nodes, '*Nearest Neighbour Selection Policy*' was used with varying reachability. A reachability of 25% is considered as SmartCampus and reachability of 100% is considered as SmartLab setting. For IoT Device discovery, NodeBalanced and TypeClustering policies were used (figure 20). For our demonstration purpose in this document, let us assume the P2P network shown in figure 21 is resultant of EdgeNode Discovery mechanism and query arrives at node marked with arrow.

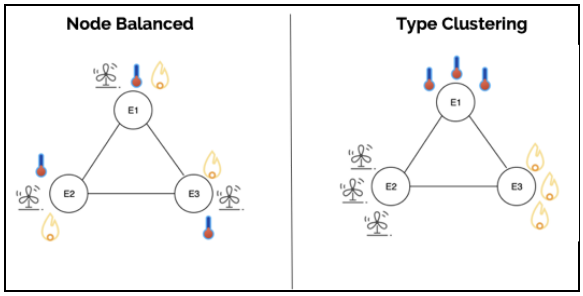


Figure 20 -- IoT Device Discovery Policies

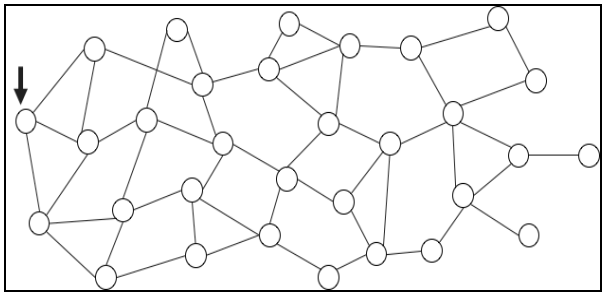


Figure 21 -- Sample Constellation Network

➢ FIND

There are multiple ways to execute a FIND query in a P2P network. In this work we present major policies which show potential to serve majority use-cases and give good quantitative results.

1. ***Full Flooding Policy***

   FullFlodding Policy usage will forward the query to all its neighbors (including potentially the sender). If a node has already seen a query, it will reject that request, and the forward chain in that branch will stop. Since there can be many cycles in a P2P network as shown above in figure 21, a node might receive the same request multiple times from different routes. When the algorithm ends, all the nodes will be visited and the graph will be converted to a FIND tree as shown in figure 22 below. Since we flood to all the neighbors, the collision overhead of query rejection is very high. Especially, if the reachability is 100%, each node will have more shared neighbors and hence contention becomes even higher.

2. ***K- Hop Flooding Policy***

   K-Hop Flooding Policy will only forward the query to a certain depth K. Hence it is used to find devices in the vicinity. Of course, when the reachability is is higher, each node will have more neighbors and most of nodes will be reachable a low K-Hop value. All nodes will be available in a finite hop distance. This will also create a small FIND tree as shown in Figure 23 below.

3. ***Random Walk Policy***

Random Walk Policy, at each node first select a random '*r*' number of neighbors between 1 and all and then randomly select '*r*' neighbors from all neighbors to forward the query. This reduces the contention caused by query collision while still allowing the remaining neighbors to be visited via another path. Hence, this policy searches and samples in a global geo, but can still miss out some nodes as shown in Figure 24 below.

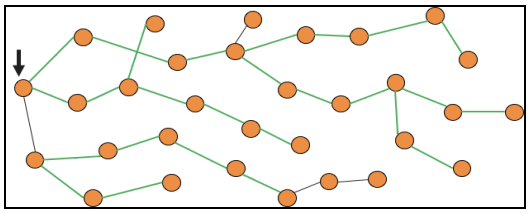


Figure 22 -- Full Flooding Policy

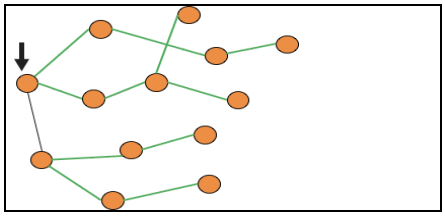


Figure 23 -- K-Hop Flooding Policy

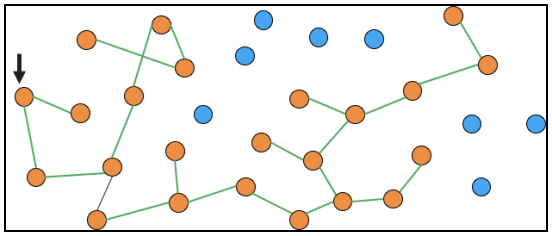


Figure 24 -- Random Walk Flooding Policy

➢ FIND Policies Evaluation

Now we will discuss these policies in a quantitative and qualitative manner along with use-cases in which they fit in. While using K-Hop, we use a percentage of edge nodes as value of K.

Figure 25 shows time to find 100 IoT devices and Figure 26 shows percentage of IoT devices for varying number of edge nodes and 25% availability. We see that FullFlooding policy takes 5.5 times more time than K-Hop 16% and RandomWalk policy. On the other hand FullFlooding policy always find all the devices and while K-Hop 16% and RandomWalk policies find a lower percentage.

- If we need to find all devices, we have to use FullFlooding policy.
- If we need to find global sampling of edge nodes in less time, use RandomWalk policy.
- If we need to find local devices, use K-Hop which is also faster.

In Figure 26, the reason for increasing IoT device discover percentage for increasing K-Value of trivial. On the other hand RandomWalk policy has a parabolic shape. This is because as you increase the number of nodes in a region, the more nodes might be left out. But as the number of edge nodes goes over a threshold, the density of edge nodes increases and each node have more neighbors, allowing the left-out nodes to be discovered via additional alternate routes. In this case the deflection is seen at 25 edge nodes.

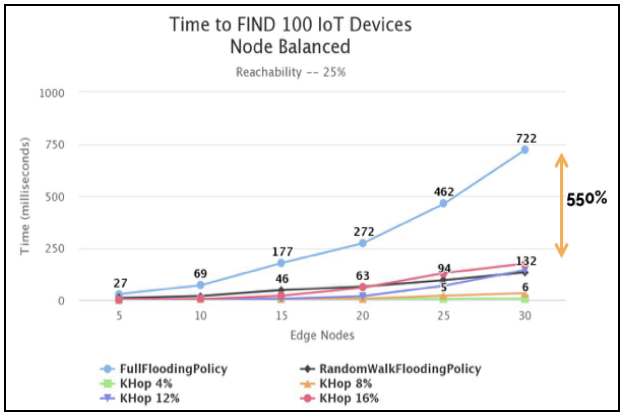


Figure 25

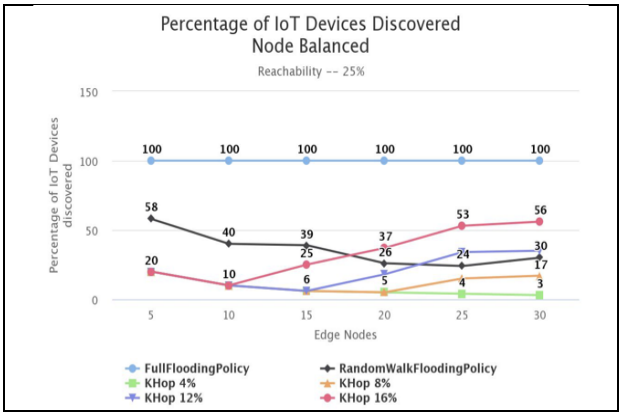


Figure 26

Figure 27 shows time to find 100 IoT devices and Figure 28 shows percentage of IoT devices for varying number of edge nodes and 100% availability. We see that FullFlooding policy takes 34% more time than K-Hop 16% and RandomWalk policy. But not we see that K-Hop 16% and RandomWalk finds almost the same percentage of IoT devices as FullFlooding.

- If we can work with approximate to all, we should never use FullFlooding policy but instead use K-Hop or RandomWalk policy.

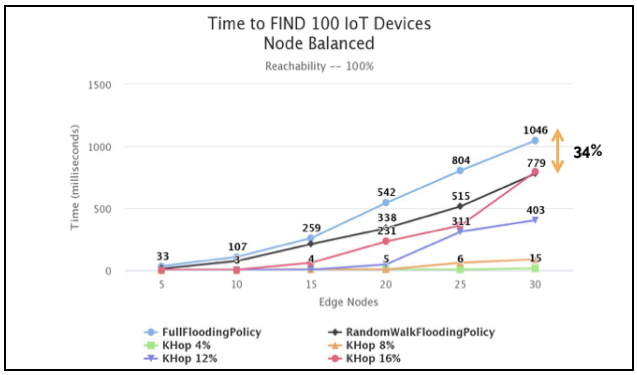


Figure 27

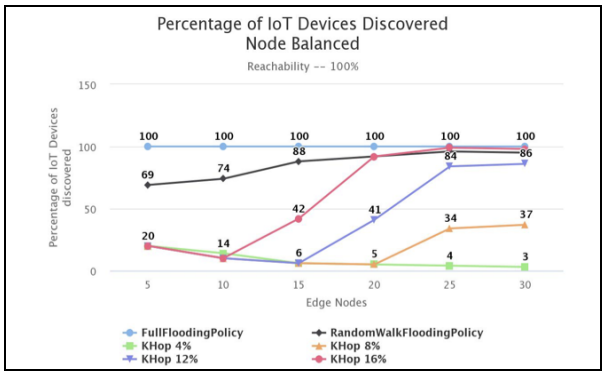


Figure 28

In Figure 28, there is no parabolic deflection in RandomWalk. In 100% reachability, all the nodes always will have almost all other nodes as neighbors, always allowing more back paths to find left-out nodes. K-Hop 16% work almost as FullFlooding because every node is within a finite hop distance which is 16% for this case.

Figure 29 shows Time to find IoT devices and Figure 30 show percentage of devices found with 20 edge network in 25% and 100% availability. We see that as the number of IoT device increases in the network, the FIND time and discover percentage remained almost constant. It shows FIND is only a function of edge nodes and not IoT devices. We also see in Figure 29 that for reachability of 100%, K-Hop 16% has almost the same time as Full Flooding but RandomWalk still has lower time. This shows K-Hop is more susceptible to network configurations than RandomWalk Policy.

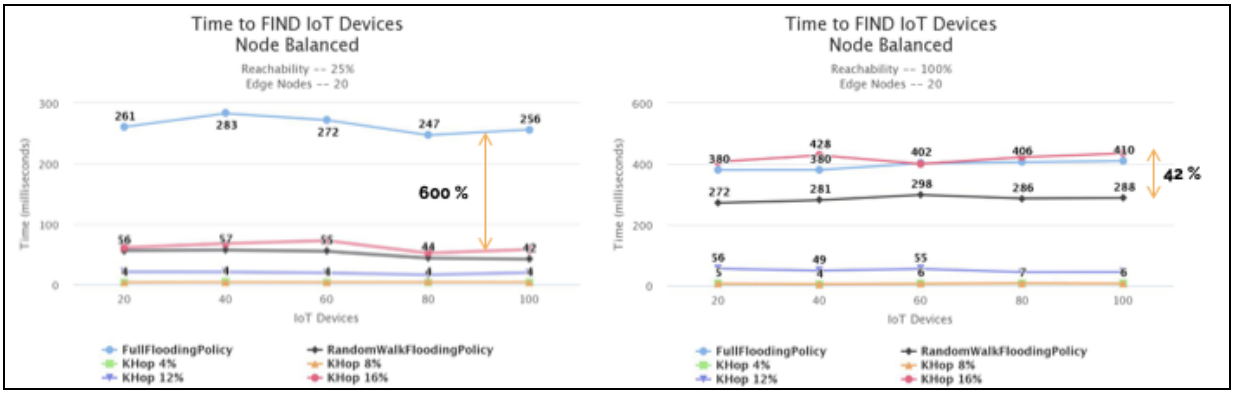


Figure 29

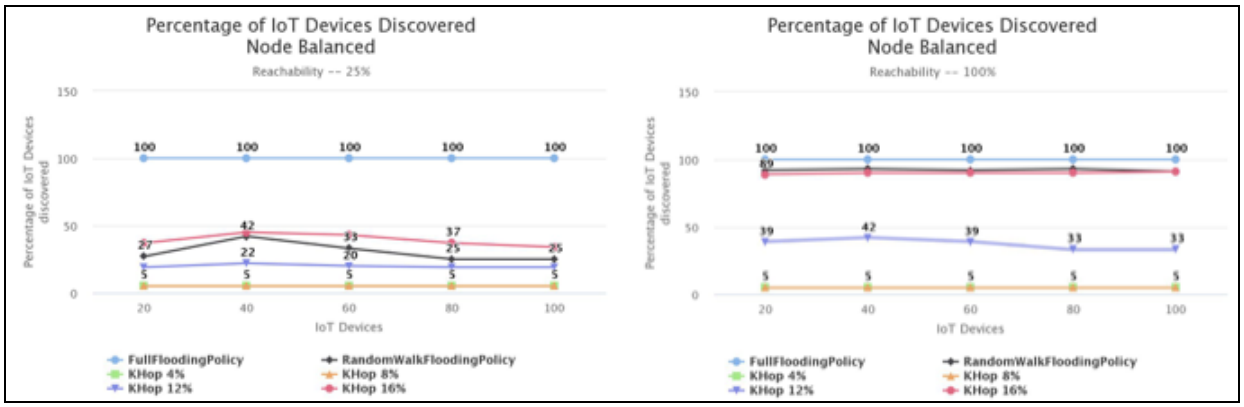


Figure 30

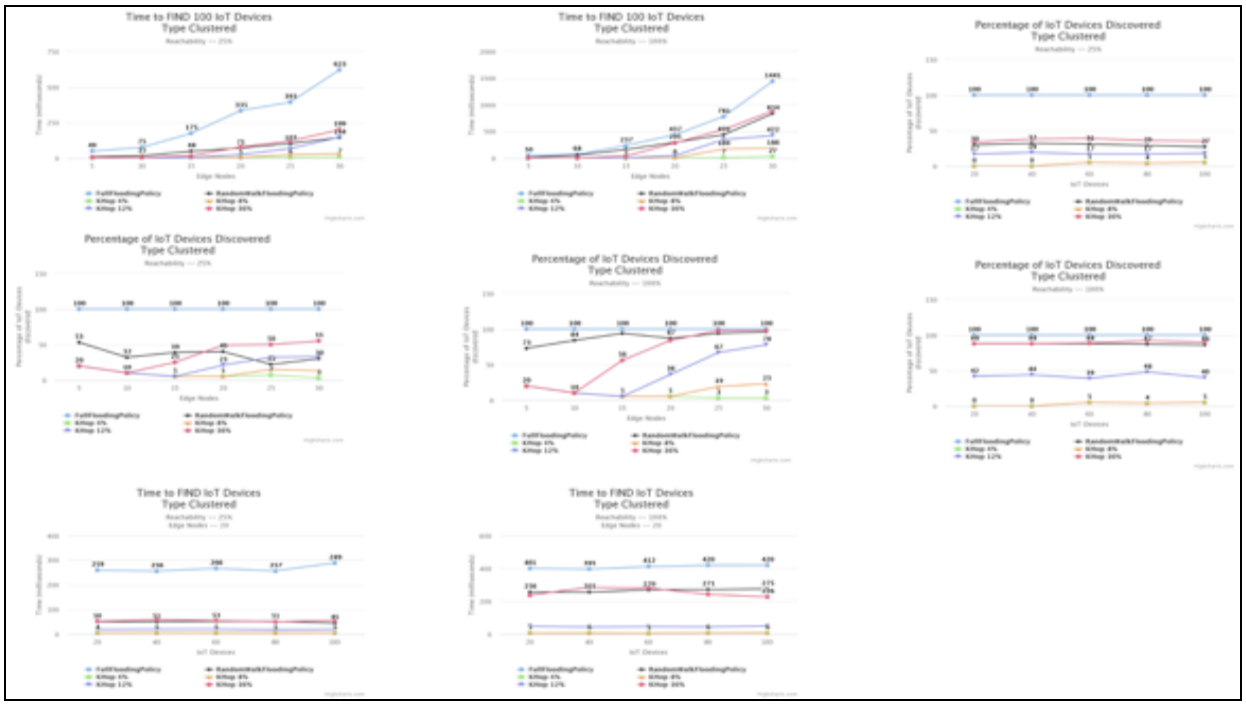


Figure 31

Figure 31 shows all the above FIND evaluations but with 'Node Clustered' as IoT discovery policy. The outcomes are the same as Node Balanced Policy and hence not discussed again.

➢ SENSE

After we FIND the devices, we can query the device set for SENSE or ACTUATE. We might not use whole device set for these queries. For example, FIND query may return 100 IoT devices but we may to get data from a subset of devices or devices possibly with additional properties (for example geo-sampling or geo-constrained). Also, there are additional factors affecting efficient routing of these SENSE / ACTUATE queries like FIND Tree depth, total number of nodes, amount of data, etc. To efficiently run queries, we create a soft state at each node of FIND tree. For each child / branch of a node in FIND tree we store in parent -

1. **Cardinality** - Number of IoT devices found in the branch
2. **Nodes** - Total number of edge nodes with matching IoT devices in the branch
3. **Routers** - Total number of edge nodes which do not have any matching IoT devices in the branch
4. **Height** - Height of the branch

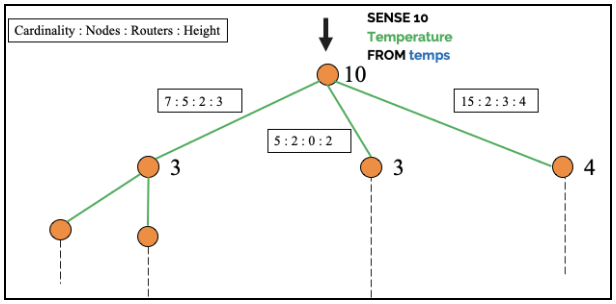


Figure 32 -- Uniform Routing Policy

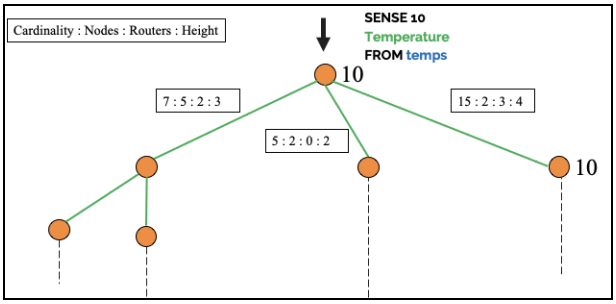


Figure 33 -- Minimal Subtree Routing Policy

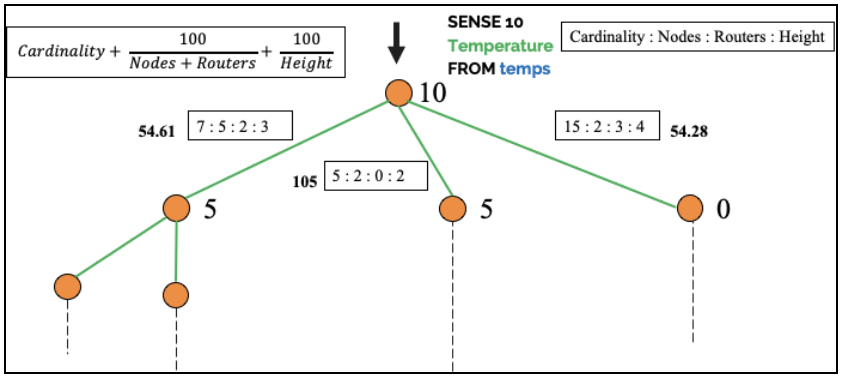


Figure 34 -- Metadata Routing Policy

Below are the three routing policies which use the above described metadata for running SENSE queries given a FIND TREE.

1. ***Uniform Routing Policy*** - The required cardinality at node is equally partitioned between all branches. As shown in Figure 32, required cardinality of 10 is equally routed to branches as 3, 3 & 4.

2. ***Minimal Subtree Routing Policy*** - The required cardinality at node is equally partitioned such that minimal number of branches have to be used to for routing. As shown in Figure 33 required cardinality of 10 can be satisfied by just one branch.

3. ***Metadata Routing Policy*** - Each branch is scored by a function and the branches are used to full capacity in order of decreasing score. The scoring function used is →

$$Cardinality + (100 \div (Nodes + Routers)) + (100 \div Height)$$

The rationale is -- more the cardinality less the branches to use; less the height and number of nodes faster the execution will complete. This is shown in Figure 34.

➢ SENSE Policies Evaluation

Now we will discuss these policies in a quantitative and qualitative manner along with use-cases in which they fit in. In these experiments, FullFlodding Policy was used to create FIND Tree. The queries in these experiments are ad-hoc queries and IoT devices can emit two types of data - ***small*** which is a double and ***large*** which is a string of 1 Mb.

Figure 35 shows reading small data from 30 - 80 IoT devices for NodeBalanced IoT discovery and a 25% availability. We see UniformRouting policy takes the most time, followed by MetadataRouting policy and MinimalSubtree policy performing the best. The reason for this is when the data is small, there is more connection overhead than data transfer cost. UniformRouting policy has the largest parallel connection overhead while MinimalSubtree has the least. Similarly for 100% reachability in figure 36 we see the same results. Though we see a deflection when SENSE cardinality becomes 80. This is because at 100% reachability, height of all subtrees will be equal and smaller than in 25% case and the transfer overhead start to overcome connection overhead.

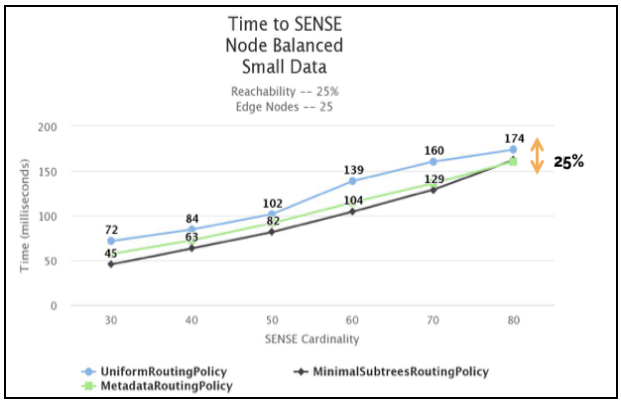


Figure 35

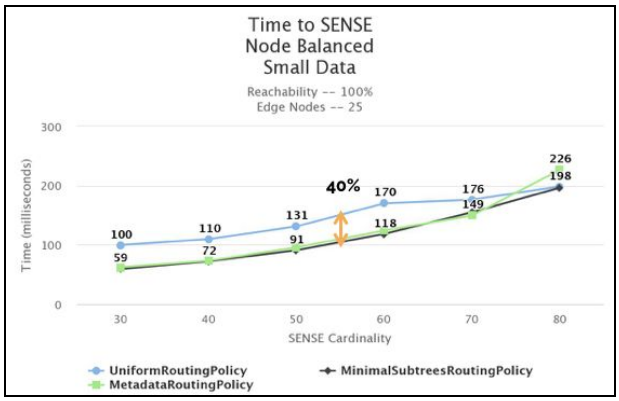


Figure 36

We see this same behaviour in figure 37 , figure 38 when IoT device sends large chunks of data and transfer over is always larger than connection overhead and hence results are flipped with MinimalSubtree policy performing the worst and UniformRouting policy performing the best.

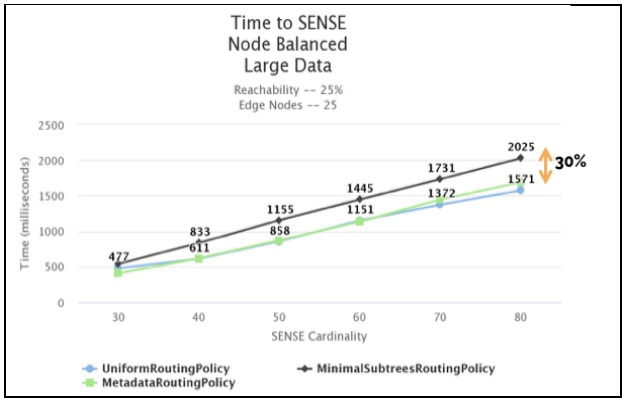


Figure 37

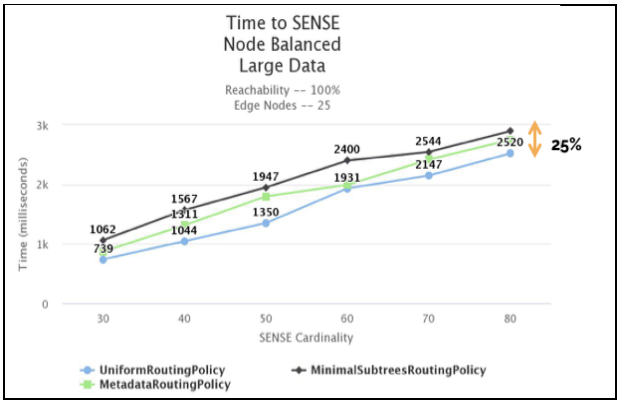


Figure 38

Figure 39 shows another deflection case, where initially the graphs are as expected but later all policies perform just the same. The reason for this is that FIND tree in Clustered IoT discovery is very sparse and the deflection seen in figure 36 comes in quite early in this case. Figure 40 and figure 41 shows other cases for 'Type Clustered' IoT device discovery and perform same as Type/Node Balanced' IoT discovery and need not discussed again.

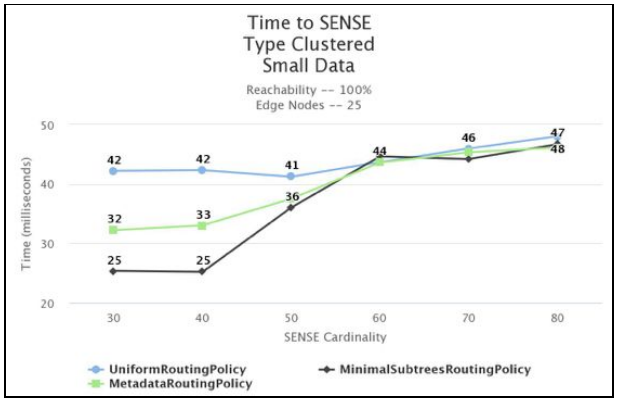


Figure 39

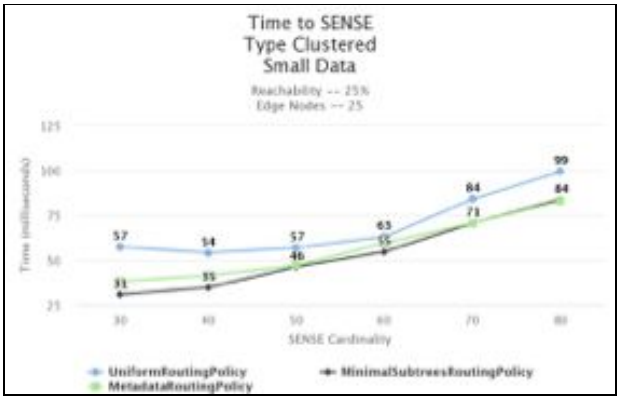


Figure 40

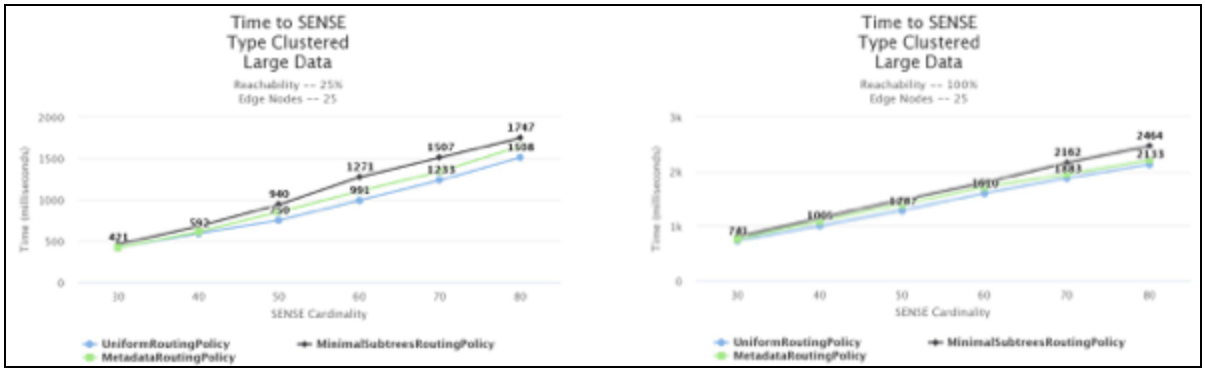


Figure 41

GeoBased Queries →

Queries in an IoT system may also want to use IoT devices in a specific geo-region. But the client can be anywhere and the query can originate at any edge node. We can think of using geo location in both FIND and SENSE but upon further deliberation, we decided to only support in FIND queries with additional parameters LOCATION [**Center Geo Location**] [**Radius**]

FIND DEVICES
WITH Temperature AS TEMPS
LOCATION ***44.974359 -93.229468 400.00***

Using geolocation in SENSE is not beneficial because FIND tree by itself does not store geolocation metadata. Even if do store it while creating a FIND tree, we would still end up traversing almost the full tree to ensure we do not miss a branch which might somehow finally end up going into the region asked for. By the above discussion, it is clear that none of the previously discussed FIND policies will not work here. K-Hop might not even reach geo-region, FullFlooding and RandomWalk will create the case where we have to traverse the full FIND tree for SENSE. Hence, we need a more intelligent policy for geo based queries. A more efficient approach will be route the FIND query to the required geo region and then use FullFlooding policy cutting off when a neighbour goes out of the region.

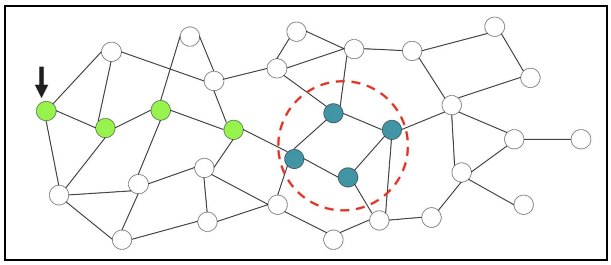


Figure 42 -- Geo Based Routing Policy

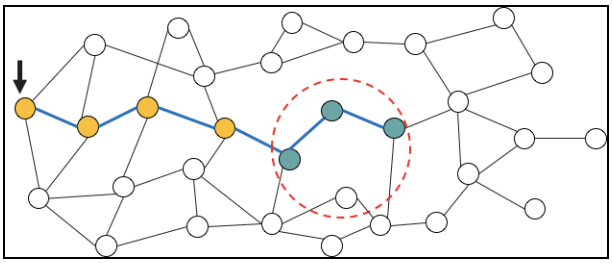


Figure 43 -- Missing edge node in geo-region

This is shown in figure 42 where the FIND query is first routed and then flooded. However this might not cover all the bases as we see in figure 43 where a node is lost because it can not be reached using FullFlooding within the region. So we need to allow some leeway for search to come back in the region. We propose to use K-Hop flooding policy to allow search come back in geo region. Figure 44 shows this mechanism with K = 2.

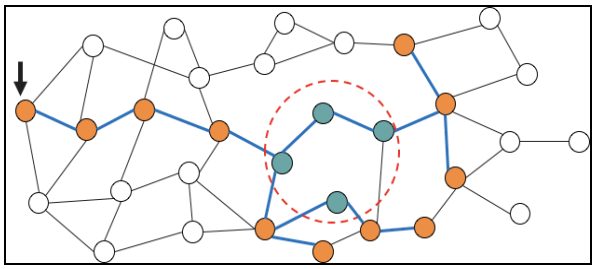


Figure 44 -- Geo Based Routing with K-Hop

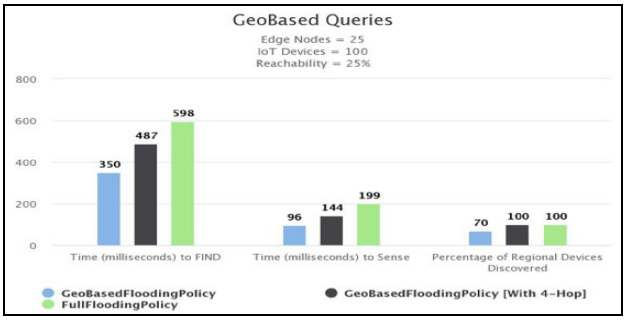


Figure 45 -- Time to FIND and SENSE

Figure 45 shows all of these comparisons in one holistic view. The network is created with 25 edge nodes, 100 IoT devices and reachability of 25%. As we see FullFlooding policy takes maximum time to both FIND and SENSE, followed by K-Hop based GeoBased Flooding policy and finally vanilla GeoBased Flooding. But we do see that with K = 4 are able to get 100% results at much lower run time and vanilla GeoBased flooding policy do miss out some IoT devices.

The routing mechanism for GeoBased policy currently is to forward the query to neighbour which is next nearest to geo-region center. However, this also might not work in all cases as shown in figure 46, we see that there is no neighbour which is nearer to the center that the origin node. Even if we do send to one of the upper nodes, the next time they will send to origin node and the query will be rejected and incorrectly completed. Instead we can use a view frustum approach to route the query to all the nodes in view frustum (based on geo location) so that we can reach the region in most of the cases. This is still a work in progress and needs more deliberation and is shown in figure 47.

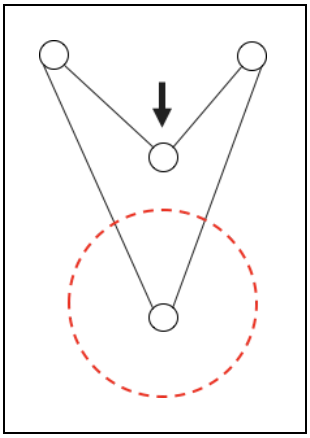


Figure 46

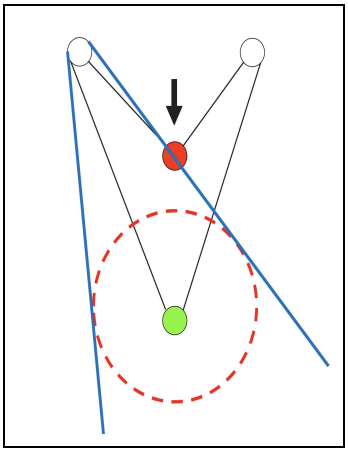
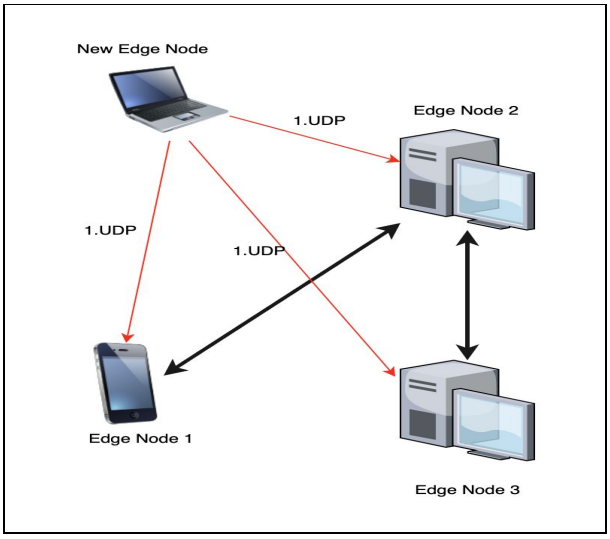


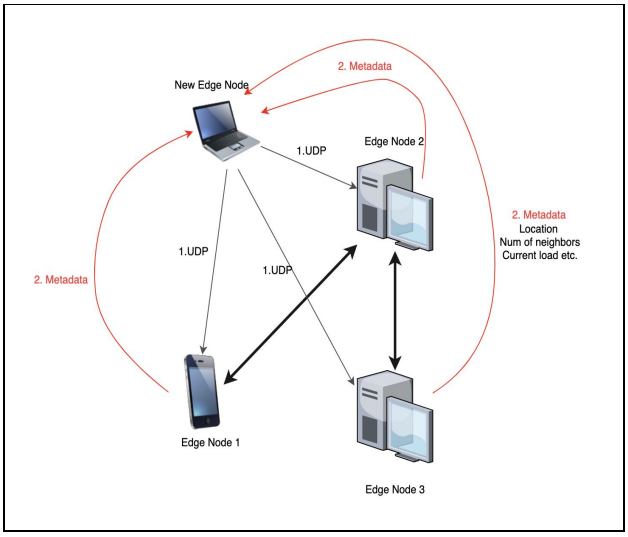
Figure 47 -- View Frustum

Appendix

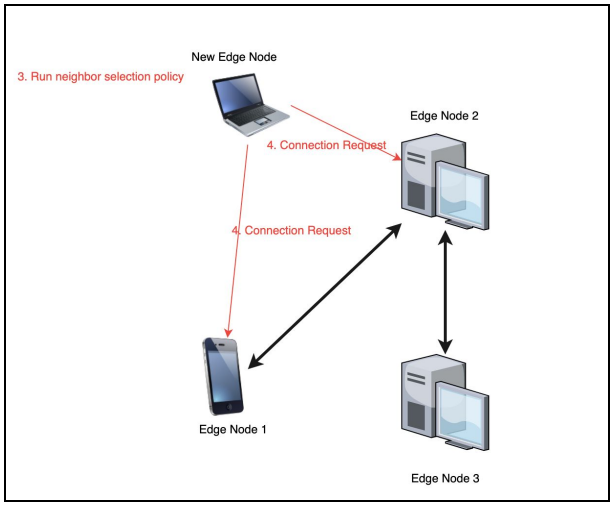
Steps in Edge Discovery:



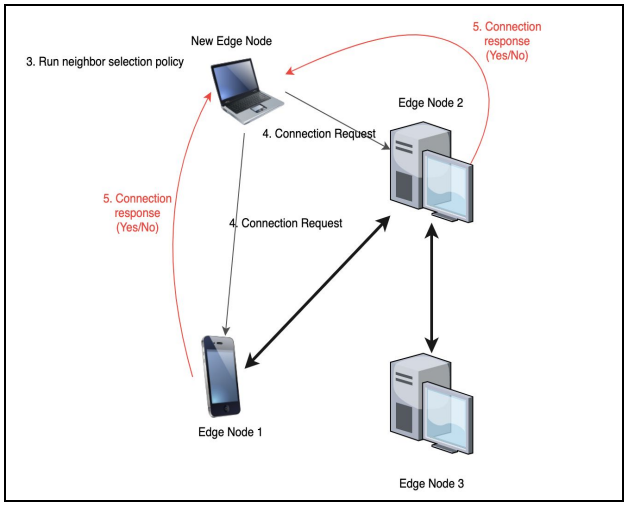
(1)



(2)

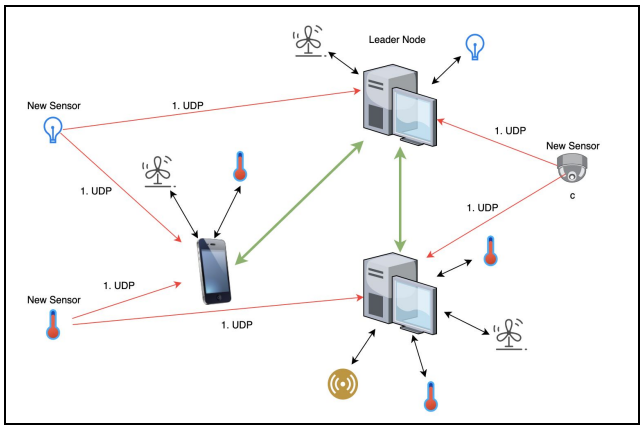


(3)

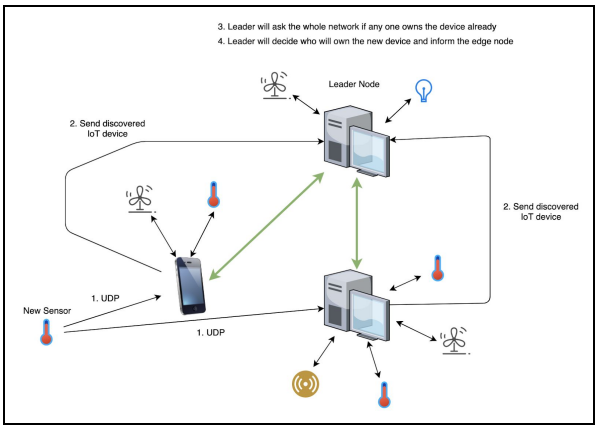


(4)

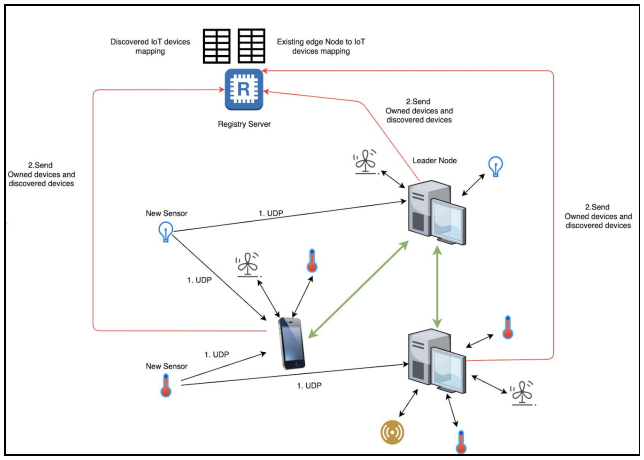
Steps in IoT discovery:



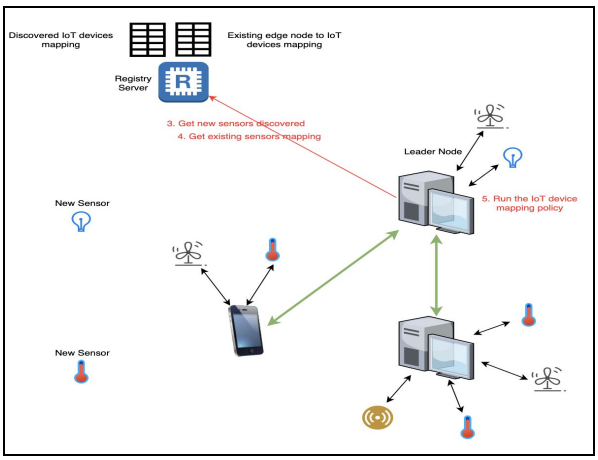
(1)



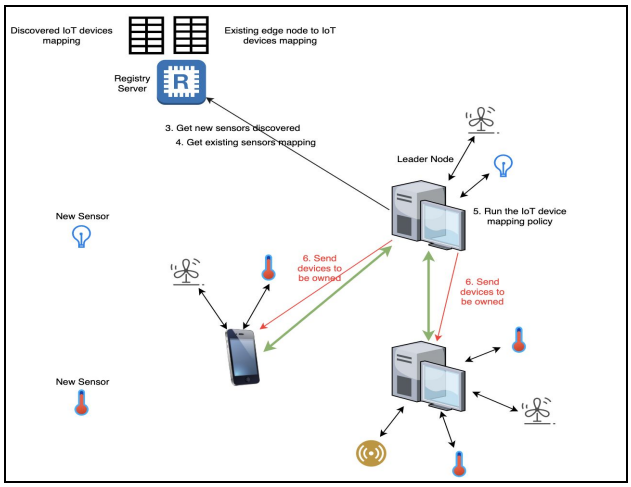
(2)



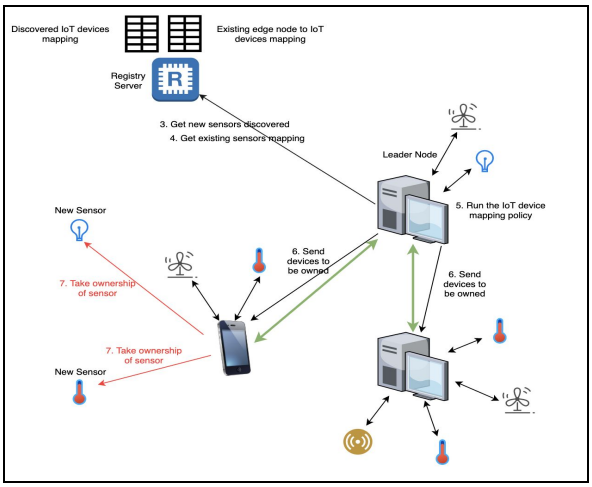
(3)



(4)



(5)



(6)