**졸업 프로젝트**

**구현 최종보고서**

-아두이노와 웹서버를 이용한 스마트홈 오토메이션-

고지혜 201311193

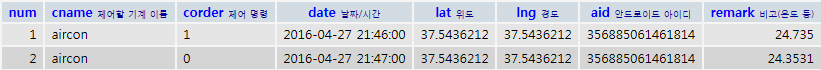
김예찬 201311200

**1. 디비 설계**

웹서버의 데이터 베이스는 다음과 같은 테이블들로 이루어져 있다.

스마트폰에서 보낸 명령들을 저장할 control테이블이다. Primary key인 num과 제어할 기계 이름인 cname, 제어 명령(기계를 On할 것인지, Off할 것인지 등)을 저장하는 corder, 날짜와 시간을 저장하는 date, 위도와 경도인 lat, lng, 명령을 보낸 스마트폰의 아이디를 저장할 aid칼럼이 있다.

사용자의 패턴을 분석하려면 위의 기본적인 필드들 외에 부가적인 정보가 필요하다. 본 프로젝트에서는 패턴을 통한 자동 제어 서비스로 에어컨 제어 기능을 제공하는데, 사용자가 어떤 온도일 때 에어컨을 제어했는지의 정보가 필요하다. 따라서 remark라는 비고 필드를 추가하였다. 에어컨의 경우엔 이 필드에 온도를 저장한다.

****

<control 테이블>

현재 웹 서버에 등록된 안드로이드 기기들을 저장하는 registered 테이블이다. 아이디를 저장하는 aid칼럼이 있고 사용자의 이름, GCM을 받기 위한 instance id token값과 승인 여부, 사용자의 집 위치의 위도, 경도의 칼럼이 있다.

****

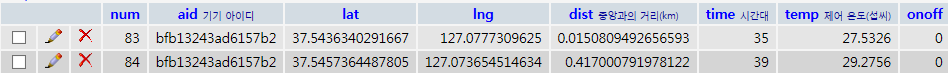
<registered 테이블>

사물들의 상태를 저장할 status테이블이다. Primary key인 num과 상태를 갖는 사물의 이름인 sname, 사물의 상태(0이면 off, 1이면 on 등)인 sstatus, 해당 상태로 바뀐 시간인 date가 있다.

****

<status 테이블>

사용자의 패턴을 저장할 pattern 테이블이다. 해당 패턴을 가진 사용자의 기기 아이디인 aid와 제어가 이루어지는 장소인 lat와 lng, 그 장소에서 집과의 거리인 dist, 제어가 이루어지는 시간대인 time, 제어가 이루어지는 온도가 있다.



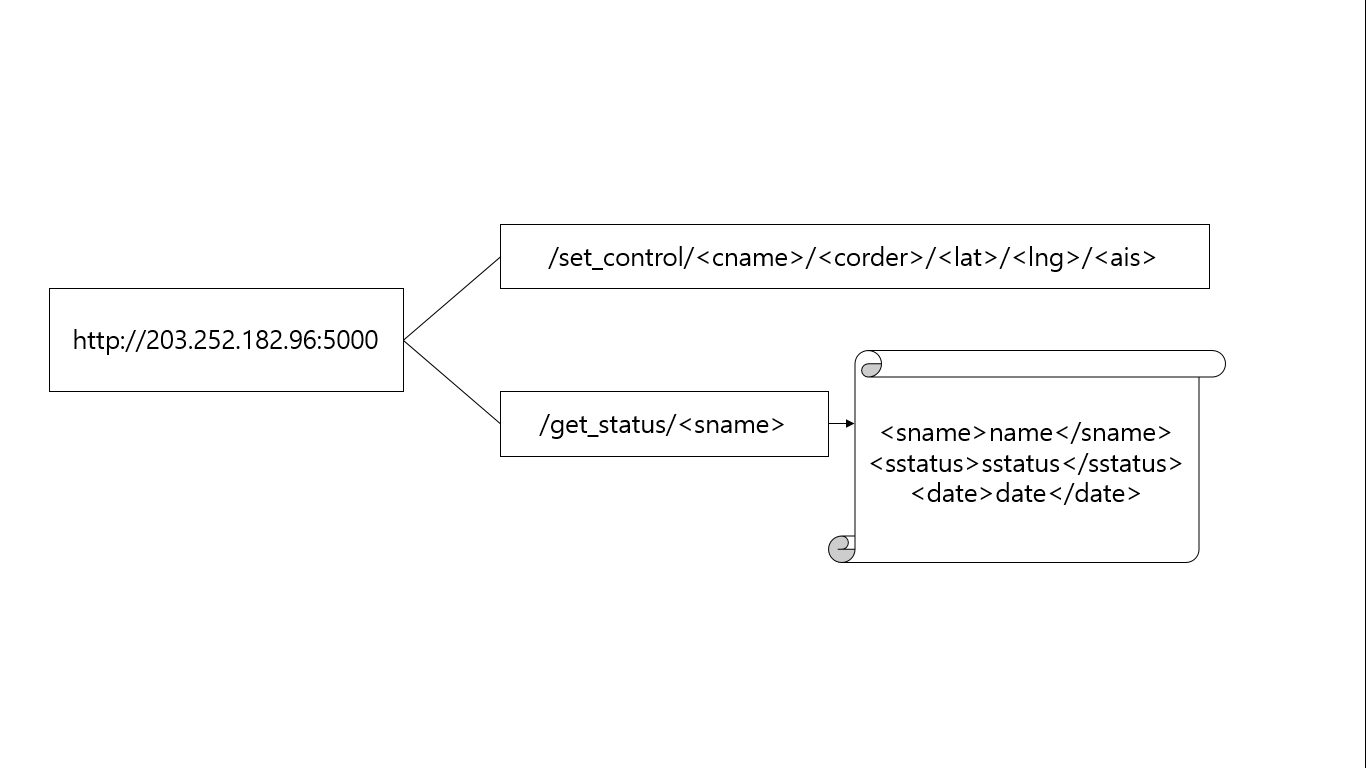
<pattern 테이블>

**2. 파이썬 웹 서버 구축**

효율적으로 패턴인식 알고리즘을 구현하기 위해 파이썬을 사용하는 것이 좋다는 점을 알게 되었다. 따라서 기존에 apmsetup을 사용하여 아파치 웹서버를 구축하였던 것을 파이썬을 통해 구축하기로 하였다.

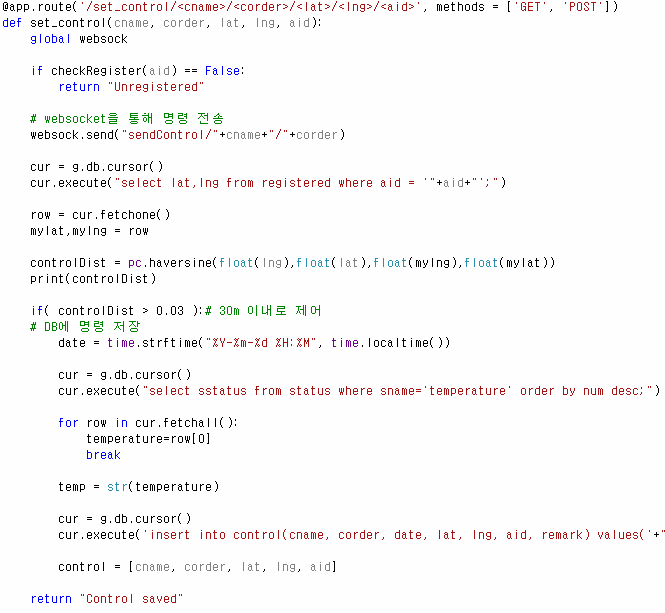
웹서버 구축을 위해 사용한 파이썬 라이브러리는 Flask이다. Flask는 파이썬 기반의 마이크로 웹 프레임워크이다. 기존의 php파일들 없이도 동적 라우팅과 탬플릿 렌더링을 통해 예전의 웹서버와 동일하게 동작하는 웹서버를 구현하였다. 또한 pymysql을 사용하여 기존의 DB에 연결하였다.

Flask에서는 URL을 호출하면 Route Decorator를 통해 해당 URL에 맞는 동작을 하도록 할 수 있고, 동적인 변수를 사용할 수 있다. 아래가 바로 라우팅 목록이다.



1) @app.route('/set\_control/<cname>/<corder>/<lat>/<lng>/<aid>',methods=['GET','POST'])

안드로이드에서 보낸 명령을 DB에 저장한다. control테이블에 GET방식으로 받은 cname, corder, lat, lng, aid와 현재 시간을 생성하여 넣은 date를 삽입한다. 그 후 해당 명령을 웹소켓을 통해 중앙 아두이노로 데이터를 보낸다. (웹소켓 부분에서 자세히 설명)



<웹 서버 사물 제어 부분 코드>

2) @app.route('/get\_status/<sname>')

안드로이드에서 물건들의 상태를 알고 싶을 때 호출하므로, DB에서 해당 물건(sname)의 가장 최근 상태를 가져와 출력한다. 안드로이드에서는 출력된 텍스트를 파싱하여 상태를 알게 된다.

웹서버가 어느 IP주소에서, 어느 포트로 동작할지 정해주고, 해당 IP주소가 외부에서 볼 수 있도록 하기 위해 포트 포워딩을 해주었다.





<웹 서버 사물들의 상태 보기 부분 코드>

**3. 웹소켓**

기존에 구현한 것에서 문제가 되었던 것이 pulling방식이었다. 중앙 아두이노에서 서버로 2초정도의 딜레이를 두고 계속해서 리퀘스트를 보내는 방식으로 명령을 가져왔었다. 따라서 불필요한 연결이 많았고 배터리도 빨리 닳았다.

HTTP프로토콜에서는 이렇게 요청이 있어야 웹 서버가 응답을 보내는 형식이기 때문에 TCP통신과 같이 서로 주고받는 통신이 어렵다. 이 점을 보완하기 위해 찾아낸 방식이 long polling방식과과 HTML5부터 지원되는 WebSocket방식이 있었는데, long polling은 지연시간이 있고 복잡하다는 단점이 있었고, WebSocket은 호환성 문제가 있었다. 최근에는 WebSocket을 지원하는 브라우저도 많고, 아두이노역시 WebSocket을 지원하기 때문에 WebSocket을 사용하기로 하였다.

1차로 구현해보았을 때에는 중앙 아두이노가 웹소켓 서버가 되고, 웹 서버가 웹소켓 클라이언트가 되어 안드로이드에서 명령을 보낼 때 호출되는 html페이지의 자바크스립트에서 웹소켓을 호출될 때마다 연결하여 명령을 전송하도록 구현하였다.

웹서버가 웹소켓 클라이언트가 되는 것도 이상하고, 연결이 유지되지 않는다는 점, 아두이노 ip에 외부에서 연결할 수 있도록 포트포워딩을 해줘야 한다는 점 등 문제점이 많았다. 따라서 웹 서버를 서버로 하고, 중앙 아두이노를 클라이언트로 하고 연결이 유지되는 시스템으로 다시 구현하였다.

3.1. 웹 서버측

웹소켓을 통해 통신하기 위해서는 웹소켓 통신 프로토콜을 따라야한다. 웹소켓은 다른 HTTP 요청과 마찬가지로 80번 포트를 통해 웹 서버에 연결한다. 다른 점은 Upgrade 헤더를 추가하여 웹 서버에 요청한다.

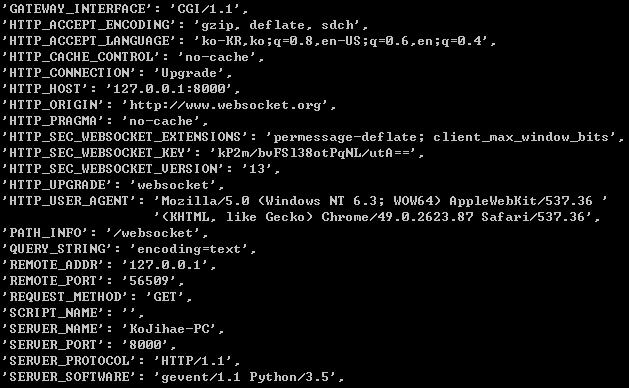


클라이언트는 "Upgrade: WebSocket" 헤더 등과 함께 랜덤 하게 생성한 키를 서버에 보낸다. 웹 서버는 이 키를 바탕으로 암호화와 인코딩 과정을 거쳐 토큰을 생성한 후 클라이언트에 돌려준다. 이런 과정으로 WebSocket 핸드쉐이킹이 이루어진다.

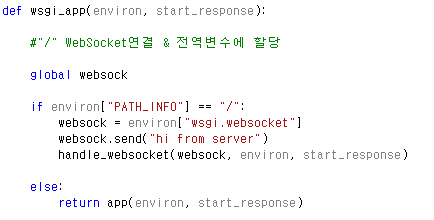
그 뒤 Protocol Overhead 방식으로 웹 서버와 클라이언트가 데이터를 주고 받는다. 이 방식은 여러 TCP 커넥션을 생성하지 않고 하나의 80번 포트 TCP 커넥션을 이용하고, 별도의 헤더 등으로 논리적인 데이터 흐름 단위를 이용하여 여러 개의 연결을 맺는 효과를 내는 방식이다. 스마트홈 서비스에서는 서버와 중앙 아두이노와만 연결을 맺으므로 크게 상관이 없을 것으로 보인다.

파이썬에서 Flask와 함께 구동될 수 있는 웹소켓 서버를 구현하기 위해 WSGI인 gevent.pywsgi와 geventwebsocket라이브러리를 사용하였다. 이 라이브러리가 데이터 타입 등으로 인해 Python3에서 제대로 사용될 수 없었기 때문에 해당 라이브러리의 파일들을 수정하여 사용하였다.

서버를 구동시키면 클라이언트로부터 받는 요청은 다음과 같다.



웹소켓 클라이언트(아두이노)가 웹소켓에 연결을 시도하면 먼저 전역 변수에 웹소켓을 저장하고 웹소켓에서 일어나는 이벤트를 관장하는 handle\_websocket함수를 호출한다.



<서버 측 웹소켓 연결 부분 코드>

3.2. 중앙 아두이노측

중앙 아두이노는 인터넷에 연결되어 있어야 하기 때문에 이더넷 쉴드를 적층시켰다. 이더넷 쉴드는 아두이노 보드를 인터넷에 연결할 수 있게 해준다.



중앙 아두이노는 웹소켓 클라이언트가 된다. 아두이노 웹소켓 라이브러리는 꽤 있었지만, 공식적으로 제공하는 것은 없었다. 또한 최근의 웹소켓 프로토콜에 맞추어 최근까지 수정, 보완하는 웹소켓 라이브러리는 없었다. 따라서 일정부분 수정하여 사용하였다.

아두이노가 처음 시작할 때 호출되는 함수인 setup()에서 이더넷 연결을 시작하고, 웹소켓 연결을 시작하였다. 웹소켓에서 이벤트가 일어날 때 호출할 함수도 등록하였다.

아두이노의 기본 함수인 loop()에서는 서브 아두이노가 보내는 데이터가 있으면 받는 함수를 호출하고, 웹소켓을 유지하는 webSocket.loop()를 호출한다.

웹소켓 클라이언트에서는 웹소켓 이벤트에 따른 함수를 정의할 수 있다. 웹소켓이 연결 되었을 때, 웹소켓 연결이 끊어졌을 때, 웹소켓으로부터 데이터가 들어왔을 때이다. 웹소켓으로부터 데이터가 들어왔을 때 들어온 데이터는 사물 제어 명령이므로 바로 Xbee를 통해 명령을 서브아두이노로 송신하도록 하였다.

웹소켓 연결이 간혹 끊길 때가 있으므로 끊기는 주기를 평균 내어 그 주기만큼 중앙 아두이노에서 웹 서버로 끊기지 않도록 메시지를 보내도록 하였다.

**4. 메시지 프로토콜**

웹소켓을 통해 주고 받는 메시지가 많기 때문에 메시지 프로토콜이 필요하다.

4.1 사물 제어 명령

사용자가 URL을 통해 사물 제어 명령을 보내면 웹 서버는 웹소켓을 통해 중앙 아두이노로 명령을 전송한다. 보내는 메시지의 형식은 “sendControl/cname/corder”이다. (cname – 사물의 이름, corder – 명령의 내용) 메시지를 받은 중앙 아두이노는 서브 아두이노로 Xbee통신을 통해 메시지를 그대로 전송한다.

서브 아두이노는 받은 메시지를 “/”를 기준으로 분리하여 어느 메시지인지 구분한다. sendControl메시지인 경우 사물의 상태를 바꾼다. 이 때 받은 명령의 내용이 이미 사물의 상태라면 상태를 바꾸지 않는다. 상태를 바꾸게 된 경우 바뀐 상태를 Xbee통신을 통해 중앙 아두이노로 전송한다. (상태 정보 부분에서 자세히 설명)

4.2 상태 요청

사용자가 사물의 상태를 보기 위해 URL을 통해 상태 요청을 보내면 웹 서버는 어느 사물의 상태를 요청하였는지 본다. 사물의 상태는 에어컨이나 조명처럼 On/Off만 존재하는 상태가 있고, 실시간성이 중요한 온도가 있다. 따라서 전자의 상태를 요청한 경우 단순히 DB에서 가장 최근의 상태를 가져오고 후자의 경우엔 실시간 상태를 가져와야 하기 때문에 웹소켓을 통해 중앙 아두이노로 상태 정보 요청 메시지를 보낸다.

보내는 메시지의 형식은 “requestStatus/temperature”이다. 그 후 웹 서버는 상태를 기다리고 있다는 플래그 값을 True로 바꾸어 현재의 온도 값이 들어오기를 기다린다. 메시지를 받은 중앙 아두이노는 서브 아두이노로 메시지를 전송한다. 서브 아두이노는 받은 메시지를 구분하여 requestStatus인 경우 현재 상태를 Xbee통신을 통해 중앙 아두이노로 전송한다.

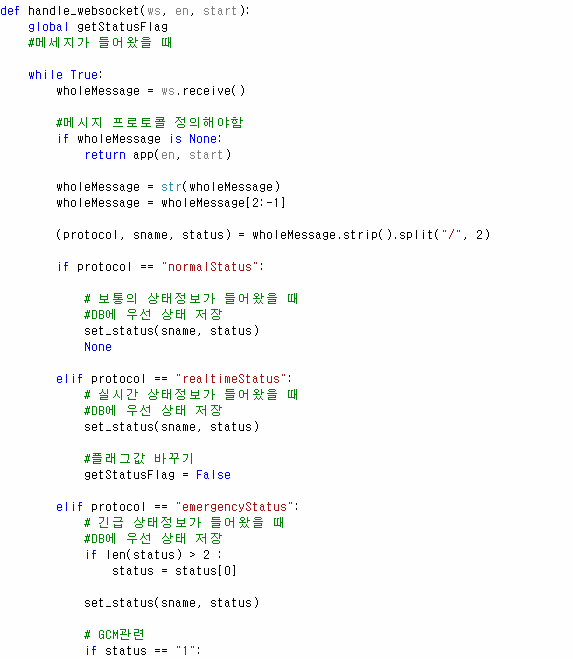
4.3 상태 정보

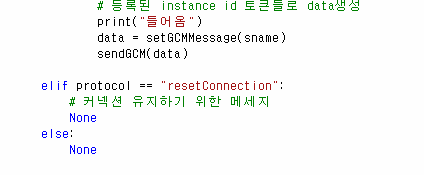
서브 아두이노(사물)가 보내는 상태 정보는 세가지로 나뉜다. 에어컨이나 조명의 On/Off 여부인 보통의 상태, 실시간성이 중요한 온도, 창문의 여닫힘 여부인 긴급성이 있는 상태가 있다.

보통의 상태정보를 보낼 때의 메시지 형식은 “normalStatus/sname/sstatus”이다. (sname – 사물의 이름, sstatus – 사물의 상태) 웹 서버에서 웹소켓을 통해 메시지가 들어오면 “/”를 기준으로 분리하여 어느 상태정보인지 구분한다. normalStatus인 경우 단순히 DB에 상태를 저장한다.

실시간 상태정보를 보낼 때의 메시지 형식은 “realtimeStatus/sname/sstatus”이다. 웹 서버에서 메시지를 구분하여 realtimeStatus인 경우 우선 DB에 상태를 저장한다. 실시간 상태정보가 왔다는 것은 상태를 요청한 후 그 정보가 오길 기다리고 있다는 뜻이므로 상태를 기다리고 있다는 플래그 값을 False로 바꾼다. 그 후 DB에서 상태를 가져와 XML형태로 안드로이드 기기로 보내고 요청을 마무리한다.

긴급 상태정보를 보낼 때의 메시지 형식은 “emergencyStatus/sname/sstatus”이다. 웹 서버에서 메시지를 구분하여 emergencyStatus인 경우 우선 DB에 상태를 저장한다. 긴급 상태정보의 경우 사용자가 요청하지 않았어도 알려야 하므로 GCM을 이용하여 사용자에게 푸시알림을 보낸다.

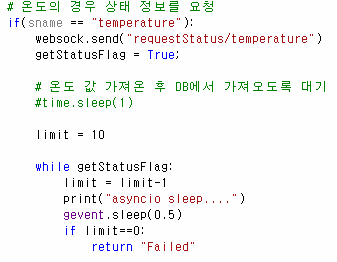




<웹 서버 측 메시지 프로토콜에 따라 수행하는 부분 코드>

기존에는 실시간 상태를 받아올 때(realtimeStatus) 서브 아두이노에서 온도 상태를 보내고, 그 상태를 중앙 아두이노를 통해 웹 서버에서 받기 까지 웹 서버에서는 새로운 페이지로 redirect하였다. 새로운 스레드를 만들어 온도를 받았는지 체크하는 작업을 수행하려 했지만 플라스크에서는 새로운 스레드를 만들어도 계속해서 synchronous한 작업이 이루어졌기 때문에 Sleep를 잠깐씩 하면서 redirect하였다.

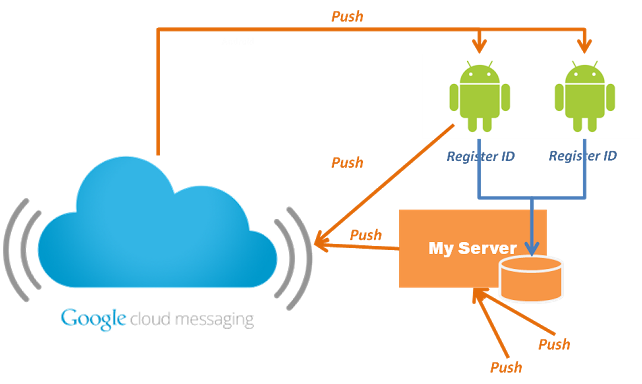
하지만 이 때문에 온도를 받아오는 데에 시간이 8초쯤 걸리게 되었다. 이 시간을 줄이기 위해 더 연구 해보니 asynchronous한 sleep를 gevent에서 제공한다는 것을 알게 되었다. 이를 통해 온도를 받아오는 시간을 5초 이하로 줄일 수 있었다.



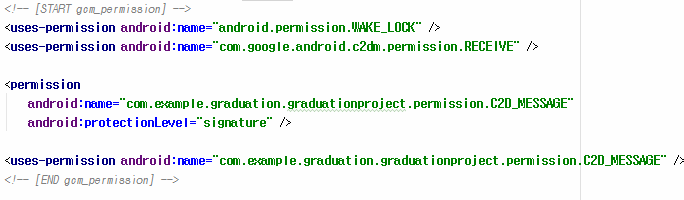
<온도 요청하는 부분>

**5. GCM**

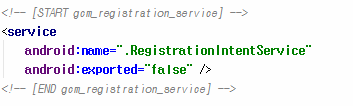
GCM은 Google Cloud Message로, 푸시알림을 보낼 때 사용한다. 긴급 상태정보의 경우 사용자가 요청하지 않았어도 알려야 하므로 GCM을 이용하여 사용자에게 푸시 알림을 보낸다.



먼저 구글 개발자 페이지에서 GCM을 사용할 앱을 등록한 후 서버 API키를 얻었다. 그 후 configuration file인 google-services.json을 다운받아 앱 디렉토리 안에 넣었다. Google Play Services를 설정하고 manifest에서 각종 퍼미션과 서비스, 리스너를 등록해주었다.







<AndroidManifest.xml의 일부>

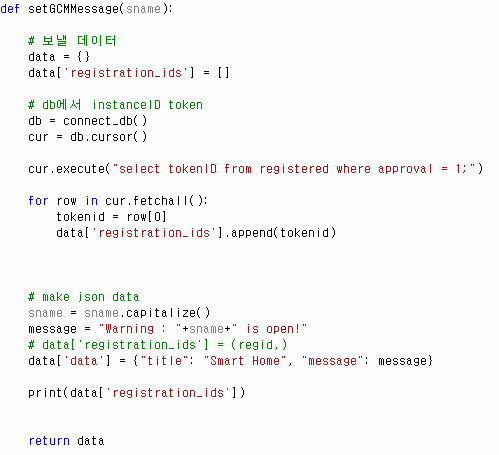
서버가 앱으로 GCM을 보낼 때에는 instance token id가 필요하다. 이 토큰 아이디는 instanceId\_listener\_service를 통해 얻는다. 앱을 최초로 실행했을 때 사용자를 등록하는 부분에서 이 서비스를 통해 토큰 아이디를 얻어 서버로 전송한다. 서비스를 통해 나온 토큰 아이디를 받기 위해 브로드캐스트 리시버도 정의해주었다.

웹 서버에서는 받은 사용자 정보를 데이터 베이스에 저장한다. GCM을 보낼 이벤트가 발생하면 API key와 보낼 data를 json형식으로 만들어 보낸다.



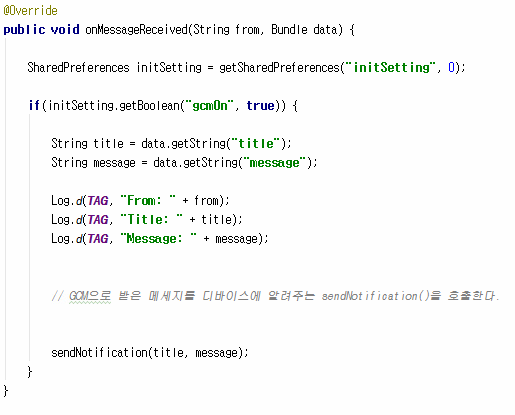
<GCM을 보내는 부분>

이 때 보낼 데이터에 instance id token이 들어가게 된다. 데이터베이스에서 사용자의 토큰 아이디를 select하여 리스트로 구성한 뒤 실제 보낼 메시지와 함께 json형식으로 보내게 된다. 데이터베이스의 registered테이블에서 등록된 사용자들의 토큰을 가져오기 때문에 등록된 기기에만 GCM이 보내지게 된다.



<GCM data 구성 부분>

안드로이드 앱에서 GCM메시지를 받았을 때엔 notify를 해주어야 한다. GCM Listener 서비스에서 메시지를 받았을 때 Notification을 생성하여 notify한다. 이 때 기기가 잠금 상태라도 알림이 뜰 수 있도록 wake해준다.



<GCM 메시지가 들어왔을 때>

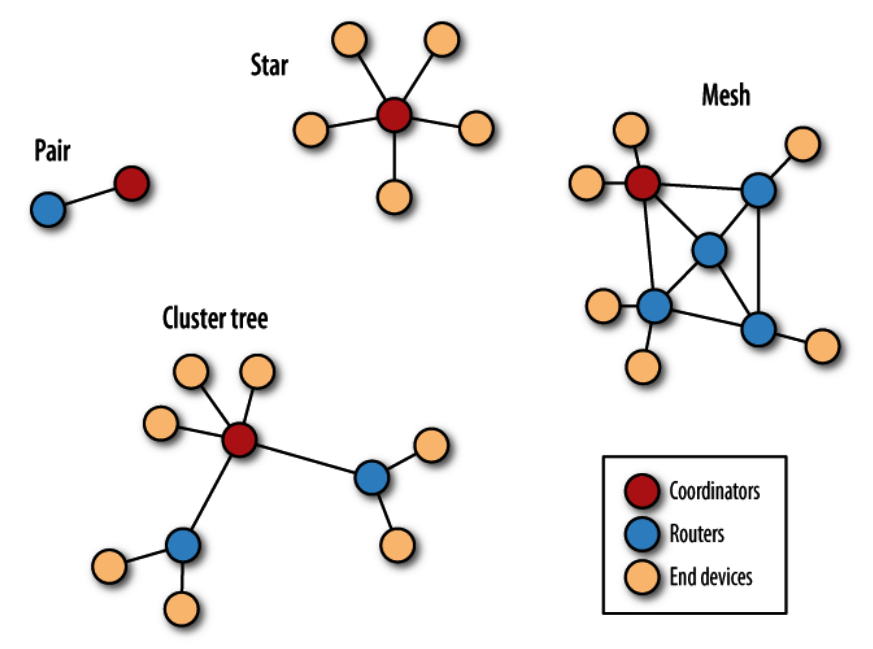


<Notification만드는 부분>

**5. Xbee**

5.1 Xbee 모듈 설정

Xbee간의 코딩을 하기 위해선 Xbee 쉴드에 올라가는 Xbee 모듈들을 설정을 해주어야 한다. 즉, 보드에 올라가는 Xbee 모듈이 통신에 있어서 어떠한 역할을 할 것 인지를 설정해주어야 하는데 일반적인 Xbee 네트워크 형태는 아래와 같다.

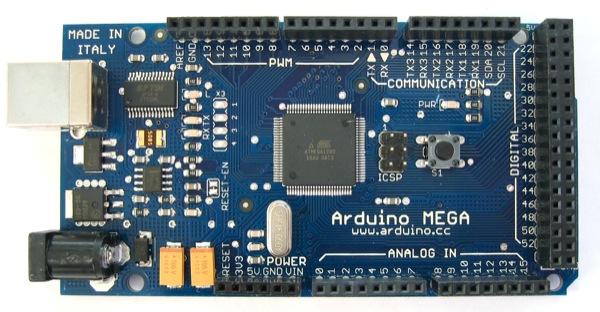


본 프로젝트에서는 이더넷 기능을 하는 중앙 아두이노에서 다른 서브 아두이노와 통신을 하는 것이기 때문에 Star 형태의 네트워크를 갖도록 중앙 아두이노는 Coordinator로 나머지 서브 아두이노는 End device로 설정하기 위해서 각각의 모듈의 Channel , PIN 코드, 통신속도와 목적을 설정하였다.

5.2 Xbee 통신

아두이노에서는 다른 아두이노와 통신을 할 때 시리얼을 이용한 통신을 사용하고, 이 시리얼 통신은 크게 두 가지로 나뉜다. 하나는 하드웨어 시리얼이고 다른 하나는 소프트웨어 시리얼이다. 시리얼 통신의 경우 비 동기 전송방식을 취하기 때문에 ‘수신 버퍼’ 라는 공간과 그에 따른 제어가 필요하다. 하드웨어 시리얼은 이에 대한 하드웨어가 준비되어 있기 때문에 시스템에 영향을 주지 않고 안정적으로 수신이 가능하다. 반면 소프트웨어 시리얼의 경우엔 시스템의 자원을 할당해서 시리얼 통신이 들어오는지, 또 시리얼 통신이 들어왔다면 이것을 따로 지정한 버퍼에 저장하는 과정 등을 수행하게 된다. 따라서 소프트웨어 방식은 하드웨어 방식과 비교해서 안정성, 효율성 등이 매우 떨어지게 된다.

아두이노의 보드 종류는 다양한데, 본 프로젝트에서는 아두이노 MEGA 2560(이하 메가)을 사용하였다. 메가는 다른 아두이노와 시리얼 통신이 가능하도록 시리얼 포트가 0번에서 3번까지 제공이 된다. 우리는 위에서 언급한 소프트웨어 방식의 단점을 생각하여 하드웨어 방식을 사용하였고, 3번 포트를 중앙 아두이노를 기준으로 송신하는 포트로 사용하였고, 수신하는 포트로는 2번 포트를 사용하였다.

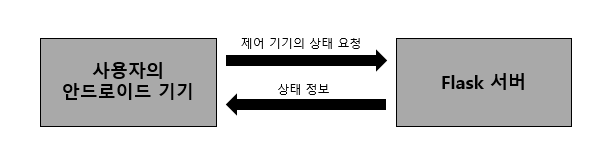


< 아두이노 MEGA >

이에 대한 모듈 테스트는 간단하게 아두이노 보드 3개 (중앙 아두이노,2개의 서브 아두이노) 로 구현하였다. 중앙 아두이노에서 이더넷 쉴드를 통해서 서버로부터 읽은 데이터를 String 타입으로 / 로 나눠서 Serial 3 번 포트로 Xbee 모듈에 전달 하였다. 그러면 Xbee 모듈에 저장된 목적지 주소로 데이터를 전송하게 되고, 서브 아두이노에서는 전달 받은 String 타입의 데이터를 읽고 String 을 ‘/’ 로 분할하여, 제어 명령과 제어 기기를 구분하도록 하였다. 그리고 제어 명령에 따라서 8번 포트에 연결되어 있는 led를 digitalWrite하였다.

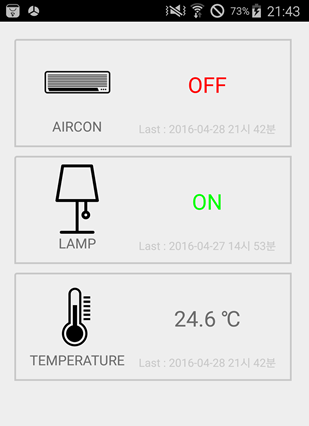
**6. 제어 기기의 상태 확인**

사용자의 안드로이드 기기를 이용해서 사물들의 상태를 확인할 수 있다. 현재 이 부분에 대해서는 사용자의 안드로이드 기기에서 제어할 사물의 상태를 확인할 수 있도록 구현된 Flask의 라우트로 기기명을 네트워크 작업을 하는 Thread를 통해 GET방식으로 전송을 한다. Flask 서버는 기기명을 받고 해당 기기에 대한 상태를 저장하 데이터베이스 테이블(Status)에 SELECT 쿼리를 보내어 현재 상태를 받아온다. 그 상태를 라우트 함수에서 XML형태로 반환을 하고, 반환된 XML형태의 데이터를 사용자의 안드로이드 기기에서 파싱을 하고 파싱된 데이터를 사용자의 화면에 보여주도록 하였다.



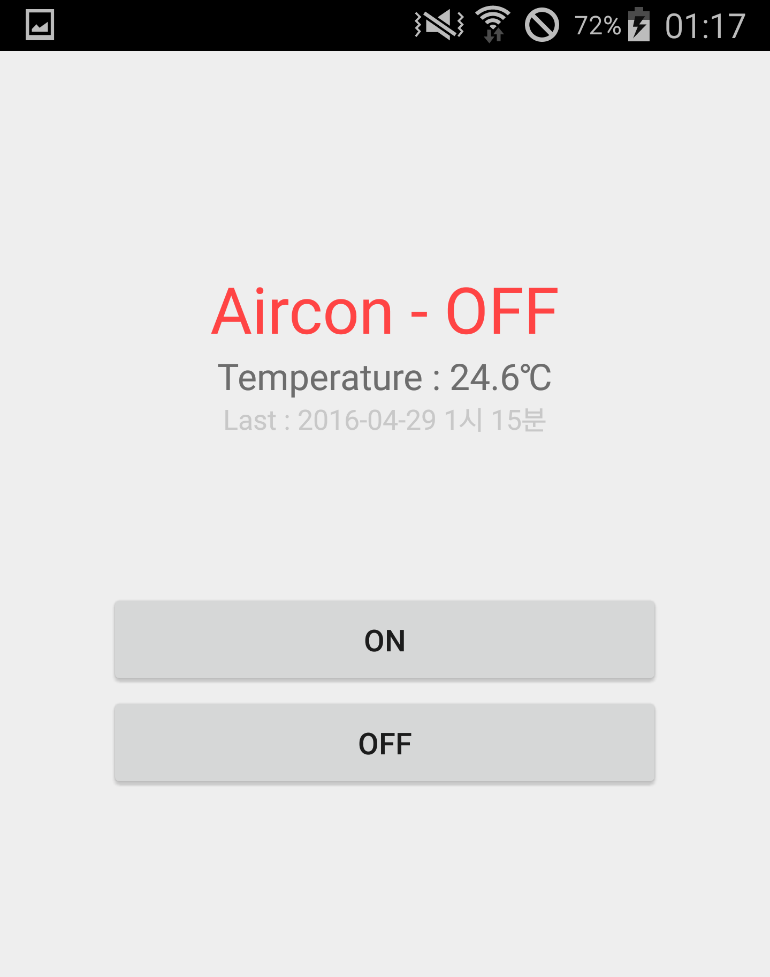
안드로이드 앱에서의 상태보여주기 창은 각 사물에 대한 정보를 가지고 있는 thread를 만든 후, 각 thread에 대해서 서버에 사물의 상태에 대한 정보를 요청하여 얻은 정보(사물의 상태, 가장 최근에 상태를 변경한 시간)를 액티비티에 수직적으로 보여주도록 하였다.

보통 상태보기에 더하여 사용자의 스마트 폰에서 현재 집의 온도를 액티비티에서 확인할 수 있도록 UI를 구성하였다. 현재의 온도를 가져오는 방법은 사용자의 기기에서 현재 온도를 가져올 수 있도록 ROUTE된 URL로 접근을 하게 되고, 서버에서 현재 온도에 대한 정보를 XML형태로 반환을 하면 사물의 정보를 받아오는 방식과 마찬가지로 XML Parser를 이용해 파싱을 하여 사용자의 액티비티에 나타나게 하였다.

****

< 상태 액비비티 >

또한 사용자가 에어컨 기능을 제어하는 액티비티에서도 상태 액티비티에서 사용했던 방식과 동일한 방식으로 온도를 실시간 온도를 가져온 후 화면에 보여주었다.

****

< 제어 액티비티 >

**7. 사용자의 패턴 분석 알고리즘 비교**

7-1 K-Means 알고리즘과 Elbow Method

서버는 사용자의 안드로이드 기기를 이용한 원거리와 직접 제어에 대한 기록(제어 시간, 사용자의 위치, 현재 집의 온도 등)을 데이터베이스에 저장을 한다. 그리고 서버에서는 이 기록들을 이용하여 에어컨에 대한 제어에서 사용자의 패턴을 추출하여 사용자에게 맞춤화된 서비스를 제공한다.

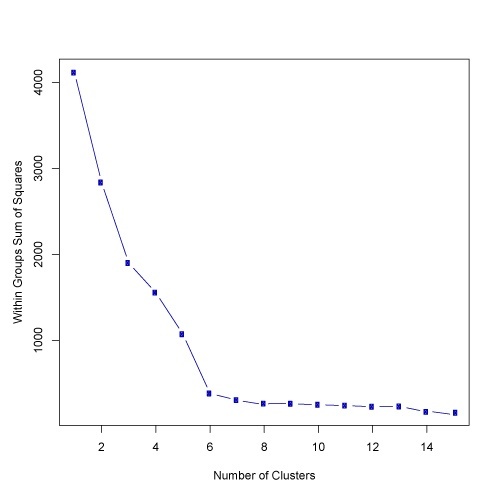
에어컨 제어에 대해서 사용자의 패턴을 뽑을 수 있는 특징(feature)으로는 사용자가 어떠한 시간대에 에어컨 제어를 몇번했는지에 대한 시간대별 에어컨 제어 데이터와 에어컨을 제어했을 때의 집의 온도 그리고 원거리 제어시 사용자의 제어 위치가 있다.

이를 이용한 패턴으로는 사용자가 에어컨을 제어했을 때의 온도에 대해서 생각해볼 수있다. 이 데이터들에 대한 분포는 특정 온도에 대해서 사용자가 반응할 수 있으므로 ‘한곳에 집중되어 분포되어 있을 것이다’라고 가정을 하여 모델링을 하였다. 이를 위하여 온도에 대한 적절한 확률 분포를 Exponential Family(Poisson, Gaussian, Laplace) 중에서 하나로 가정하고, python에서 제공하는 scipy.stats 라이브러리를 이용하여 구현한다. 이렇게 가정한 분포를 이용해서 분포의 평균에 해당하는 파라미터 값의 확률값과 데이터 개수를 threshold하여 데이터베이스에 저장한다. 그리고 아두이노 센서로부터 얻은 값이 해당 온도가 되었을 때 사용자의 자동 제어 설정이 활성화되었다면, 자동으로 제어를 하도록 서버에 요청을 보낸다.

다음 패턴으로는 사용자가 원거리 제어를 할 때의 위치를 생각해 볼 수 있다. 이에 대한 패턴은 온도와는 다르게 어느 한곳에 집중적으로 분포하기 보다는 한 곳 이상에서 집중적으로 분포할 수 있다. (예를 들면 집 앞 지하철, 회사) 그렇기 때문에 이 패턴에 대해서는 확률 분포를 가정하기 보다는 클러스터링 알고리즘을 적용하였다. 클러스터링 알고리즘도 종류가 있는데, 여기서는 거리 기반 클러스터링을 하기 위해서 k-means clustering을 선택하였다.

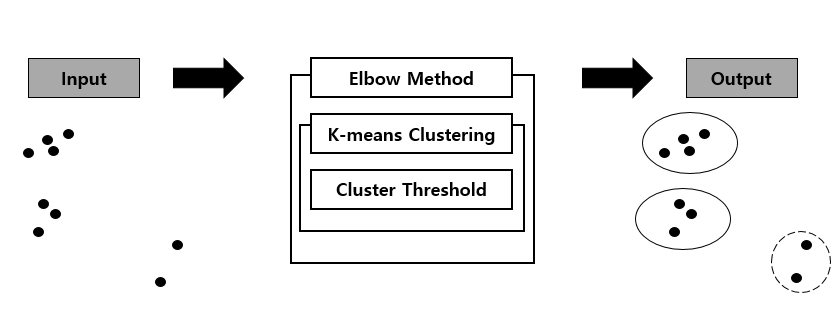
이 알고리즘에선 입력으로 데이터와 K, 즉 클러스터의 개수를 미리 정해줘야 하는데, 최적화 된 클러스터의 개수를 입력으로 주기 위해 Elbow Method를 이용하였다. Elbow Method의 방법은 클러스터의 개수 k값에 대한 SSE(Sum of Squared Error)를 구한 다음 이 Error가 가장 크게 변할 때(Elbow effect)의 k값을 찾는 것이다. 그래서 k값을 작은 값부터 증가시키면서 각 클러스터에 대한 SSE의 합을 구하고 변화율을 구한 후에 가장 크게 변한 구간을 찾음으로써 K값을 정하였다.





< K = 6이 최적일 때의 Elbow Method >

이 알고리즘을 이용하여 데이터들을 클러스터링 한 후에 출력값으로 얻은 클러스터들의 표준편차 및 분산을 이용하여 클러스터의 크기를 확인을 하고, 이것도 마찬가지로 데이터의 개수와 각 클러스터의 표준편차를 이용하여 threshold하여 클러스터를 구분한다.



< 알고리즘 구성도 >

이 과정을 통해 나온 출력 클러스터의 mean에 해당하는 위치를 사용자의 안드로이드 기기에 전송을 하고 안드로이드 기기에서는 이 값을 내부 데이터베이스에 저장을 한다. 그리고 사용자가 해당 위치 부근에 도착했을 때 사용자의 자동 제어 설정이 활성화되었다면, 자동으로 제어를 하도록 서버에 요청을 보낸다.

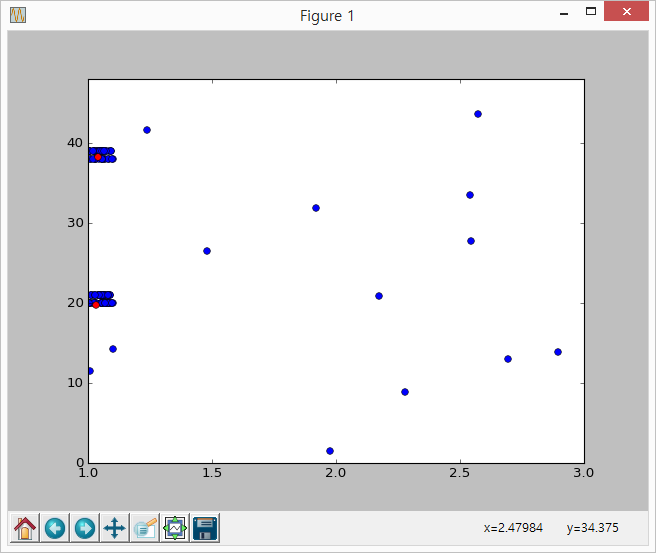
7-2 K-Means 알고리즘과 K-Medians 알고리즘 비교 및 구현

K-Medians 알고리즘과 K-Means 알고리즘을 비교하였다. 알고리즘의 차이는 각 클러스터의 중심점을 선택을 할 때, 전자는 클러스터에 속한 중간 값을 선택하는 것이고 후자는 클러스터의 속한 데이터들의 평균값을 선택하는 것이다. 사용자의 제어 패턴을 분석할 때 몇몇의 outlier들은 당연히 생기기 마련이고, 이에 대해서 알고리즘을 수행해보았을 때 두 알고리즘의 차이가 확연하게 드러났다. 결과는 아래와 같고, 각 축의 의미는 다음과 같다.

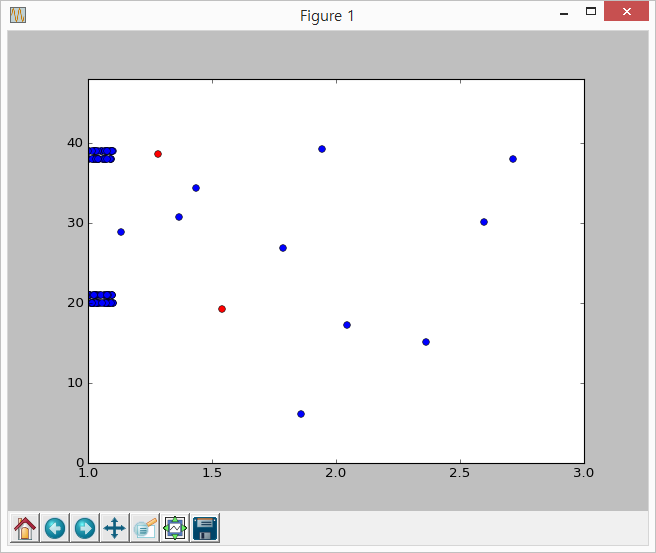
x축 : 제어했을 때의 사용자의 집까지의 거리(사용자의 위치)

y축 : 제어했을 때의 시간(30분 간격으로 나눔)

그리고 입력으로는 120개의 제어 기록을 사용자의 두가지 패턴에 대해서 모이게 하고, 그 중 10%에 대해서는 random하게 입력으로 주어 outlier(만약 이게 outlier가 아닌 한곳에 뭉쳐 있었다면 Elbow Method에 의해서 새로운 클러스터가 생성되었을 것임) 에 얼마나 민감한지를 확인하려고 한다. 먼저 K-Medians 알고리즘을 수행했을 때의 결과이다.



10%의 outlier에도 사용자에게 패턴서비스의 값으로 제공되는 클러스터의 중심값은 중간값을 선택하기 때문에 데이터가 많이 보여져 있는 사용자의 패턴 부분에 속해있음을 볼 수 있다. 반면에 K-Means 알고리즘을 수행하였을 때의 결과는 아래와 같다.



이 알고리즘은 중심값을 클러스터 데이터의 평균값을 선택하므로 사용자가 이전에 수행한 적이 없는 위치, 시간에 대해서 클러스터의 중심을 선택해 패턴을 선택하게 된다.

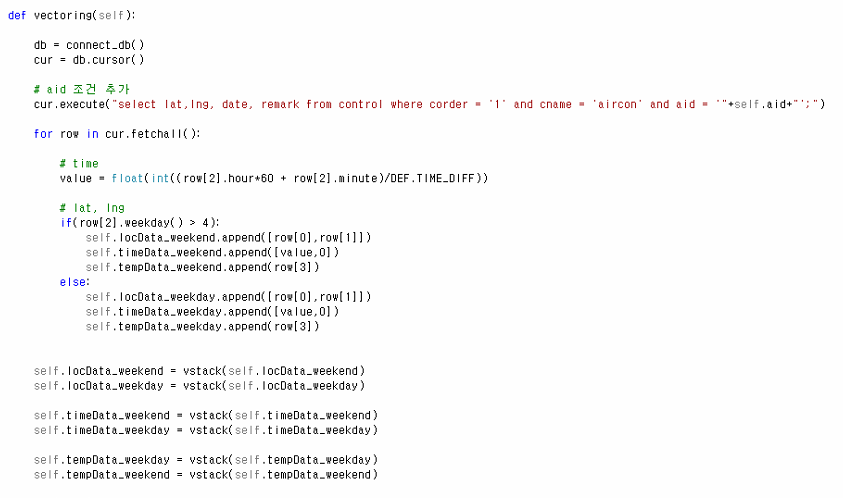
물론 Elbow Method를 통해서 큰 오차는 피할 수 있지만, 좀 더 정확하게 수행하기 상대적으로 outlier에 덜 민감한 K-Medians 알고리즘을 선택하기로 하였다.

이렇게 클러스터링 알고리즘으로 나온 각 클러스터의 개수가 사용자의 패턴의 개수가 되는데, 이 패턴의 개수가 Elbow Method를 통해서도 어느정도 조절은 할 수 있지만, 만들어진 클러스터를 항상 패턴으로 보기에는 무리가 있다. 그래서 클러스터의 표준 편차, 데이터의 개수에 대해서 threshold 하였다. 표준 편차의 경우 각 축에 대한 단위가 다르므로, 각 축에 대해서 따로 표준편차 값을 threshold 값으로 선택하였다. 그리고 이를 통과한 클러스터의 정보(위치, 시간)를 서버 데이터베이스 pattern 테이블에 넣는다.

**8. 패턴 자동 제어 구현**

8.1. 패턴 생성

사용자의 패턴은 테스트를 통해 Elbow Method 와 K-Medians 알고리즘의 결합을 이용해서 추출하기로 하였다. 그래서 테스트 후 실제 데이터를 이 알고리즘에 사용하기 위해서 사용자의 제어 기록(DataBase 튜플)을 가져와서 필요한 데이터로 모두 split 하였다. 이때 패턴은 주말 패턴과 주간 패턴이 존재하므로 각 데이터의 시간 데이터 값을 확인하여 주간과 주말을 나눠주었다. 클러스터링은 위치, 시간에 대해서 수행하도록 하였다.



< 디비 튜플 값을 필요한 list에 적절하게 배치하는 부분 >

먼저 위치에 대한 클러스터링을 하면 적절한 개수의 cluster가 생성되는데, 해당 cluster에 대해서 그 cluster에 속한 데이터들의 시간 값들에 대해서 다시한번 클러스터링을 한다. 그렇게 해서 결과 데이터를 패턴 데이터에 삽입을 해준다.



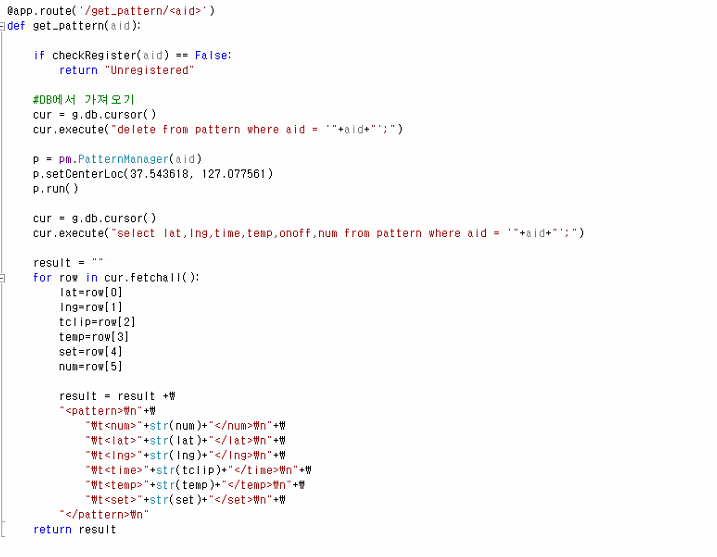
< 클러스터링 결과 처리 >

이렇게 만들어진 패턴을 서버 DataBase에 넣는 수행을 하도록 하였다.



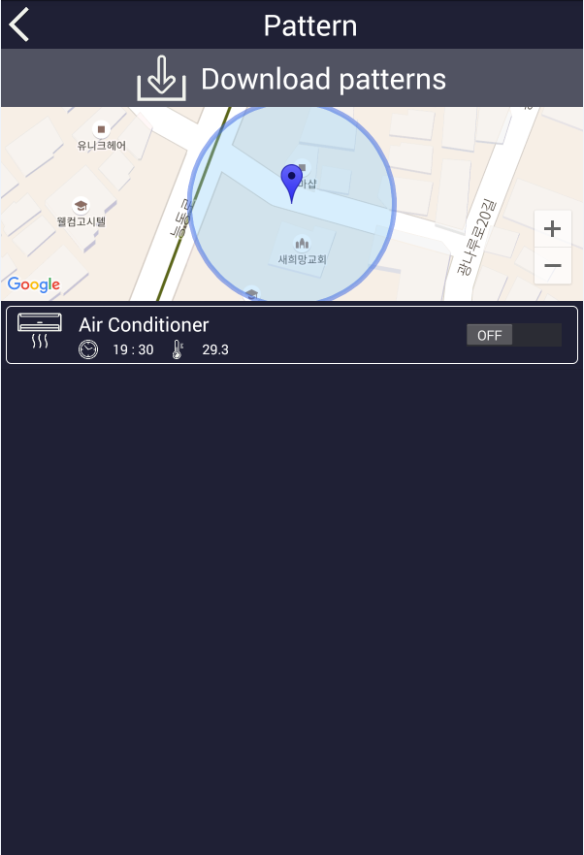
< 패턴 데이터를 서버 Database 에 삽입하는 부분 >

삽입된 패턴 데이터들에 대해서 사용자의 안드로이드 기기에서 가져올 수 있어야 하므로, 서버에 패턴 데이터를 xml형태로 얻을 수 있도록 서버단에서는 사용자의 안드로이드 기기 아이디로 데이터베이스 패턴 값을 읽어와 xml 타입의 데이터를 만들고 뿌려주도록 하였다.



8.2. 안드로이드 앱 패턴 확인

사용자의 패턴 데이터에 대해서 사용자가 확인을 할 수 있도록 액티비티를 구현하였다.



< 패턴 액티비티 >

Download pattern을 선택하면 서버에 패턴을 얻을 수 있도록 route된 위치에 접속하여 xml 형태로 패턴 데이터를 받아온다.

이렇게 얻은 패턴 데이터들에 대해서 보기 쉽게 리스트뷰 형태로 사용자에게 보여주었다. 그리고 그 리스트 뷰를 선택했을 때, 사용자의 제어 패턴 위치를 Google map 을 이용하여 보여주도록 하였고, 제어가 자동으로 수행되는 범위를 사용자에게 보여주었다.

또한 사용자가 해당 패턴에 대해서 자동 수행 여부를 선택할 수 있도록 토글 버튼을 위치시켰다.

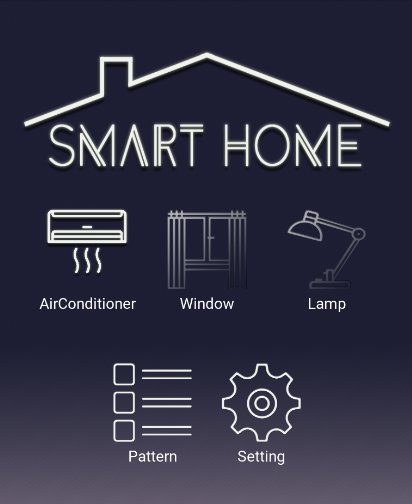
**9. 안드로이드 전체 UI 수정**

안드로이드 전체적인 디자인과 상태를 보여주는 방식을 수정하였다. 먼저 상태 같은 경우, 이전엔 상태 액티비티를 만들어서 사용자에게 보여주었다. 그러나 꼭 상태 액티비티가 필요한지에 대해 피드백을 받았고, 사물에 대한 상태를 메인 액티비티에서 보여주도록 하였다.



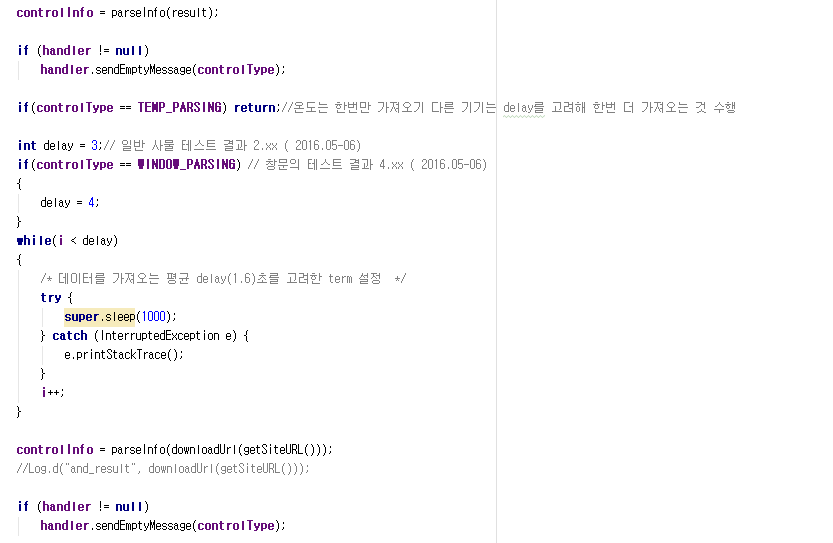
< 메뉴 액티비티에서 각 사물에 대한 정보를 가져오는 부분 >

메인 액티비티가 실행되었을 때, 네트워크 thread를 run하도록 하여 각 상태에 대한 xml 데이터를 서버로부터 받아 파싱 후에 그 상태에 대해서 메인 화면에 보여주는 이미지를 다르게 하도록 하였다.

< 메인 액티비티>

또한 이전에는 각 상태창에 대해서 사물을 제어했을 때, 그 상태 정보가 바뀐 상태가 한 액티비티내에서 변경이 되지 않았었다. 그래서 테스트 결과서에 따른 사용자에게 상태 정보를 보여주는 시간의 평균치만큼 thread sleep을 하도록 하고 그 이후에 다시 한번 파싱을 하도록 수행함으로서 변경된 상태를 제어 후에 액티비티의 이동 없이 바로 보여주도록 하였다.

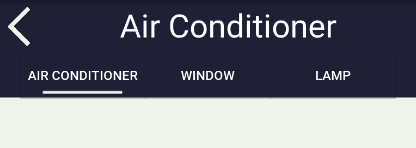


<각 사물에 대해서 파싱을 하도록 호출하는 부분 + 사물에 대한 딜레이 설정>

또한 파싱 후 결과를 UI에 나타나도록 하기 위해(파싱을 하는 thread는 async한 통신이다.) 파싱 결과가 나왔을 때 handler로 message를 보내어 ui를 변경하였다.



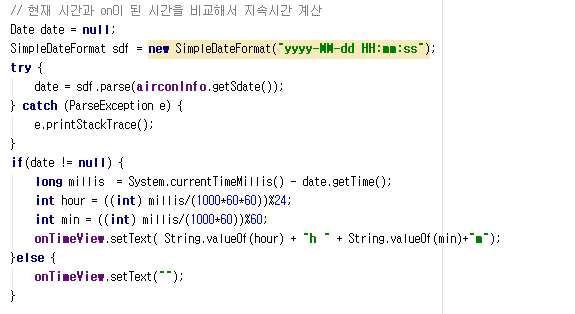
그리고 사용자에게 좀 더 편리한 UI를 제공하기 위해 탭메뉴를 만들어 여러 사물들에 대하여 제어를 할 때 이동이 간편하도록 하였다.



< 상단 탭 매뉴 >

AirconActivity같은 경우는 다른 사물들과는 다르게 현재의 온도를 확인하고 제어를 하는 것이 도 좋으므로, 현재의 온도를 아두이노 센서로부터 바로 받아와서 사용자에게 보여주었다. 그러나 여기엔 round – trip delay가 소모되므로 어느정도 시간이 필요하다. 그러므로 사용자에게 이 액티비티에 들어올 때 progress dialog를 띄워 약간의 delay를 가진 후에 보여주었다.

게다가 이 액티비티에서 사용자가 aircon을 얼마나 지속적으로 사용했는지를 보여주기 위해서 앱에서 현재 시간과 aircon이 on된 시간을 객체로 변환 후에 그 차이 값을 사용자애게 보여주도록 하였다.



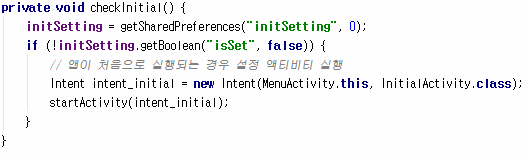
< 현재 시간과의 차이를 이용하여, 에어컨 지속시간을 보여주는 부분 >

**10. 보안**

10.1. 사용자 등록

보안을 강화하기 위해 사용자는 안드로이드 앱을 최초로 실행하였을 때 사용자 등록을 하고, 웹서버에서 이를 승인하여 승인된 사용자만 스마트홈의 모든 기능을 이용할 수 있도록 하였다.

앱이 최초로 실행되었는지의 여부를 알기 위해서 SharedPreferences를 사용하였다. 최초로 실행된 경우 해당 shared preference의 isSet부분이 false이기 때문에 등록 액티비티를 띄운다.



<첫 액티비티에서 최초 실행 체크 부분>

그 후 등록 액티비티에서 디바이스의 아이디와 instance id token을 가져온다. 토큰을 가져오는 데에는 시간이 걸리므로 그 동안엔 등록 버튼이 안 보이도록 하였다. 사용자에겐 사용자의 이름과 사용자 집의 위치를 입력 받는다. 등록이 끝나면 SharedPreference의 isSet부분을 True로 바꾼다.



<등록 버튼 클릭 시 수행 부분>

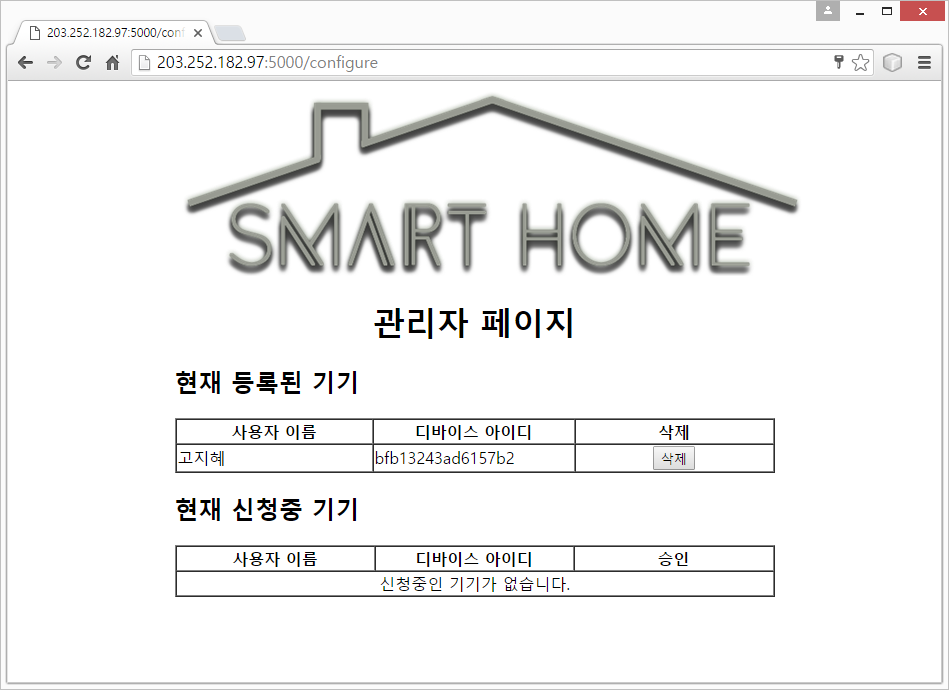
10.2. 사용자 승인

스마트홈 사용자가 모두 앱을 설치를 하고 바로 제어 및 다른 수행을 하지 못하도록 관리자 페이지를 두었다. 사용자가 앱을 최초에 실행하였을 떄, 자신의 정보를 입력을 하고 register버튼을 선택하게 되면, 서버 데이터베이스의 registered 테이블에 삽입이 된다. 그러나 그 테이블의 approval 필드의 값은 0 으로 들어가며 1이 되어야만 모든 수행을 가능하도록 하였다.

이렇게 신청을 하게 되면 서버의 관리자 페이지에 해당 유저에 대한 정보를 볼 수 있도록 하였고, 승인 및 삭제 버튼을 이용해서 해당 유저에 대해서 제어를 허락할 것인지 거절할 것이지에 대해서 선택할 수 있도록 하였다.

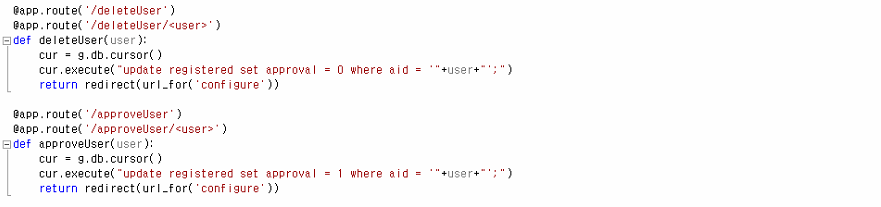


< 관리자 메인 페이지 >



< 현재 등록된 기기와 신청중인 기기 확인 >

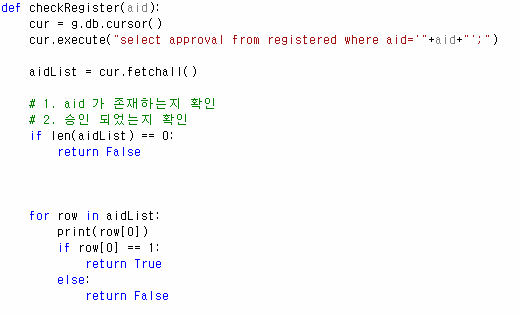
버튼을 누르는 행동에 대해서는 데이터 베이스 registered 테이블의 approval 필드에 변화를 주도록 함수를 구현하였다.



< 버튼 선택에 따른 함수호출 및 데이터베이스 update >

10.3. 제어 및 상태보기

안드로이드 앱에서 웹서버에 무언가를 요청할 때 뒤에 안드로이드 기기 아이디(aid)를 같이 붙여서 보낸다. 서버는 작업을 하기 전에 항상 checkRegister(aid) 함수를 호출하여 해당 아이디가 승인된 아이디인지 확인한다. 승인되지 않은 아이디면 작업을 수행하지 않고, 승인된 아이디면 작업을 이어나간다.



<서버 측 chetRegister함수>

승인 되지 않은 아이디면 서버는 승인되지 않았다는 결과를 안드로이드 앱으로 보내고 안드로이드 앱에선 승인되지 않았다는 팝업 창을 띄운다.

11. 집 프로토타입 구현 (모터, LED와 센서 이용)

아두이노를 이용하여 원거리 제어 스마트홈을 구현하기 위해서, 모터 드라이버와 LED 그리고 센서(GPS, 온도)를 이용하여 스마트홈의 집 모델을 구현하였다.

에어컨 기능을 하도록 모터와 프로펠러를 결합하여 기기를 구현하였고, 모터 드라이버와 바퀴를 결합하여 자동 창문을 구현하였다. 그리고 조명의 기능을 하도록 LED와 사용자의 안드로이드 기기에서 집 상태를 알 수 있도록 센서(GPS, 온도)를 설치하였다.

창문이 열렸는지 열리지 않았는지 그 상태정보를 알아내기 위해서 마그네틱 도어센서를 사용하였다. 도어센서를 브래드보드를 이용하여 창문 쪽에 연결하고 신호가 들어오는 지 delay를 가지고 확인을 하도록 하였고, 상태에 변화가 생겼을 때 중앙 아두이노로 자신의 상태를 바로 보내도록 하였다.



< 마그네틱 도어 센서 >

또한 에어컨 같은 경우엔 온도센서가 필요하므로, 온도 센서를 브래드 보드에 설치를 하고 들어오는 값에 대해서 적절하게 변환을 한 후 celcius 온도를 사용자가 실시간 온도를 요청을 하였을 떄 보내도록 하였다.



< 온도에 대해서 처리하는 부분 >