# Design, Development and Deployment of an Information-Centric Networking based solution for the Internet-of-Things

Project CS 2016


## Authors:

Andrew Aziz
Andrew.Aziz.2356@student.uu.se

Ludwing Franquiz
Ludwing.Franquiz.2071@student.uu.se

Max Wijnbladh
Max.Wijnbladh.2971@student.uu.se

Eric Wang(Yiqing)
Yiqing.Wang.6263@student.uu.se

Maria Rajabzadeh Namaghi
Maria.Rajabzadehnamaghi.5780@student.uu.se

Theodosios Malatestas
Theodosios.Malatestas.1471@student.uu.se

Johan Snider
Johan.Snider.3503@student.uu.se

Sami Kaivonen
Sami.Kaivonen.4995@student.uu.se

Adrian Amigues
Adrian.Amigues.7647@student.uu.se

Eirini Petraki
Eirini.Petraki.5059@student.uu.se

Uppsala University


January 12, 2017

**Abstract**

Information Centric Networking (ICN) is based on the idea that information is independent from its location and directly addressable and routed based on a name. Any node within the network can request and receive the data from the nearest node that has the content in its cache. This information-oriented internet structure is well-suited for a large scale Internet-of-Thing application.

In this project we have deployed and built a wireless sensor network using Content Centric Networking (CCN) relay services and developed an Android application for users to be able to request the sensor's data. Additionally, we also setup a datastore that can store the information. This architecture has advantages and disadvantages that we analyze in this report.

# Table of contents

# 1 Introduction

Information-Centric Networking (ICN) is a networking paradigm which promotes a data-centric network perspective over a traditional host-centric IP network perspective[1]. Instead of basing communication on named hosts, ICN bases communication on named data. Centering communication around data allows that data to become independent from its location, application, storage and means of transportation [1].

Internet-of-things (IoT) is an emerging technology trend which has already made a noticeable impact in the Information and Communication Technology (ICT) area. The Internet-of-things has given rise to new technology and business opportunities in applications such as smart cities, industry automation and big data [2]. To go further in depth, IoT is about connected physical things communicating with each other and this communication generates data which can be collected and analyzed. Several cities such as Bristol, UK and Santander, Spain have already made impressive technological advances integrating internet of things into the daily lives of citizens [3][4]. Santander, which has been dubbed one of Europe's first smart cities, has thousands of sensors deployed in its city which give citizens information about everything from available parking spaces to trash collection. Additionally, Bristol, which has taken on the moniker "The programmable city" uses a fleet of 1,500 sensors mounted to lamp posts in order to analyze traffic patterns to minimize traffic congestion. Both of these cities are examples of how IoT technology can affect change in city landscapes and give rise to new cutting edge applications.

ICN has been researched in several different protocol applications such as Named-Data Networking (NDN), Network of Information (NetInf) and Content-Centric Networking (CCN)[5]. In this project we have focused on Content-Centric Networking (CCN). CCN is a protocol that routes and delivers named pieces of data at the packet level of the network. This allows the network to cache data on any CCN relay and this is how the data is made independent from the location [6]. In other words, once data enters the CCN network it can be cached on any CCN relay, which means that any CCN relay can respond to request for data if the CCN relay has that data in its cache.

For this project, we have taken advantage of these ICN caching and naming designs and applied them to the IoT sensor domain, essentially by directly mapping sensor readings to a CCN name space. In this way we have built a functional ICN-IoT network infrastructure. Furthermore, we have devel-

oped an Android application that uses this ICN-IoT network infrastructure to display sensor data which is requested and transported using the ICN networking.

The ICN development comes with several challenges to be aware of, such as naming data, handling data streams, handling mutable data entities, name-based routing, and searching for data in the network [5]. During the course of the project, we met all of these challenges and through research, testing, and discussion, found solutions that worked in the scope of this project. This report focuses on the design choices and trade-offs that were made.

## 1.1 Project computer science & the GreenIoT project

The development of the system which this report covers is developed by students taking the project computer science course at Uppsala University [7]. The course gives student the opportunity to get insights on how a big development project is run from planning to realization with a industrial partner as a customer [7].

This year's project is carried out in the context of the GreenIoT project, which is funded by the Swedish Innovation Agency Vinnova. The project is a collaboration between Uppsala University, Royal Institute of Technology, SICS, Uppsala Kommun, Ericsson, IBM, SenseAir, Upwis, and 4Dialog [8]. In our project, we have had contact with Uppsala University, SICS, and Ericsson.

The GreenIoT project aims to develop an environmental sensor system and a platform that enables both public sector and private actors to develop applications for sustainable urban development, based on open data from sensors. Currently, the GreenIoT project and the Uppsala Kommun are deploying a testbed of sensors along Kungsgatan to measure air quality. When operational, the measurements will give the municipality a better foundation for understanding air quality in Uppsala and hopefully lead to improved urban development[9]. Another key commitment of the GreenIoT project is to make the collected sensor data available to the public as open data to encourage third party development[9]. The idea is that by making the air quality readings available, industries in the private sector will use this information to create applications that will serve the public.

## 1.2 Objectives of the project

The overall goals of the project was to:

1. Experiment with the deployment of an ICN-IoT network infrastructure.
2. Develop and use sensor software which integrates sensors into the ICN network.
3. Develop an Android application that requests sensor data through the network and makes use of the sensor data.

The purpose of experimenting with ICN in the area of IoT is to identify the challenges of combining these two different technologies such as naming the data, handling data streams, handling mutable data entities, name-based routing, and searching for data in the network. Performance and scalability testing has not been in the scope of this project, however we have tried out some small scalability tests which will be described in the testbed section (section 9) of the report.

# 2 Background

## 2.1 ICN and CCN

The project is based on Information-Centric Networking (ICN) where the communication is based on named data [1]. In ICN, devices communicate with each other by providing or requesting named data which is independent of its location. As the data is independent from its' location, it means that all the nodes in the network can provide any piece of named data [1]. Therefore it is possible to cache data on any node in the network. Each node in a ICN network is called a relay. The intent of ICN is to replace host-centric IP networking in the future.

For our project we used an implementation of ICN called Content-Centric Networking (CCN). CCN is a protocol that routes and delivers named pieces of data at the packet level of the network. This allows the network to cache data on any CCN relay and this is how the data is made independent from the location [6]. In other words, once data enters the CCN network it can be cached on any CCN relay, which means that any CCN relay can respond to request for data if the CCN relay has that data in its cache.

### 2.1.1 Naming

Just like common IP routers, ICN uses prefix matching to forward interest messages and to respond with content objects in a network. The main difference is that in ICN's case, prefix matching doesn't use IP addresses, but the names of the data that is being requested. To illustrate, in a traditional IP network, a request could be made for a web page by a URL that would contain an IP address and optionally a page name, say for example: 100.92.168.72/bob_car.html. This request is made to a specific IP for a specific HTML page, and the response will contain information pertaining to Bob's car. In ICN, on the other hand, the equivalent request would be made by sending a request for: /car/bob. This is a request for a named piece of data called: /car/bob and the request can be responded to by any node in the ICN network that has the data: /car/bob. And that response will contain information pertaining to Bob's car.

In order to maintain the operation of an ICN network, three specific data structures are used:

1. Content Storage (CS)
2. Forwarding Information Bases (FIB)
3. Pending Interest Tables (PIT)

### 2.1.2 Content storage

Every relay has its own content storage, which acts a content object cache for that relay. The goal of the CS is to store data objects so that if they are requested again they can be sent back immediately. If the content object is not in the relay's cache, the request is forwarded to another relay in the network using the Forwarding Information Bases and an entry is made in the Pending Interest Tables to denote that the relay is waiting for that content object. When the content object is received, it is sent back to wherever the original request came from and a copy of the data object is stored in the CS.

### 2.1.3 Forward information base

The role of the FIB is to keep track of the information that the relay will use to forward named data requests to other relays in the network. In CCN-

lite, each relay can have multiple connections to other relays in the network. These connections are referred to as faces [10]. For example, if relay A has a connection to relay B it can be said that relay A has a face to relay B. Furthermore, if relay A knows that it can ask relay B about a certain name prefix, for example "GreenIoT", we can build a forwarding rule that says if relay A receives a request for data objects starting with "GreenIoT" it can forward those requests, along its corresponding face, to relay B. These forwarding rules are all stored in the Forwarding Information Base data structure in the CCN-lite relay and these rules make up the network connections in the ICN network.

### 2.1.4 Pending interest tables

The role of the PIT is to store what interests have already been requested by that relay and where that request came from. Whenever a request is made for a prefix, this request traverses through the network, each time storing an entry in each relays' PIT. When the request finally reaches a relay where the data exists, a response object is created and sent back along the path of relays that forwarded the request, to the relay that originally requested the data.

For every new node that the interest is forwarded to, an entry in the relays PIT is created to keep track of what content objects have already been requested and where each request came from. If an additional request for the same content objects arrives at the relay, the relay will add that request to the already created PIT entry for that content object, so that when the named data object arrives at the relay, the relay knows it needs to send two response objects, one for each request it received. After which the entry for that content object is removed from the PIT, and the content object is added to the content storage.

## 2.2 Software used

In this section we will briefly describe the software we have used during the project.

### 2.2.1 CCN-lite

CCN-lite is a lightweight and functionally inter-operable implementation of the Content Centric Networking protocol CCNx of XEROX PARC [11], which has been developed and is maintained by the Computer Networks group at the Mathematics and Computer Science Dept of the University of Basel. CCN-lite was developed in part because PARCs' CCNx router software was too complex. CCN-lite is instead a smaller alternative for educational and experimentation purposes [12].

In the context of our project, CCN-lite covered all the CCN functionality that we needed. CCN-lite includes CCN data structures such as PIT and FIB, matching of publishers' public key to fight cache poisoning, nonce and/or hop limit tracking to avoid loops as a minimal safeguard and encoding of messages to the CCN format [12].

### 2.2.2 MongoDB

MongoDB is an open source NoSQL datastore. NoSQL is a collective name for the increasingly popular ways to build a datastore that violates the rules of traditional relational datastores. MongoDB is a document datastore, where a document in this case is not a PDF or a Word document, that consists of JSON documents. A JSON document, in turn, is a collection of pairs of keys and values. Below is an example of a document for a book [13].

```
{
title: "JSON Title",
author: "Ludwing Franquiz",
published: "2016-10-04"
}
```

Listing 1: Example JSON

### 2.2.3 Contiki

Contiki is an open source RTOS developed by Adam Dunkels et al. at the Swedish Institute of Computer Science. The operating system is written in C and uses an event-driven kernel. Contiki was developed for the Internet of

Things and offers official support for Texas Instruments MSP430 and Atmel AVR [14].

### 2.2.4 MQTT

MQ Telemetry Transport (MQTT) is a lightweight publish-subscribe broker on top of TCP/IP. MQTT is widely used and has implementations in C, Java, Javascript and Python. The lightweight broker is responsible for receiving messages and distributing these to all subscribers. Every interested client subscribes to a topic which refers to a specific data. In our project, each topic represents a data stream of sensor values produced by a sensor.

## 2.3 Related work

The internet of things (IoT) has become a vivid research topic over the last few years. It provides new possibilities, and shapes the future of mobile devices. Unfortunately, the novelty of the GreenIoT project where we built our work, does not allow for an extensive comparison with other research ideas, because such related approaches are rather limited and only partially intersect with our approach. Perera et al. [15] discuss the the concept of sensing as a service, and its correlation with the IoT in technological, economical and social perspectives. Zanella et al. [16] provide a comprehensive survey of the enabling technologies, protocols, and architecture for an urban IoT. This work presents the technical aspects and best-practice guidelines adopted in the Padova Smart City project.

Content Centric Networking (CCN) is a content-based network architecture. Instead of relying on a host based protocol CCN allows for access and retrieval of content by name, irregardless of where the data is physically stored. Oh et al. [17] propose an Content Centric Networking in tactical and emergency MANETs by extending a basic CCN architecture to support disruptive networks. Kim et al. [18] addresses broadcast storming during content dissemination, as well as the increasing content announcement overhead in MANETs using a topology aware CCN variation.

## 2.4 Examples of IoT for smart cities

### 2.4.1 SmartCity Santander, Spain

As mentioned in the introduction, Santander in Spain is one of the most prominent examples of incorporating IoT into a city. Starting in 2011, with 11 million Euros in funding, Santander started to design and deploy a sensor fleet to help drivers find available parking spaces. Since then they have deployed approximately 3,000 sensors in the city for a number of different projects. Around 2,000 of these sensors are used for environmental monitoring in the city center and collect data for temperature, CO, noise, and light. Santander also has about 60 sensors for monitoring traffic volumes, road occupancy, vehicle speed and queue length. As well as 50 moisture temperature and humidity sensors for monitoring and optimizing irrigation in the city. All of these sensor readings are made available for the public via an online portal and an iPhone and Android mobile app.

### 2.4.2 Bristol, UK The programmable city

Bristol, UK is also a noteworthy example of incorporating IoT into a city. Their aim is a little different than in Santander. In Bristol, their focus is on opening up the cities data sets to the public and businesses to encourage development and experimentation with IoT devices. They currently have a "City Experimentation as a Service" beta which allows researchers and companies to run experiments with the sensors deployed in the city. Additionally, they are planning to roll out a fleet of IoT sensors as well as provide public WiFi for their harbor.

# 3 System architecture

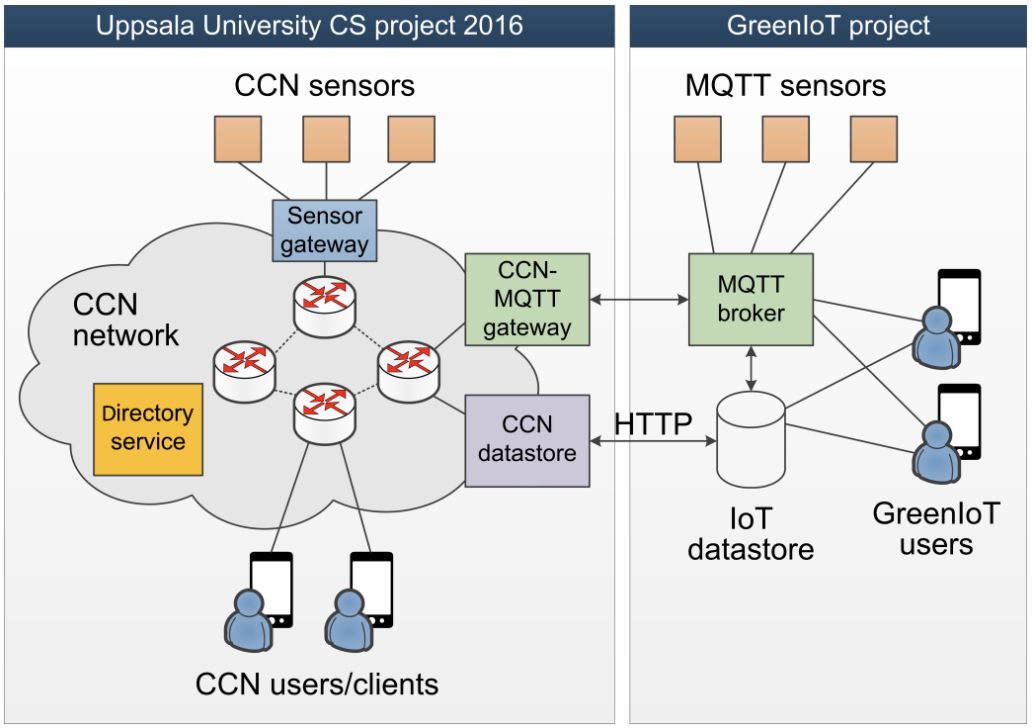



Figure 1: System architecture

Figure 1 is a bird's eye view of our project and the bridge to the GreenIoT project. The whole project is divided into two systems, the one to the right in figure 1 is developed by us and the one to right is developed by the GreenIoT project team. The project has two type of sensors, CCN and MQTT sensors, and in this project users retrieve sensor data through the CCN network which consists of CCN relays, which are the routers in the CCN network, the directory service and a CCN-MQTT gateway. Relays are responsible for forwarding interests, keeping track of all the interests in the PIT, storing content objects for potential reuse and creating content objects when interests are sent.

The CCN network also hosts a directory service (DS) which function is to keep track of the sensors and the routing between relays. The DS has all the information about the relays, border routers and sensors within the network. By possessing this information, the DS is able to administrate the network.

The CCN-MQTT gateway is the bridge between the sensor data produced by the GreenIoT project and our project. The gateway is able to produce a bidirectional stream of sensor data to both sides of the system architecture in figure 1. A MQTT subscription is implemented to get sensor data from the GreenIoT sensors and a MQTT publisher is implemented to publish the data produced by the CCN sensors and make it available for the users in the MQTT domain.

The Border Routers devices are the sensor gateways that connect the sensors to a computer running a CCN relay by creating routes from the CCN relays to the sensors, using the sensors' IPv6 addresses. On every computer running a border router a register service is running, which is responsible for register and unregister the sensors in the network. The sensors produce values at constant intervals and their values are requested by the relays. Whenever their values are stored in the relays, they do not have to be requested again, since they will be stored in the relays' content storage.

In the GreenIoT project the data sensed by the sensors is stored in the IoT datastore. The IoT datastore serves as long-term storage for all raw and processed sensor data. The goal from the beginning of the project was to use the same datastore for the CCN domain. However we decided to create a separate CCN datastore for our project, since we wanted to have working system without being dependent on that the GreenIoT system is up and running.

The CCN datastore is a non-relational MongoDB datastore which is used for storing historical data. The function of the datastore is for us to have the possibility of requesting old sensor values and display them in the Android application. Another function of the datastore is the possibility it gives the GreenIoT project to request historical sensor data. This has been done by implementing a RESTful API which makes the historical data available for the the GreenIoT project.

The datastore also allows us to practice machine learning to predict future sensor values in the Android application. The prediction function is implemented as a named function which means that it is triggered and creates the content object for the predicted values only when a specific interest is sent.

The CCN users use the Android application in order to make use of the sensor data within the network. We decided upon a monitoring application which displays the temperature, humidity and light at lunch places in the university.

In order for the application to be part of the CCN network, the application communicates with a CCN relay that runs as a service on the phone. The relay is a separate Android application package which runs in the background and the purpose of which is to transmit the applications' request to the CCN-network through a gateway and to return the content objects.

On application startup, the CCN users first needs to set the ip of a computer running a relay in the CCN network, this in order to create a connection between the relay running on the application and the relays in our CCN network. The clients will then through the relay running as a service send an interest to receive the list of active sensors. The list if gathered by the directory service by remotely querying the datastore for all sensor entries with the a-flag set to True. These sensors can be used to get current sensor values and also historical and predicted values which will be described thoroughly in section 9.

Historical data can be requested by the users using a different prefix than for current sensor values. The interest is propagated further to the relay running on the computer where the datastore is running. When the interest reaches the datastore relay a named function is triggered, which provides access to functions that perform operations using multiple data objects, which can produce passive content once it is needed [19]. A content object is then created with all the sensed values during the previous 24 hours. This content object is then sent back the same way that it came from, finally reaching the clients.

Predicted data can be requested by the users using a named function similar to the historical named function presented above. The interest is propagated to the computer running the datastore using the same path as for historical data. The prediction named function is triggered, and a content object with predicted values for the coming day in that specific requested location is created and propagated back to the user.

When a specific historical value is requested it is forwarded to the other relays in the CCN network as well as the relay running on the datastore computer. If it reaches the datastore relay a named function will be triggered which will query the datastore for the specific requested value, create a content object with the requested value and send it back the same way it came from. If it reaches another relay having the value before the datastore relay the named function will not be triggered and the relay which already has the value will be the one responding the interest with the content object.

# 4 ICN-IoT implementation

## 4.1 UML implementation diagram

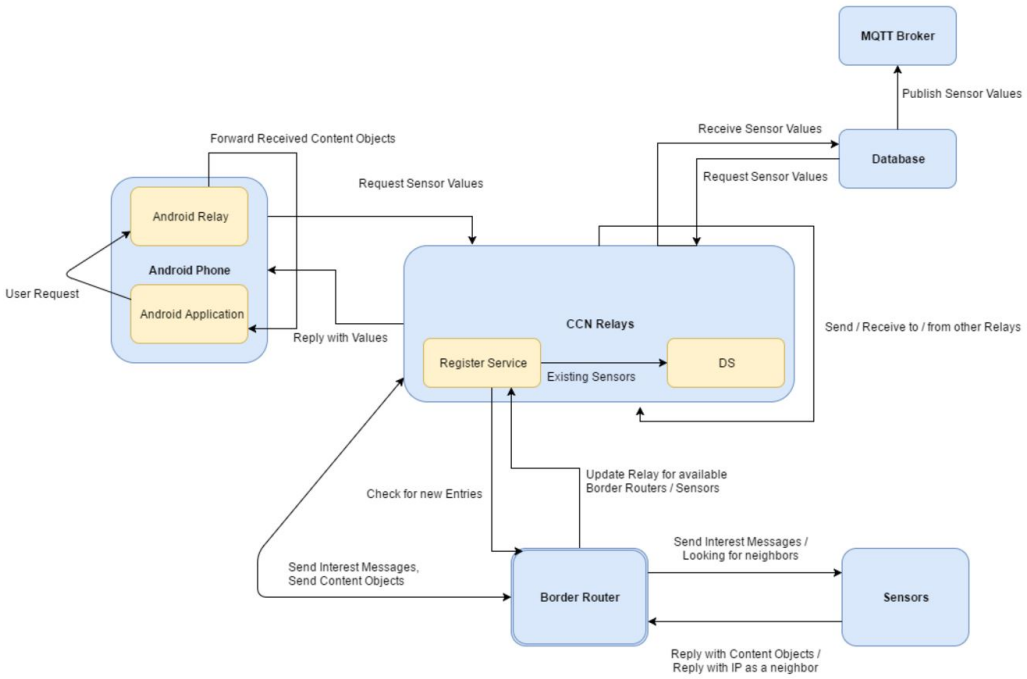



Figure 2: UML implementation diagram

In figure 2 the rectangles represent the entities of the system and the arrows with the corresponding labels represent the functionalities when the entities interact with each other. The blue rectangles are the main entities (ex. Android Phone) and the yellow rectangles included in the blue ones are the key entities (services) that are running in the main ones (ex. Android Relay, Android Application).

The Android Device includes two different entities, the Android Application, containing the UI that users interact with and the Android CCN relay which is responsible for sending Interest Messages and receiving Content Objects to the main CCN relays. Whenever a user requests a value, the Application communicates with the relay to send an Interest Message. In the same way when the relay receives a Content Object it parses the information and shows them in a human readable format to the user through the Application.

The datastore is requesting data from the CCN relay in constant intervals and stores them. This information will be served to the user in case the information from the sensors is not available and also for archiving reasons.

The MQTT Broker is the bridge between the GreenIoT project and our project and allows us to subscribe to the data produced by the GreenIoT sensors and to publish the data produced by the CCN sensors and make it available for the users in the MQTT domain.

The Border routers acts as "middle man" between the sensors and the CCN relay. They are looking for sensors in their range, which they store and inform the relay about their existence. They also receive Interest Messages from the relay which they forward to the sensors through the routes they have created and also receive Content Objects from the sensors which they forward back to the CCN relay.

The Sensors are collecting measurements from the environment and transmit these values in a CCN format whenever they are requested through the border routers. They are also signaling their IP when they are online , so the border routers can locate them.

The CCN relays are the main entities of the system. They are running on PCs and are responsible for handling all the traffic in the Network. They contain two key sub-entities, the DS and the Registry Service.

The Registry Service is a sub-entity of the CCN relay and is responsible for requesting the border routers for their sensors list and forward this information to the DS.

The DS entity administrates the network by keeping track of the sensors and the routing between the relays. By receiving the sensor list from the register service the DS is responsible for updating the list of active sensors in the datastore. The DS also is responsible for configuring the forwarding rules of the relays in the network.

The CCN network also hosts a directory service (DS) which function is to keep track of the sensors and the routing between relays. The DS has all the information about the relays, border routers and sensors within the network. By possessing this information, the DS is able to administrate the network.

## 4.2 Directory Service (DS)

In order to organize and configure the collection of sensors, the sensor data and the forwarding planes of the relays in the network, we implemented a controller drawing upon concepts from Software Defined Networking (SDN) [20]. By separating the forwarding and control plane of our network architecture, we're able to configure and optimize the data forwarding in the network without having to configure each relay separately. The DS has complete knowledge about each relay, border router and sensor in the network and is able to make forwarding decisions based on this information. The DS is also responsible for retrieving a list of available sensors from the datastore and propagating this to each Android client that wants to request sensor data.

**Startup**
When the DS is started, it reads all global variables from the init.cfg file in the /config folder, such as the IP address to the sensor datastore and several other parameters. After a connection is successfully made to the datastore, it opens an UDP socket on port 9999 to listen to incoming packets in a while loop.

**Relay and link registration/de-registration**
Once a relay boots up, it sends a JSON packet with its status - On. When the DS receives this packet, it adds the corresponding IP of the relay to its dictionary NETWORK, where each key represents the IP of a relay in the network. Furthermore, whenever a relay in the network creates a face towards another relay, a JSON packet with the IP of the new face is sent to the DS as well. The DS then adds this IP as a value in the list belonging to the key (IP) of the relay which created the face. The structure looks like the following:

```
NETWORK:
[RELAY_IP] : {PEER1, PEER2, PEER3, ...}
[130.238.15.242] : {130.238.15.241}
[130.238.15.241] : {130.238.15.242}}
```

Additionally, before a relay goes offline in the network it sends a JSON packet to the DS with the status - Off. Once this packet is received, the DS removes the key corresponding to that IP from the NETWORK graph, and removes all entries with this IP from each relays list of faces. By doing this, the DS is able to keep a graph of each relay in the network and how they're connected each other. This allows us to manage the forwarding plane of

our CCN network from one centralized entity. This implementation is dependent on the relays being shut down in a controller manner, see section 9.1.

**Register Service (RS)**
The register service is a program that runs on every computer that has a border router. These programs work in conjunction with the DS to publish the list of active sensors to the clients. The purpose of this software is to detect new sensors that connect to the border router and to send their information to the DS via UDP message. The RS reads all connected IPv6 sensor addresses from its local border router and registers those sensors to the DS. The RS has its own configuration file that includes the IP address of the closest relay, path to the border routers IPv6 list and, most importantly, the name of the place where the border router is located. When a sensor is connected to a border router the RS calculates the MAC address of the sensor from the IPv6 address and creates a route from the relay to the sensor and requests sensor information from the sensor. After receiving the sensor information, the RS sends an interest message to the sensor containing the border router location. This interest message acts as a control message. When the sensor receives the interest message, it parses the location name out of the prefix and rebuilds its naming prefix using that location name.

When the sensor goes offline or out of the border routers range, the RS sends a deactivation message to the DS which then sets the sensor state to inactive. To clarify, the sensor isn't completely removed from the DS, it is just set to inactive.

**Sensor registration**
The register service (RS) sends information about one, or multiple, new sensor(s) to the DS, by sending a JSON object. The DS parses this JSON object to see what the MAC addresses of the sensors are, as well as to get the prefix of the content that the sensors produce. If no sensor with the indicated MAC-address exists in the sensor table, an insertion is made. In the case that the sensor was previously registered, the status of the sensor is set to active or inactive depending on the type of message.

Since the border router and a relay run on the same machine concurrently, the DS knows that new sensor information coming from a relay is actually coming from the RS running where the border router is. When the DS receives new sensor information it adds that information as one of the keys in the NETWORK graph.

**Forwarding plane**
Once a new sensor is added to the datastore, the DS proceeds to update the forwarding table of each relay in the network with the prefix of the content that the sensor produces. It does this by running the following shortest-path algorithm on the each relay in the network graph:

```
def find_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    if not graph.has_key(start):
        return None
    for node in graph[start]:
        if node not in path:
            newpath = find_path(graph, node, end, path)
            if newpath: return newpath
    return None
```

The following function will return a list, where the first entry is the start (relay to be updated), and the following entries are the nodes that need to be visited (relays) to be able to reach end (border router). The IP of the relay following start (next) is extracted, and a UDP packet is sent to the start relay, with a command to update its forwarding table with a path to the prefix through the face corresponding to next. Once this is done for all relays in the network, each relay knows the shortest path to reach every prefix. In order to be able to reach old content that is no longer available in the sensor or relay caches, each relay also receives an entry in their forwarding tables which indicate which path to take to reach the datastore of sensor data, in accordance with the algorithm above. When a relay propagates an interest through any of its faces, it also sends a duplicate interest through the face leading to the data store.

This forwarding mechanism is limited to work only in a small-scale network with few sensors and relays, see 9.1

**Sensor lists**
When a CCN interest packet is received by the DS with the prefix *sds/sensor*, it retrieves sensor information from the datastore, and builds a JSON object of all of the active sensors (see 9.1). The resulting JSON is parsed to extract

the necessary information that the Android client needs to request data, such as the prefix of the content, the time the sensor came online, what kind of data it produces, and how often it produces new data. This is formatted as a CCN content packet and sent back to the client. Since each forwarding table of the relays in the network contains a path to the content indicated by the prefix that belong to a sensor, as well as a path to the datastore, the Android client is able to retrieve all content being produced in the network with those prefixes.

## 4.3 Routing

As previously stated, we have used CCN-lite as the foundation of our Content Centric Network. The existing CCN-lite repository that we have used during the project covers all the basic Content Centric Networking operations e.g. sending and receiving interest messages and content objects. However, automated forwarding rules between relays were not included in the basic repository. Furthermore, we implemented a new cache eviction algorithm using the first in, first out (FIFO) method, where the oldest content object in the cache is removed if the cache gets full. In the current implementation, the cache size (N content objects) is decided when the relay is setup.

Each relay is auto configured when introduced into the network. On startup, each relay sends a broadcast packet to all direct neighbors. This packet is formatted into an CCN-lite interest object with the prefix */ccnx:/discover* which is received by all neighbors. To reach all neighbors, we open a socket to the CCN relay default port 9695 and send the interest using UDP to the broadcasting address of the local network "255.255.255.255". When a relay receives an interest with this prefix, it takes the senders IP address and creates a face towards the sender if and only if there is no prior established connection. The relay then constructs a CCN-lite content object with the prefix */ccnx:/discovered* and sends it back to the broadcaster who then creates a face back to each active relays that replies to the initial broadcast interest using the same technique described above. This back and forth message is used to automatically create routes between the relays in the network. As previously mentioned, the relays send a JSON packet to the DS with the status "inactive" whenever it goes offline. The DS is then tasked to remove all faces connected to the relay that sent the message. This means that only the DS is fully responsible of de-registering inactive relays. However, one issue is that this message is not sent if the relay crashes or the computer

running the relay shuts down. One solution to this problem is to create a heartbeat system, where the DS sends packets to all active relays. All relays that do not respond to this packet are treated as inactive and removed by the DS from the network.

Another major step of our implementation of the relays is auto populating the FIB tables. Our algorithm is divided into several steps depicted in the following figure:

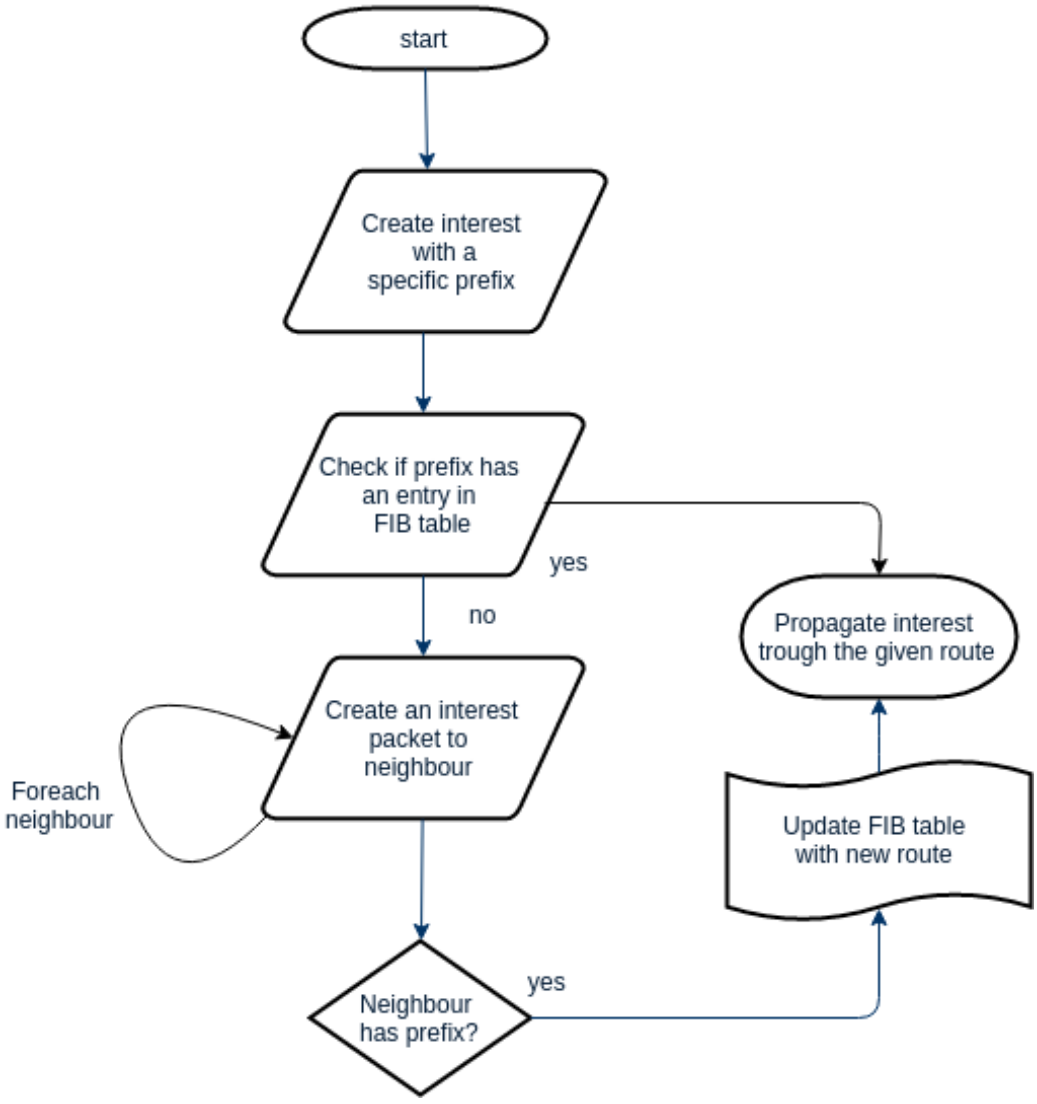



Figure 3: Auto populating FIB table algorithm.

Using the algorithm presented above, the relays are able to auto populate

their FIB tables every time an interest with a new prefix is propagated in the network.

The combination of broadcast messaging and FIB auto population creates a foundation for a robust CCN network where multiple nodes can be inserted into the network without any extra configuration.

## 4.4 Sensors

The sensors we are using in this project are the Texas Instruments CC2560 with the Debugger DevPack XDS110. The sensors are running Contiki OS and are able to implement CCN functionalities. The sensors are able to communicate with neighboring devices wirelessly using 6lowPAN protocol [21] over 802.15.4 [21]. The sensors are also using RPL protocol [22]. The DevPack XDS110 has a microUSB port which is used to flash the sensors and provide power to them when they are connected to a computer. The core CCN source code for the sensor was provided by SICS and we have done some modifications on it, in order to adapt it to our system. The commit id of the code is included in the references section [23]. The ccn-lite version we used is *Release 0.3.0* - July 2015 from University of Basel [24]. Although the TI CC265O Devices can use a variety of sensors not all of them are available in Contiki OS. The ones that have drivers currently working for Contiki are:

- Temperature
- Humidity
- Light
- Gyroscope
- Pressure

Because of the sensors limited memory, they are able to cache only a small amount of content objects, 15 in our case. The sleep interval of the sensors is easily configurable and can be set to any value that serves the projects design. For this project we are using only 3 sensors which can sense temperature, humidity and light. The sensor is able to receive CCN interest messages from nearby border routers and configure its prefix according to that. Since the border router knows its own location and the sensor can configure its own prefix based on that location, the prefix is given to the sensor by the border router and then that information is reported to the DS. The sensor, on

startup, produces a content object with the prefix ***/p/12345678/regist***. *12345678* represents the last 8 characters of its MAC address, since the buffer was not big enough to fit all 12 characters. This initial content object contains information about the sensor model-type and the current sensor measurements. Since the sensor should always be able to contain a Content Object with its own information, this initial content object is produced once every 15 loops, so there is always one cached and available to be requested. The location - and the prefix - of the sensor can be changed by the border router. The text contained in every Interest Message is parsed into a string which is searched to see if it contains the word "regist". The word "regist" indicates that this is a registration message so the border router either requests the sensor information or wants to register the sensor to a new location. In case after the word "regist" more text follows, this text represents the actual new location that the sensor will have to use. For example by receiving the prefix ***/p/12345678/regist/myroom***, the sensor knows that *myroom* is the new location that it should create content objects for. The registration now has been done and the prefixes from now on will have the following format: ***/p/myroom/12345678/seq_no***. The sequence number starts from 1 in every new registration. The location can change again by sending again a registration message containing a new location. The sleep interval is currently set at 10 seconds. Every 10 seconds, the sensor is triggered to produce new values. When all 3 values have been produced, the sensor creates a CCN content object by calling the corresponding function. The 10 seconds interval is configurable and can change easily. If an interest message arrives while the TI CC2650 is sleeping, it will wake the device up and look into its cache about the requested content object. This process is not affecting the timer which will create the next content object 10 seconds after the last creation happened. Depending on the prefix that has been set, the object will be cached in the sensor in the following format: ***seq_no - sensor_interval - Light_value - Temperature_value - Humidity_value***. The sensor does not contain a clock that keeps track of the actual date and time so it is not possible to include time in the content objects. Although by including the content object creation interval and the sequence number in the content object, the rest of the systems are able to keep track of the real time by correlating these values with their working clocks.

## 4.5 Naming

Named data is the core functionality in Information Centric Networking. Naming in CCN is hierarchical and segment based, which means data names

are ordered and split into different parts. Naming is implemented by labeling each piece of data in an application, in a structured way, so that each piece of data has a name associated with it and vice versa. Take for example the name: **/seg1/seg2/seg3**. We can see that there are three parts to this named data object and that segment one is at the highest hierarchical level and segment three is at the lowest. Furthermore, CCN names are generally classified into different parts. At the beginning there is a globally routable name segment which is typically the application or project name. Next there are application dependent name segments, which in our application includes the sensor location and sensor id. And at the end there can be protocol dependent name segments which are similar to parameters and can specify information about different data chunks or different versions.

Throughout the course of the project, we used several different naming schemes. In the beginning, when we were testing with only one sensor, we used a naming scheme that looked like **/demo/mote_id/seq_id**. Here **/demo** was the project name, **/mote_id** was the sensor ID and the only application dependent name segment, and **seq_id** was the sequence number of that data that the sensor had produced. In this naming scheme we could specify which sensor and which piece of data we were requesting, but we did not have the sensor location incorporated into the naming. We debated as to whether or not the sensor location should be included in the naming. One on hand we did not have a massive amount of sensors and the project could have worked with us hard coding which sensor is in which location, however, due to the specification that sensors should be automatically configured, we decided to incorporate the sensor location into the naming hierarchy. Specifically, each border router is manually configured with a location, and every sensor that connects to that border router receives that location from the border router and uses it in its naming prefix. Thus we came up with a naming scheme of **/p/location/mote_id/seq_id**. In this way, we are able to organize sensors based on which border router they are connected to.

The next challenge we faced we was adding support for historic and predicted values. In other words, we needed a way to distinguish between normal sensor data, sensor data that was stored in the datastore and requests for predicted data values. Idealistically, we wanted to keep the same naming convention of **/p/location/mote_id/seq_id** and have the network be smart enough to be able distinguish between the different requests. For example, if a client requested older data that would most likely be found in the datastore, that request would be routed to the datastore. If a client requested current data, the request would be routed to a sensor and if a request was made for future data the request would be sent to the relay responsible for computing

predicted values. When the Android Application requests historic values it requests data with the naming scheme **/historical/location** and when it wants predicted values it uses **/prediction/location** and the network knows to route those requests directly to the datastore or the relay responsible for computing predictions.

The application knows what to send interest to and how to distunguish between the different naming conventions by having a interest function implemented where the user needs to specify what task it wants to execute. For example if the user in the application wants to retrieve the sensed values from the past 24 hours, the interest function will use the historical task which includes the naming convention for historical values. If the user instead wants to retrieve a specific historical value from a certain date it will run the specific historical task which first needs to calculate the sequence number from the date provided by the user in order to send an interest with the correct naming convention, which for specific historical values is the same as for current values.

## 4.6 Sensor markup language

All the sensor data in the project is stored using sensor markup language (SenML) [25], which is simply JSON containing named events together with an associated value and unit. In the JSON there is both information about the sensor but also about the events, which in our case are new sensor readings. The purpose of the sensor markup language is to make the gathering of data from multiple devices easier. Here is an example of our implementation of the sensor markup language:

```
{
        "a": true,
        "md": "TI CC2650 SensorTag",
        "ver": 1,
        "lo": "Office",
        "bn": "00:12:4b:49:8c:82",
        "but": 10,
        "bt": 1481194072,
        "bsn": "145",
        "pf": "/p/office/4b498c82",
        "e": [{
                "v": 26.625,
                "sn": 1,
                "u": "C",
                "t": 1481194072,
                "n": "Temperature"
        }, {
                "v": 26.86,
                "sn": 1,
                "u": "Lux",
                "t": 1481194072,
                "n": "Light"
        }, {
                "v": 22.02,
                "sn": 1,
                "u": "%",
                "t": 1481194072,
                "n": "Humidity"
        }]
}
```

Listing 1: Sensor markup language

What each field stands for in the JSON is described below:

- a - Indicates of the sensor is active or not.
- md - The model of the sensor.
- ver - Version.

- lo - Location where the sensor is placed.
- bn - MAC address.
- but - Looptime, the interval between sensor readings.
- bt - Epoch time when the sensor was initialized.
- pf - Prefix of the sensor.
- e - Contains array of events.
- sn - Sequence number for the specific reading.
- u - Unit.
- t - Epoch time when the values was sensed.
- v - value

## 4.7 Border routers

The Border Router devices are the sensor gateways that connect the sensors to a computer running a CCN relay by creating routes from the CCN relays to the sensors. The hardware is Zolertia RE-MOTE and the software they are running is *Sparrow-Border-Router* provided by SICS. Sparrow border router is a Contiki implementation in purpose of serving IoT applications as a border router. The commit id of the code is included in the references section[26]. More information about the sparrow border router can also be found here [**sparrowinfo**]. Border Routers are connected serially to the Computer using microUSB cables. On the border routers we have done very little modifications to the original code.

- The first modification was a change to the interval that the border routers uses to look for new neighbor routes. The initial time was 3600 seconds, so we reduced it to 36 seconds because we wanted to have an almost real-time behavior when sensors go on or offline.
- The second modification was writing all the routes of neighboring sensors to a text file which can be used by the system to create faces.

The border router crashes occasionally so a bash script was implemented to re-launch it whenever it crashes. The reason the router crashes has probably something to do with the serial connection, but we cannot state that with certainty.

# 5 Datastore implementation

As described in the section 3 the GreenIoT project has a MongoDB datastore which is used for storing historical data. This datastore would be sufficient for our implementation but because we wanted to have local access to the datastore we decided to standup and use our own MongoDB . This datastore has a MongoDB collection called *sensors*. This collection is filled with an array of MongoDB Binary JSON (BSON) objects with the SenML format shown above.

Furthermore, we have implemented a RESTful API to make the data stored in the *sensors* collection available for the GreenIoT project. This API is implemented using the python micro framework Flask [27]. The API has multiple routes that can be used to query the sensor data stored in the datastore. In the current implementation, a user can query all the data produced by all sensors, or query data produced by only one sensor or in one location. Users with access to the API are also able to insert sensory data into the datastore using POST-requests.

## 5.1 Storing of CCN sensor data

In order for us to be able to request and display historical sensor values in the Android application we needed to store the values from the sensors as they are produced. We developed a python program which queries the MongoDB to detect if there are any active sensors. If the sensor is up the python program creates a thread object for each sensor with the information needed to calculate the sequence number and to send an interest to the sensor. When the thread for the sensor is started we first calculate the sequence number. This is done by taking the current time in epoch minus the time when the sensor was initialized, which is called "bt" in the sensor markup language JSON. To get the current sequence number we then take difference divided by the looptime, which is the interval for how often the sensor senses new values. The current sequence number gives us the entire prefix we need to peek the sensor for new values. We also update a field about the sensor in the datastore called "bsn" which stands for the current sequence number so that other parts in the system do not need to do the calculation as well.

After the calculation of the current sequence number we can send an interest to the sensor for new values by using the full prefix, for example: pf: /p/office/4b498c82/23, where the number 23 is the sequence number calculated.

When the content object is received with sensor values for light, temperature and humidity we extract those values, create a SenML JSON object with those values and store the SenML JSON object in the MongoDB datastore. Before executing these functions again, the thread will sleep for the looptime of the sensor and then calculate the new sequence number and peek for new values again.

To account for sensors coming on and offline, there is another thread in the python program that listens for changes to the active sensor list from the DS. If the datastore receives a message from the DS that the sensor list has changed. All of the threads are killed, a new query to the MongoDB datastore is made to get the list of active sensors, and then the threads are restarted, one for each active sensor. This is not a scalable approach since we are killing threads that does not need to get killed and restarting threads that may contain active sensors, a future solution would be to add a function which compares the sensor list in the MongoDB datastore with the version of the sensor list that the python program is sending interests to. By adding this function we will only kill or start the threads that are necessary, instead of killing all and then restart the ones that are still alive.

## 5.2 CCN-MQTT gateway

The CCN-MQTT gateway acts as a bridge between the sensor data produced by the GreenIoT project and our project. This gateway should be able to produce a bidirectional stream of sensor data to both sides of the system architectures. During this project, the Eclipse Paho MQTT Embedded MQTT C/C++ Client Libraries [28] were used as a code base for the MQTT connection.

Firstly, a MQTT subscription is implemented to get sensor data from the GreenIoT sensors. Given a MQTT topic for each GreenIoT sensor, we use the MQTT broker to subscribe to a set of data streams by connecting to the correct domain address (mqtt.greeniot.it.uu.se) and ask for all data objects with the given topic. If all data objects are of interest, the wildcard topic "#" can be specified instead. All the data objects are in a JSON format similar to the datastore JSON objects listed above in section 5. Each MQTT message that is received is inserted into the cache of the CCN relay running on the same machine as the MQTT subscriber. Furthermore the received messages are stored in the MongoDB datastore using the MongoDB C Driver [29]. A program using this driver is used to connect to our datastore and inserts the

JSON payload received from the MQTT message into a separate collection (a MongoDB collection).

Secondly, a MQTT publisher is also implemented to publish the data produced by the CCN sensors. Similar to the storage of CCN sensor data, the MQTT publisher needs to trigger a publish message each time the sensor produces data, therefore we have implemented the MQTT publisher in the same python program described in section 5.1. The python library *paho-mqtt* [30] is used to establish a connection to the same broker as above and publish the data produced by the CCN sensors. The CCN prefix produced by the sensor is used as the topic for each published data. Using the CCN prefixes as topics, the wildcard can be placed to receive all published data for one sensor or for one location using the following topics:

```
Topic to receive all data from a sensor: /p/location/macid/#
Topic to receive all data for a location: /p/location/#
```

# 6 Machine learning for prediction

One of the functionalities in the Android application is that users can ask for predicted data values. When the Android application makes a request for prediction values, the request is made and the results are presented on a graph for the user. Using machine learning we can calculated likely future values based on previous sensor values.

## 6.1 Introduction to Time-series

A time series is a series of observations of a random variable indexed by time. A general model for a time series can be written as:

$$S_t = g(t) + \varphi_t$$

where g(t) is the deterministic function of time and the residual term $\varphi_t$ is the stochastic noise. It's a typical signal and noise pattern.

By studying and analysis of the time series we can understand the mechanism that generates the data and moreover we can make a prediction of the data in the future.

## 6.2 Weakly and strictly stationarity

Before we continue to next section, the concept of stationarity needs to be introduced. A strictly stationary process[31] means that the statics property of the data such as mean, variance and autocorrelation stay the same at any time internal. In order for prediction to make sense we need at least weakly stationary[31] which means either mean, variance or covariance stay the same during any period. One can verify this theoretically or can observe the patterns by the graph.

## 6.3 AR, MA, ARMA and ARIMA model

Prediction models exploit data patterns that exists in the data series and build a mathematical model upon those patterns. Modeling of time series data can have many different forms. Two widely used linear time series models are Autogressive(AR) and Moving Average(MA). In AR(p) model[32], the next value in the series is a linear combination of the past values of underlying random variable itself plus some random error term.

$$y_t = \varrho + \alpha_1 y_{t-1} + ... + \alpha_n y_{t-n} + \varepsilon_t$$

where $y_t$ is the value of random variable and $alpha_n$ is model parameter. For estimating parameters of AR process the Yule-Walker equations[33] are used.

However instead of past values of series, MA(q) models[34] is using past error and the model is given by :

$$y_t = \delta + a_1 \varepsilon_{t-1} + ... + a_n \varepsilon_{t-n} + \beta_t$$

where $\beta_t$ is model parameter assumed to follow normal distribution. The MA model is actually a linear regression of the current value against the random shocks of prior observations. The mainly difference between the AR model and and MA model is the correlation over time. In the MA model, noise disappears quickly, however, the correlation declines gradually in AR model [35] [36].These two models explain different aspects of stochastic dependence. AR model encapsulates a quality that the future depends on the past whereas MA model combine randomness to the series from past[37]. Naturally AR(p) and MA(q) can be combined together to describe time series known as ARMA(p,q) model. The model is given by:

$$y_t = \nu + \alpha_1 y_{t-1} + ... + \alpha_n y_{t-n} + \delta + a_1 \varepsilon_{t-1} + ... + a_n \varepsilon_{t-n} + \beta_t$$

In real-life application not always times series are stationary where the model is assumed but it can be made stationary by differention. Including differention ARMA(p,q) model becoming ARIMA model(p,d,q) where coefficient d stands for order of differention.

ARIMA model is generalized model for discrete time prediction. It can eliminate residual, remov trend which makes it fits our data better, Moreover, it's easy to implement with imperative programming language e.g. Python.

## 6.4 General approach for ARIMA

There is a step by step approach to analyze and predict time series data using ARIMA model.

- **Step 1: First the time series data should be visualized** in order to find any seasonality or random behavior.
- **Step 2: The time series data should be stationarized.** As long as the time series is not stationary, a time series model cannot be built. In cases where the data is not stationary, the time series should be stationarized by either detrending, differencing or transformation etc.

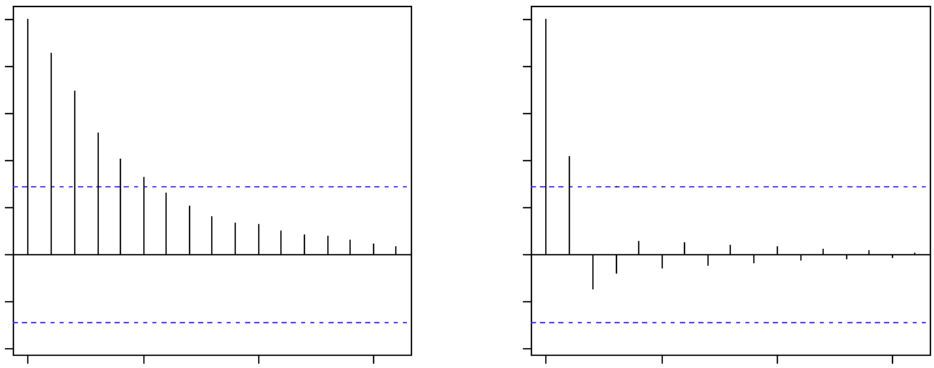
  There are some solutions to make a time series data stationary. Here, "Detrending" and "Differencing" are explained:

  - Detrending: In this technique, the trend component from the time series is simply removed. For example, if the equation of a time series is:

    $$x(t) = (mean + trend \times t) + error$$

    The "trend" can be removed before building a model for the time series.

  - Differencing : In this technique, the differences of the time series equation are modeled to stationarize the time series data. This differencing is called as the Integration part in AR(I)MA.
- **Step 3: Optimal parameters should be found for the ARIMA model.** As it has been mentioned earlier, there are three parameters (p, d, q) that should be optimized. In fact, it should be determined whether the model is AR or MA. In addition, the order of AR or MA should also be indicated. They can be found using ACF and Partial-ACF plots. These plots are explained later.



(a) ACF (b) PACF

Figure 4: ACF and PACF graphs for AR model

- **Step 4: The ARIMA model can be built by the indicated parameters.** However, sometimes the parameters should be investigated more.
- **Step 5: Predicted data can be obtained by the ARIMA model.**

[35].

## 6.5 AutoCorrelation Function (ACF) and Partial Correlation Function (PACF)

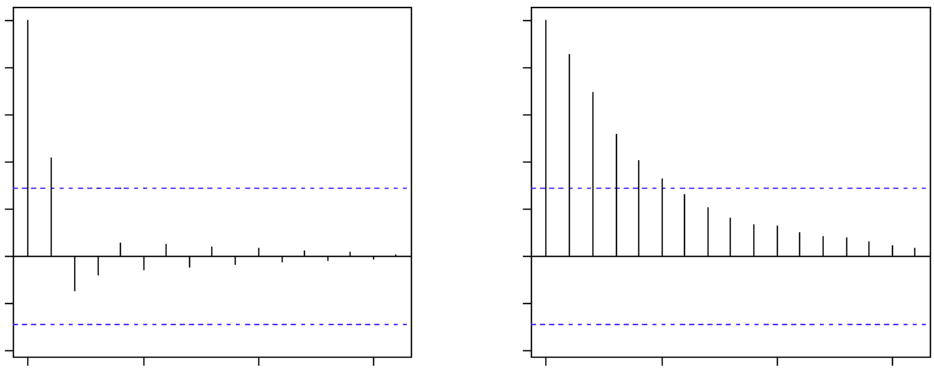
The plot of ACF is a plot of total correlation between different lag functions. For example, if x(t) is a random variable, the correlation of x(t) with x(t - 1) , x(t - 2) and so on are plotted.

There is no correlation between x(t) and x(t - n - 1) in a moving average series of lag n. So, the total correlation chart cuts off at nth lag. So the lag for a MA series can be found.

On the other hand, if it is an AR time series data, the correlation will gradually go down without any cut off value. In this case, PACF can be plotted to find out the partial correlation of each lag. The chart will cut off after the degree of AR series. Following are the examples which will clarify how to used ACF and PACF charts.

In the figure 4, the ACF graph shows that since the correlation will gradually go down without any cut off value, it is an AR model. The PACF graph shows a cut off after second lag which means it is an AR(2) process.

In the figure 5 , the ACF graph has a cut off after the second lag that shows it is a MA(2) time series [35].



(a) ACF (b) PACF

Figure 5: ACF and PACF graphs for MA model

## 6.6 Implementation

In this project, the ARIMA model is implemented in two languages: R and Python. R is a programming language and software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing. The ARIMA model that is used in the project is the python version. However, in our design we needed to implement the prediction script as a named function, but every time this function is invoked we need to find optimal parameters by using ACF and PACF plots which is not feasible. So if we use the default parameter settings which are (2,1,2), the prediction will fail and the result will converge. This is the an obstacle we can not overcome for now and also is a limitation for this project.

Below is the humidity data we collected from sensor deloyed in moebius with time in x-axis and humdity percentage in y-axis.

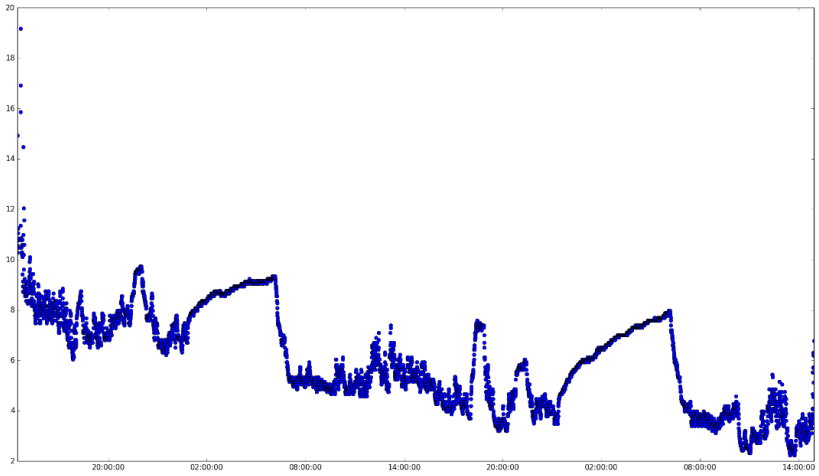



Figure 6: Humidity data from sensor deployed in moebius

As we can see from the figure, the values has a downwards trend and thus is not a stationary series. Therefore it is non-predictable. The number of samples is large which made it difficult to see the labels of axis so we partial zoomed in of the figure. And if we run the prediction with default parameters, the result is biased from the reality as we can observe from figure 8(we have to use year as x-axis which is limited by the plotting tool).

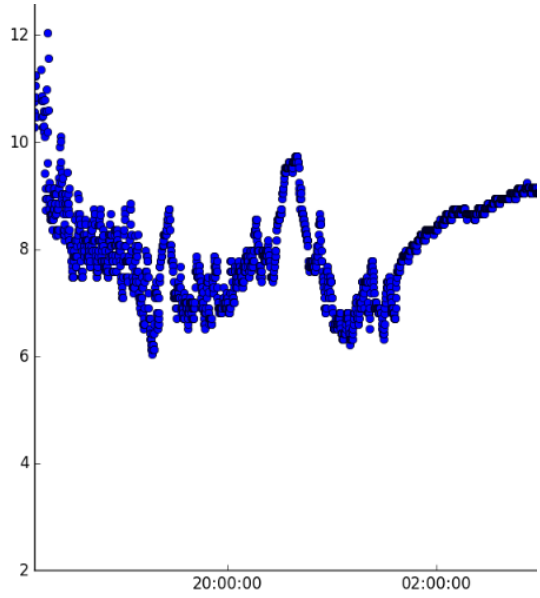



Figure 7: Partial zoomed version of figure 5

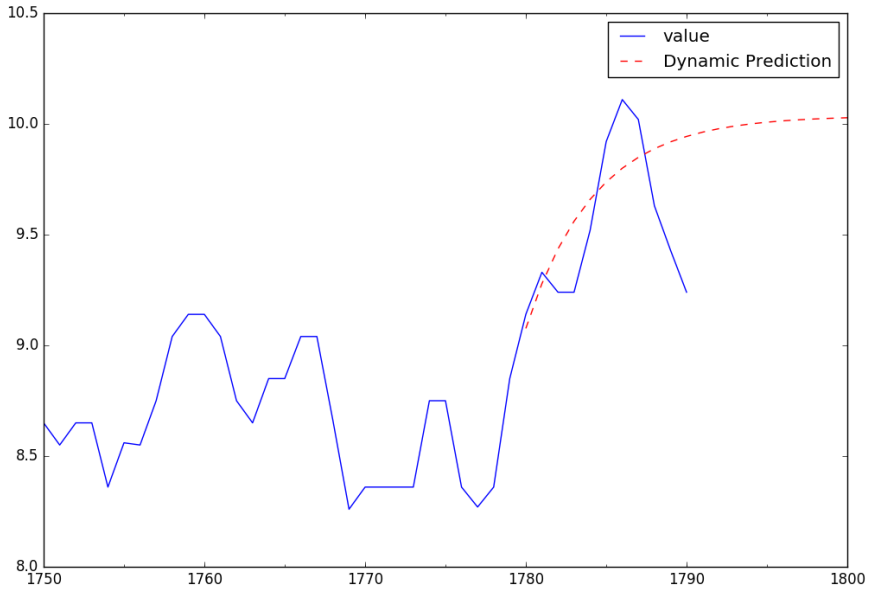



Figure 8: Prediction plot for first 90 humidity data from datastore using ARMA model

However, if we feed the prediction script a nice stationary series as input we will have a reasonable nice plot:

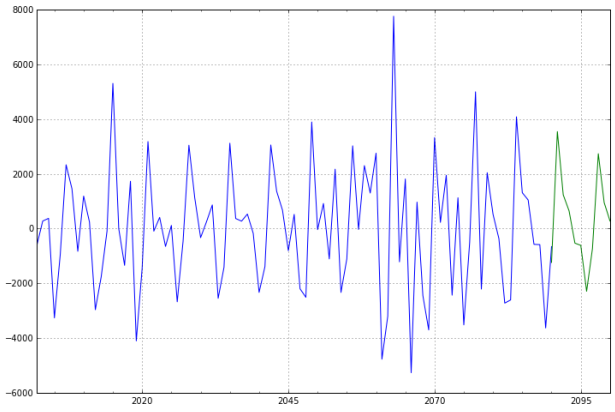



Figure 9: Prediction plot for a random stationary series

# 7 Named function networking

In order for the Android Application to receive the predicted and historical data values we have implemented named function networking. Instead of providing access to just data, named functions provides access to functions that perform operations using multiple data objects, which can produce passive content once it is needed [19]. For our project, we decided to implement this functionality in a python program. On program startup, the named function program sends a packet to the local relay. This packet is formatted into a CCN-lite interest object with the prefix */nfn* which is received by the local relay. To reach the local relay from the python program, we open a socket to the CCN relay default port 9695 and send the interest using UDP to the local network. When the local relay receives an interest with this prefix, it takes the senders IP address and creates a face towards the sender if and only if there is no prior established connection. Whenever the local relay receives a package the program extracts the prefix to determine if the prefix matches a function in the program or not.

For example, for the prediction we use the prefix: ***prediction/location***, so whenever the local relay receives a package with that prefix we start the named function for prediction. This function executes the ARIMA prediction model with the data from the specified location stored in the sensor datastore. When the predicted values have been generated they are sent back to the client in a content object. The execution time for the predicting values is fast enough which prevents the interest for timing out.

For the historical data, the same methodology is used as with predicted values except the prefix is different. The named function program listens to the relay's socket for prefixes that look like: ***historical/location***. Then by taking the looptime of the specific sensor the named function calculates how many values we should query from the datastore in order to receive the values from the past 24 hours.

In our system we the have ability to request a specific historical value, for example a value that was sensed in the office with the prefix and sequence number: ***/p/office/4b498c82/23***. When our local relay receives an interest for a specific value and does not have it in its cache it will trigger a function by having the prefix and sequence number query the mongoDB datastore for it and send it back as a content object.

# 8 Android application

## 8.1 Application idea

The goal for developing an Android application was to use the CCN network to display IoT data to the users in a useful way. We decided to make an app that uses sensors placed at different locations in Polacksbacken, and we focused on app ideas that could be useful to students at the university. We tried to find some problem or difficulty that students have that could possibly be made easier. The TI sensors used in this project have sensors for microhpone, temperature, humidity, light, pressure a gyroscope. We came up with the idea to try to help students find the least crowded lunch room at Polacksbacken. We theorized that we could use the sensor values, primarily the microphone to monitor sound data, to estimate how busy the different lunch rooms were. The general consensus what that the idea had the most promise and we felt that this app concept fulfilled the goal using the CCN network to display sensor data to users in a useful way.

### 8.1.1 Caveat

About a month into the app development, we discovered that we would probably not be able to use the microphone on the sensor because the microphone driver was not implemented for the Contiki operating system. In other words, we didn't have the software for our program to get the microphone values from the sensor hardware.

We thought that we might be able to use a combination of temperature, humidity, pressure, and light to determine the least crowed lunch room, but we had not tested the sensors in any of these places and it was unsure if these values would give a useful indication.

## 8.2 Design process

After agreeing on this app concept, we went through a design process. One obvious task was to design a way to communicate sensor values in a user friendly way. We were certain that just showing the value in an integer format wouldn't be best. We discussed using colors, integers from one to five, or some form of intuitive graphic art. We came to the final decision of

using five different smiley faces. We mocked up a few different layouts using the smileys, and also a few using colors to see how they would compare. We made one design layout using the names of the location and a single bar of color. We made a layout that had a map of Polacksbacken with different smileys over the locations of the lunch rooms. And we made a layout using the location names and the smileys next to the names. We presented the different designs in a meeting with Ericsson and SICS and they gave us the feedback that the layout with just the location names with the smileys next to them was best.

We also designed a graph layout to display more detailed data when the user clicks on the lunch room location.

Next we started gathering information related to Android design. We found documentation for the best practices for mobile and specifically Android development collected in a guide published on the Android developers website titled Material Design [38]. We found that one of the newest ways to design applications for Android was using CardViews which are containers used to display a small amount of information. The idea behind using CardViews is to keep the look and feel of the design consistent and clean in the app . Therefore, we decided to use a CardView to represent each lunch room in the app. Further in to using Cardviews, we implemented an expandable dropdown functionality to display more detailed information when the user clicks on the location, like the sensor reading values.

Further into the development process, we decided that we wanted to build a data analysis component into the project. This addition would provide the Android application with predicted values for sensor data. The idea behind this was that users would be able to use the app not only to see which lunch room was lest crowed at the current moment, but to also give an indication of what would happen in the immediate future based on past patterns. We incorporated this addition into the design of the graph in the dropdown of the CardView [**figure**]. We designed showing real values in dark blue, the predicted values in light blue and having a legend indicating that the colors were for past and predicated data.In this way we came to the final design.

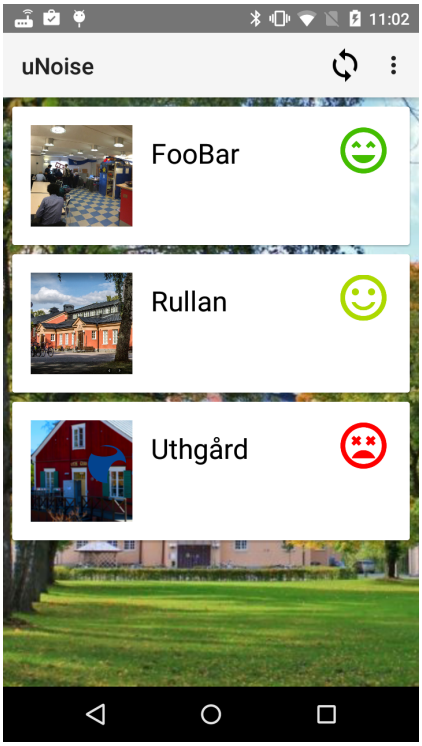



Figure 10: unoise application

## 8.3 Development process

We started off by reading and understanding the ccn-lite library we were going to use. There was and existing implementation of ccn-lite for android in their online repository [39] that we managed to run. This application was meant to act as a ccn-lite relay running through Bluetooth whose logs would be shown on the screen. We had little use of it but it got us on the right tracks on how to use a C library in an android application.

The next step for us was to import this project into Android Studio in order to have a better development environment. This proved difficult as the ccn-lite android project had been developed without it and had an outdated structure. To do that we had to first understand how the project was built and how the Java parts of the application worked with the C library. The project uses Android Native Development Kit (NDK) and Java Native Interface (JNI) [40], used respectively to permit building C files into an Android Java project and to provide access to their functions in the Java classes. The part that caused us the most trouble related to that was the fact that An-

droid Studio uses Gradle [41] to automate its building process. This caused automatic importation of the project into android studio to fail and a lot of manual configuration had to be made in order for the project to be built from the IDE.

After that we wanted to use and adapt the library to our needs, which were the ability to send interests and receive content packets. These functionalities had not been ported to android in the ccn-lite project yet, so we had to understand them thoroughly and before we could adapt them to the use in the android application. We created new files in the library whose names specified that they were used for the application (e.g. ccn-lite-peek-android.c). The communication behavior is as follows: A Java class called "AndroidPeekTask" has the responsibility to call the JNI "androidPeek" function in a separate thread to not block the UI while waiting for an answer. This JNI function then calls the native C function ccnLiteAndroidPeek that will create the Interest and send it the the phone's local IP address. Finally the response from received content packet is stripped and returned to the AndroidPeekTask through the same path and handled in the Application.

The specification required us to not only have one application running all the CCN functions but a separation of the tasks, on one hand the main application with a user interface making requests for data, and one the other a service running on the phone that would act as a relay transmitting the application's requests to our CCN network. Therefore we developed those deliverables in parallel, the application sends all its requests to the local IP address (127.0.0.1), those are then received by the service-relay that identifies them as CCN requests and transmits them via wifi to a defined gateway, an IP address that serves as an access point to the remote CCN network that we ran locally. The same path is followed backwards for the reception of the awaited content packets.

## 8.4 App functionalities

There are some functionalities for this android application that are described below:

- **Refresh Option:** The main purpose was to be able to quickly tell which place was the best one to go to, so at every refresh of the data the places are ordered from less to most noisy, with a big colored smiley face representing the level sound.
- **Showing the values of the sensors:** By selecting a location, a list

of all the sensors and their values is then displayed. The sensor values including ”light”, ”temperature” and ”humidity” are shown in figure 11.

- **Prediction Graph:** A prediction line graph that shows the estimated levels of light at the selected location for the next few hours. When a user clicks on one card related to a place, the graph shown in figure 11. The horizontal axis is time and the vertical axis is the value of a sensor. The historical data is shown by dark blue line and the prediction data is shown by bright blue dashed line in a filled drawing.

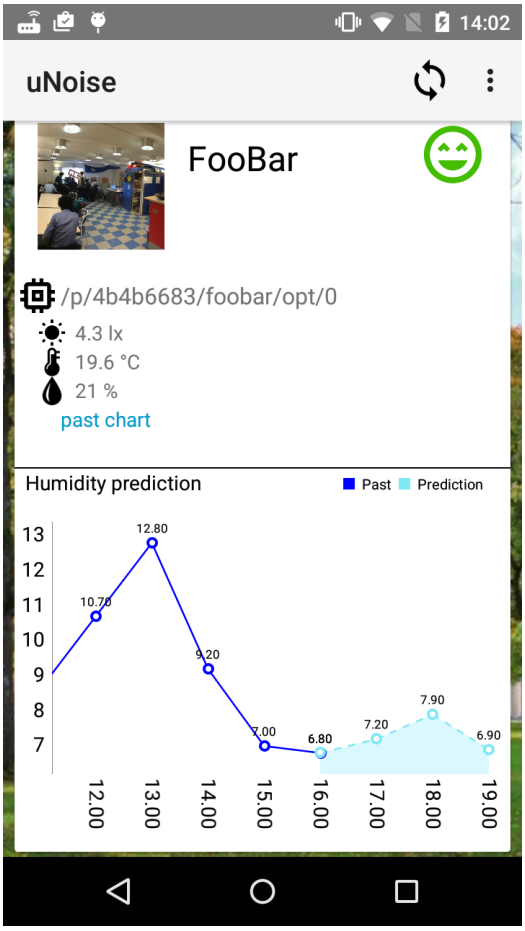



Figure 11: Sensor Values and Prediction Graph

- **Historical Graph:** A few of the historical data is shown simultaneously in the prediction line graph shown in figure 11.
- **Customizing Photos:** The places which are shown to users have

default photos that can be customized by the users, as shown in figure 12. Users can either take a photo or select one from the gallery and set it for a specific place. They can also delete a selected photo. The paths of photos are stored in a datastore in the android application.

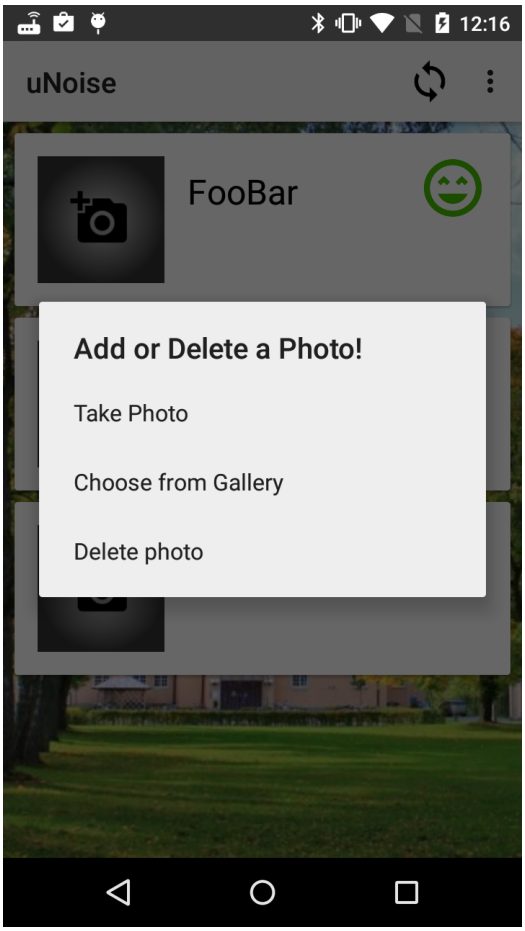



Figure 12: facility of customizing the location photos

- **Network Settings:** The application also has an option menu where network setting can be modified. The suite can be changed from ccnx2015 to ndn2013, the auto refreshing of the sensors can be toggled, and the relay service can be bypassed by choosing to request data directly from an IP address. This facility is shown in figure 13.

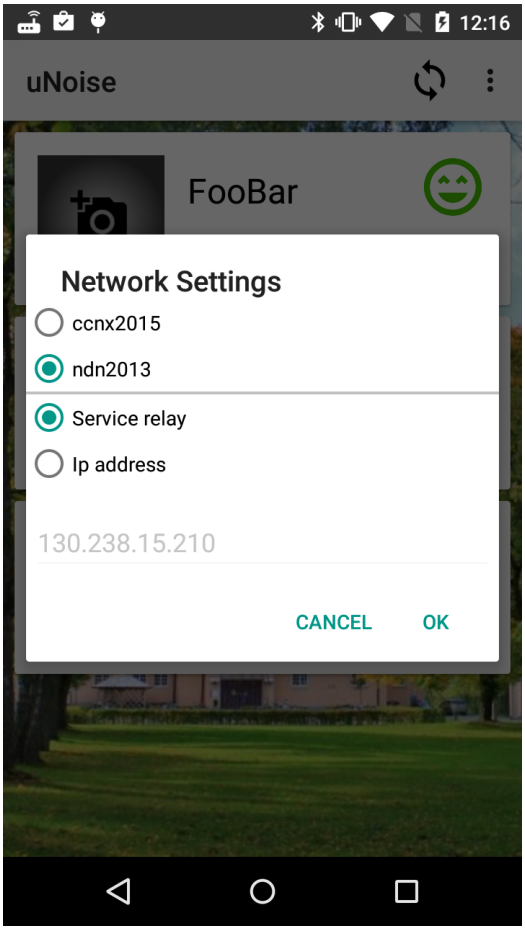



Figure 13: Network Settings

## 8.5 Predictions in the application

When a user clicks on any place, two requests for old data and predicted data will be sent to the server. Server predicts future data based on the ARIMA method that is described in the Prediction part. When the data is sent back, one graph includes some past values and predicted values are distinctly showed for the user.

## 8.6 Conclusion and future work

The Android application was a difficult part of the project in the way that we had a lot of trouble deciding what its purpose would be and how it would

look. The feedback we got about it along the project was constructive but kept pulling us in different directions, adding to the problem that we couldn't get noise data from the sensors.

In the end the application keeps the user interface of the original idea, finding quiet lunch places, and is operational for noise data. At the same time, we have added a lot of crude sensor information when a card is expanded in order to monitor which sensors the applications knows of and what data is gotten from them, as well as showing a prediction graph for groups of sensors.

Given more time we would have liked to focus on testing to see if a combination of other measures (temperature, pressure, humidity, etc.) would have allowed us to carry on with our lunch place idea without the use of noise. We would have also implemented more options for the advanced user to have more control and information about the sensors.

# 9 Testbed

In this section, the interaction between each part of the system will be presented more in depth. For this purpose, a small demo network has been created and presented below:

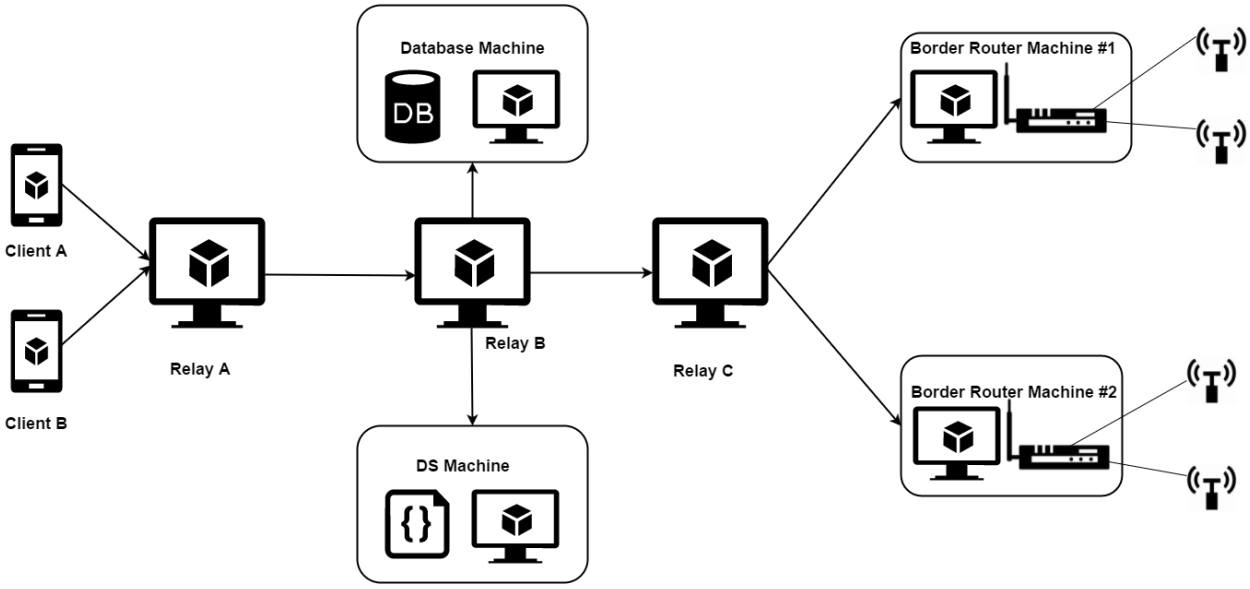



Figure 14: Final system architecture in our demo network.

The demo network above has been used during the testing stages of the project, it consists of two clients (Android devices) running our application, seven CCN Relays, a Directory Service (DS), a MongoDB datastore, two border routers with two sensors each. Each border router represents a location, in the demo network we have two locations: *office* and *confroom*. These locations are used in the naming scheme discussed in this report.

When each border router boots up, it searches for sensors in its radius. When it locates them, it creates routes to all of them using their IPv6 address. This process is repeated every 36 seconds. After the creation of the routes, the border routers transfer the Interest messages from the relays to the sensors and the Content Objects from the Sensors to the relays. Furthermore, the registration service running on the border router machine contacts the SDS Machine via UDP everytime a new sensor is located or a sensor is removed. This information is used by the SDS to update the status of the sensors in the datastore.

On startup the sensors set the channel and the 802.15.4 prefix that the sensors will use when transmitting according to their configuration file. Their IPv6 is automatically configured from the MAC address. The timeout is set to x seconds and when it expires (device wakes up) the main function requests values from the sensors. After the sensor values are produced, they are added to a content object and saved in the cache. When an interest message is received (tcpip event) the sensor checks its cache to see if it has the content object being requested. If it does it sends the content object back as a reply. At this stage, there is a connection between the border routers and their respective sensors.

The Datastore Machine is in charge of updating the MongoDB collection with real time sensor values. These values are requested directly from the sensors using the prefix */p/location/mac_id/seqno* where *location* is the sensor location given by the border router, *mac_id* is the last eight characters in the mac address of the sensor and *seqno* is the latest sequence number. The latter is calculated using the current epoch time subtracted with the initial epoch time when the sensor was first introduced into the system divided with the sensors looptime.

On the other end of the testbed, relay A is the bridge between the clients and all the other parts of the system. Therefore, the clients send interests to relay A who in turn responds with content objects back to the clients. A client can query five different types of data:

**Active sensor request**

On startup, the clients send an interest with the prefix */sds/sensors* and receives a list of active sensors. The list is generated by the DS machine by remotely querying the datastore for all sensor entries with the a-flag set to *True*. These sensors can be used to get sensor values using the other types of requests presented below. One limitation that we have in our current design is that if all sensor in a location goes down, the Android client is unable to request historical data for that location since the historical graph is showed for each sensor and if the sensor is down, the app will not show that sensor (see section 8).

**Historical data**

Historical data can be requested using the prefix */nfn/his/location*, where *location* is the area around a single border router (either office or conference room). The interest is propagated further to the relay running on the datastore Machine, through Relay B. When the interest reaches the datastore relay, a named function is triggered. A content object is then constructed with all data points gathered during the previous 24 hours. This content object is then sent back the same way that it came from, finally reaching the clients. Furthermore, the content object is saved in the cache of all the relays between the clients and the datastore machine (relay A and B in this case) for future use.

**Prediction data**

The clients are also able to request prediction data using a named function similar to the historical named function presented above. Clients create an interest with the prefix *nfn/pre/location* to get the predicted humidity values for the coming day in that location. The interest is propagated to the datastore Machine using the same path as above. The prediction named function is triggered, and a content object with predicted values for the coming day in that specific location is created and propagated back to the client. We chose to ignore the temperature and light values for the prediction since these values are pretty constant, and will not result in a good prediction. All relays on the path save the predicted values in the cache for future use. The combination of historical data and prediction data is used in the graph shown in the application.

**Latest sensor value**

Queries for the latest values can be made for a single sensor as well. This is done by sending interests with the following prefix */p/location/mac_id/seqno*.This interest is sent to:

1. The sensor with the given *mac_id*. This interest is propagated through the whole demo network from the client to one of the border routers. The content object is gathered from the cache of the sensor and sent back through the same relays as it was propagated through. This sensor value is saved in all the relays cache.
2. The datastore where a named function is triggered. The datastore is queried for that specific sensor, and the entry with sequence number *seqno* is returned in a content object.

The first reply received by the client is used and the other is discarded. The second content object is discarded since the PIT entry for that prefix will be removed when the first reply is received.

**Specific sensor value**

Querying a specific sensor value in time is similar to querying the latest sensor value, the prefix used in this case is also */p/location/mac_id/seqno*. The difference is that a request for a specific sensor value is only sent to the datastore, since the sensor do not store old values.

**Performance Evaluation**

We have not done any scalability and performance benchmark tests due to lack of time. However we have had the system up and running with the setup described above for at least a weekend.

The network is able to handle requests from both clients in the testbed and receive all the different types of data described above. The response time is pretty much instant for all types of requests.

One issue that we have encountered is that the border routers tend to crash if a new sensor is introduced into the system, due to serial connection problems, other than that, the system is stable.

## 9.1 Limitations of the system

**Application:** Our starting point for the project was to build an application to analyze noise in public areas. For this we used the TI CC2650 SensorTag. It provides several different readings that are made available through RTOS (native OS) by Texas Instruments. Since we were using Contiki OS, we were not able to use the full potential of the device; more specifically, the microphone which was one of the main sensors that we were interested in.

This forced us to change the initial application to use only the available sensor readings; that is, light, temperature, humidity, etc.

**Network:** One limitation we faced in relation to networking is that a content object which is already cached will be considered valid even if a new content object with the same name has been created "further away" in the network. For example, relay 1 has a content object with the prefix */p/office/12345678/6*, where 6 represents the sequence number. In the case that the sensor reboots or re-registers in the office before that content object is evicted from the cache, relay 1 will reply with the old content object in an incoming interest, when it should have propagated the interest to relay 2 which is connected to the border router - sensors.

Discovery function described in section 4.2 uses IP broadcasting for discovering all the direct neighbors. This enables the discovery function to find all the direct neighbors but limits the function to only that network segment making it not possible to use two or more different networks while relying only to the discovery function.

**Network underlay:** The network we're using is dependent on having an underlying running IP network that allows relays to be able to connect to each other. We've tried to make the network as independent of IP as possible, with a few exceptions. Exceptions include the connections between the DS and relays and the connections between the Registry Service and the DS, where standard UDP sockets are used.

The realization of this system would have been possible using only Ethernet as the underlying protocol. This would have forced us to use alternative means of contacting the DS, as well as updating the FIB tables of the relays. We experimented with a technique which allowed the relay running on the DS machine to advertise it's location to its immediate neighbors, which in turn then did the same thing to its neighbors, and so on. This proved to be too complicated to get up and running correctly in the time we had though and the idea was scrapped. We did use a variation of this mechanism though, for allowing the clients to be able to send interests to the DS through our network. Once the relays is registered in the DS, it gets an update from it telling it which face to send interests with the prefix /sds. The DS calculates this path using the same algorithm as the one used to populate the FIB tables to be able to reach the border-routers.

**Sensors:** Regarding the sensors and the fact that they have very limited resources (e.g., buffer-size), we had to be conservative on their use. Due to these buffer-size limitations we used the shortest possible prefixes. This

affected the interest messages when receiving but also content objects when producing. Initially we wanted to create and transmit content objects in SenML format from the sensor, which would simplify the process for neighboring systems but due to limited buffer size that was impossible. The sensor memory is also limited to 15 content objects, which depending on the production interval can represent values from a few minutes up to several hours, but not values from previous days or weeks. The CC2650 sensors running Contiki OS can not use Texas Instruments libraries made for TIs' native OS (RTOS), since they are only made for TIs' Cloud Composer Development Environment. The wireless transmission range for the sensors is about 100 meters. In Contiki OS we can not use IPv6 and RIME stack simultaneously. This limits the developer to use only one of these two protocols at a time. The memory allocation on the sensor is being done statically by explicitly declaring the size of the arrays that will be used. These arrays represent buffer spaces or other values that need to be allocated. C language built-ins such as *malloc* do not work on the sensor and specific *malloc* commands for Contiki OS are fairly complicated to use. Therefore, dynamic memory allocation in the sensor becomes rather complicated. However, because the buffer space is explicit and the structure and size of the content objects allows static allocation, the above limitation does not constitute a problem.

**Border Router:** While performing tests we noticed that multiple border routers can interfere with each other and function abnormally. For this reason each *border router* is located outside every other routers' radius. Border Routers can crash unexpectedly for no obvious reason, which affects the systems' integrity.

**DS:** The DS works as an controller in our system, making all forwarding decisions in the network. Unfortunately, the routing mechanism used is not the most scalable one, although is sufficient to handle the size of the network we've implemented. As more and more sensors and border routers are added to the network, the FIB tables of the relays quickly grows large, possibly resulting in crashes once it becomes to big. Something that would have been very interesting to implement in our system is a more advanced and scalable routing protocol, such as Dynamically Controlled Routing (DCR). Although, due to limitations set by the group and product owner during the startup meeting of this project, we decided to put our focus on getting a functional network up and running rather than devoting time on the development of an scalable routing algorithm.

The DS is only running at one machine in the network, which means that if this machine would go down all control of the forwarding plane would be

lost. To solve this, redundancy would need to be implemented by running a backup DS instance in another part of the network which would take over if the primary one would go down.

The behavior of the DS is dependent on a relay being shut down in a controlled manner. In the case that the relay crashes, or the computer that is running it is shut down, the entry remains in the DS and the network will treat it as if it was online. To handle this, a heartbeat system needs to be implemented. This could be done by sending control-messages to each registered relay in the network at regular intervals and awaiting a response for a specified duration of time. In the case that the relay doesn't respond, it would be treated as if it had sent the Status - Off message and be removed from the network. Due to time constraints during the project, this mechanism has not been implemented.

DS maps the network by receiving broadcast message from every relay that boots up in the network and registers the IP address for that relay. This means that only the relays that are in the same local network can reach the DS. To get the mapping work on every kind of network topology a proper routing protocol should be implemented. While running out of time it was decided to be left for the future work.

Another limitation in this project is the process of updating the Android client with the available sensors in the system. The current implementation builds a JSON object containing the complete list of all active sensors which have data that can be collected. A more scalable approach would be to allow the client to only receive a subset of sensors, such as for a specific location.

**Prediction:** We implemented the prediction script as a named function, but every time this function is invoked we need to find optimal parameters by using ACF and PACF plots which is not feasible. So If we use the default parameter setting which is (2,1,2), the prediction will fail and the result will converge. This is an obstacle we can not overcome for now and a limitation for this project.

# 10 Conclusion

Our implementation of an ICN network works to the extent of our expectations connecting physical sensors, android applications, a datastore and a second similar network architecture made by the GreenIoT project based on

the MQTT protocol. All the different parts of the system we built are able to communicate with each other using almost exclusively interests and content packets.

Our main points of interest at this stage regarding the network are the remaining dependence on IP, and its scalability. It is still dependent on IP for some tasks and we think it is possible to further improve it in that regard. Regarding the network's scalability, a sequence of tests would help plan how it would react under a heavier load of traffic and in a wider network.

Our implementation is split into multiple repos that can all be found on Bitbucket [42][43][44][45].

# 11 Future work

## 11.1 Datastore

Handling large amount of readings for several sensors over several days is a challenging task. We addressed those challenges to the best of our abilities, but we cannot yet claim that we can handle big-data. A future improvement to that would be to use a distributed datastore system.

## 11.2 Sensors improvements

One improvement that could be done on the sensors would be to store pending requests for the future. Since the sequence number represents time, a user could be able to request a specific value for a time in the future. The sensor would store the request and reply to it whenever it has the requested value. This improvement would add extra complexity and should be handled carefully since the sensor can store only a limited amount of requests since the resources are very limited. A second improvement would be to include microphone values in the system whenever the drivers are available for Contiki OS. CCN Lite implementation uses various encodings, but the main two are *ndn2013* and *ccnx2015*. Ideally we wanted to use *ccnx2015* but there were unexpected error coming up in the sensors that we could not solve in time, so we had to use *ndn2013*. However, this limitation did not affect the systems' functionality. So as future work we could include solving this issue. Every time the sensor receives a CCN interest message it has to check in its cache whether or not it contains that content object. We have noticed that

multiple checks in the cache can force the sensor to crash. Since we know the current sequence number of the content objects and the number of content objects can be cached so we can calculate if the content object is in the cache before we check the cache.

## 11.3 DS running on multiple relays

One possible improvement of the system would be to run multiple DS simultaneously on multiple relays. This way the system would not be dependent on one single relay - one single point of failure. However, this solution would introduce significant complexity to the system since the now distributed DS should be identical on relays, which would probably require locks and acknowledgments. This process of having distributed DS might make information more easily accessible but the system would have to update all DS before every modification, so we can not assume that the performance would be improved. Of course for a large scale system distributed DS would be necessary.

# References


[1] *Information-Centric Networking Research Group*. https://irtf.org/icnrg/. Retrieved: 2016-09-27.

[2] Daniel Burrus. *The Internet of Things Is Far Bigger Than Anyone Realizes*. https://www.wired.com/insights/2014/11/the-internet-of-things-bigger/. Retrieved: 2016-09-27.

[3] *Bristol is Open*. http://www.bristolisopen.com/. Retrieved: 2016-12-12.

[4] *SmartSantander*. http://www.smartsantander.eu/. Retrieved: 2016-12-12.

[5] Adeel M. Malik. *Design, Development and Deployment of an Information-Centric Networking based solution for the Internet-of-Things*. http://www.it.uu.se/edu/course/homepage/projektDV/ht16/specification.pdf/. Retrieved: 2016-09-27.

[6] parc. *content-centric networking*. https://www.parc.com/work/focus-area/content-centric-networking/. Retrieved: 2016-09-27.

[7] *Project CS 2016/2017 (30 credits)*. http://www.uu.se/en/admissions/master/selma/Kurser/?kKod=1DT054&typ=1/. Retrieved: 2016-12-12.

[8] SICS. *GREENIOT: AN ENERGY-EFFICIENT IOT PLATFORM FOR OPEN DATA AND SUSTAINABLE DEVELOPMENT*. https://www.sics.se/projects/greeniot-an-energy-efficient-iot-platform-for-open-data-and-sustainable-development/. Retrieved: 2016-09-27.

[9] Uppsala Kommun. *Tio miljoner kronor till innovationsdriven stadsutveckling*. https://www.uppsala.se/organisation-och-styrning/nyheter-och-pressmeddelanden/tio-miljoner-kronor-till-innovationsdriven-stadsutveckling/. Retrieved: 2016-09-27.

[10] *CCN-lite Github Project*. https://github.com/cn-uofbasel/ccn-lite. Retrieved: 2016-12-12.

[11] *CCN-lite*. http://www.ccn-lite.net/. Retrieved: 2016-10-04.

[12] *ccn-lite github*. https://github.com/cn-uofbasel/ccn-lite/. Retrieved: 2016-10-04.

[13] *How does MongoDB work?* https://www.mongodb.com/what-is-mongodb/. Retrieved: 2016-10-04.

[14] *What is Contiki?* http://www.contiki-os.org/. Retrieved: 2016-10-04.

[15] Charith Perera et al. *Sensing as a service model for smart cities supported by Internet of Things*. 2014. DOI: 10.1002/ett.2704. URL: http://dx.doi.org/10.1002/ett.2704.
[16] A. Zanella et al. *Internet of Things for Smart Cities*. Feb. 2014. DOI: 10.1109/JIOT.2014.2306328.
[17] Soon Y Oh, Davide Lau, and Mario Gerla. *Content centric networking in tactical and emergency manets*. IEEE, 2010.
[18] Jaebeom Kim, Daewook Shin, and Young-Bae Ko. *TOP-CCN: topology aware content centric networking for mobile ad hoc networks*. IEEE, 2013.
[19] named-function. *Named functions*. http://www.named-function.net/. Retrieved: 2016-09-27.
[20] Open Networking Foundation. *Software-Defined Networking (SDN) Definition*. https://www.opennetworking.org/sdn-resources/sdn-definition/. Retrieved: 2016-09-27.
[21] Alan Ott. *Wireless Networking with IEEE 802.15.4 and 6LoWPAN*. http://elinux.org/images/7/71/Wireless_Networking_with_IEEE_802.15.4_and_6LoWPAN.pdf. 2012.
[22] Tsvetko Tsvetkov. *RPL: IPv6 Routing Protocol for Low Power and Lossy Networks*. https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2011-07-1/NET-2011-07-1_09.pdf. 2011.
[23] *Contiki CNNLite*. 40bc35a9faf34fcf661a4074f6bb0bcbbbdb6e80.
[24] University of Basel. *CCN-Lite Release 0.3.0*. https://github.com/cn-uofbasel/ccn-lite/releases/tag/0.3.0. 2015.
[25] Paweł Rozynek. *What is SenML?* http://blog.rapifire.com/2015/12/21/introduction-to-senml/. Retrieved: 2016-09-27.
[26] SICS. *sparrow*. 26779a02d76aca02f96ef6d9f45f0cd73eb83c4b.
[27] *Flask*. http://flask.pocoo.org/. Retrieved: 2016-12-12.
[28] *Eclipe Paho MQTT Embedded MQTT C/C++ Client Libraries*. https://eclipse.org/paho/clients/c/embedded/. Retrieved: 2016-10-14.
[29] *MongoDB C Driver*. http://mongoc.org/. Retrieved: 2016-10-14.
[30] *Paho MQTT Python Library*. https://pypi.python.org/pypi/paho-mqtt/. Retrieved: 2016-12-12.
[31] G. P. Nason. *stationarity*. https://pdfs.semanticscholar.org/0f08/bcca67b3db328edfa5d3f48331dc71d8789e.pdf.
[32] *Autoregressive Models*. https://onlinecourses.science.psu.edu/stat501/node/358. Retrieved: 2016-11-29.
[33] Gidon Eshel. *parameterequation*. http://www-stat.wharton.upenn.edu/~steele/Courses/956/Resource/YWSourceFiles/YW-Eshel.pdf.

[34] *Moving Average Models*. https://onlinecourses.science.psu.edu/stat510/node/48. Retrieved: 2016-11-29.

[35] *A Complete Tutorial on Time Series Modeling in R*. https://www.analyticsvidhya.com/blog/2015/12/complete-tutorial-time-series-modeling/. Retrieved: 2016-11-29.

[36] Souhaib Ben Taieb Gianluca Bontempi and Yann-Ael Le Borgne. *Machine Learning Strategies for Time Series Forecasting*.

[37] Ratnadip Adhikari. *an introductory study on Time Series modeling and forecasting*.

[38] *Material Design*. https://developer.android.com/design/material/index.html. Retrieved: 2016-12-15.

[39] *CCN-lite Android Project*. https://github.com/cn-uofbasel/ccn-lite/tree/master/src/android. Retrieved: 2016-12-16.

[40] *NDK and JNI*. https://developer.android.com/ndk/guides/concepts.html. Retrieved: 2016-12-14.

[41] *Gradle*. https://developer.android.com/studio/build/index.html. Retrieved: 2016-12-18.

[42] *Basic CCN App Repository*. https://bitbucket.org/MaxWijnbladh/saviorapp. Retrieved: 2017-01-11.

[43] *UNoise CCN App Repository*. https://bitbucket.org/Aranor/ccn-lite-android. Retrieved: 2017-01-11.

[44] *CCN Lite Implementation Repository*. https://bitbucket.org/MaxWijnbladh/ccn_lite_greeniot. Retrieved: 2017-01-11.

[45] *CCN Sensor code Repository*. https://bitbucket.org/theodosismalatestas/sensorcode. Retrieved: 2017-01-11.