Project: *insert cool name here*
Home automation proposal
Version 1.1

Updates:

Since the last release:

Added project goals. Added problem/solution notes to a few sections. Expanded example walkthrough. Expanded general overview a bit. Added highlighting to make the messages easier to see.

Notes

This is a proposal for the general design of the system. Since this has to be a group project I've added on to the project goals in the hope that it will provide enough work for 4 or 5 people, and so that the finished project is flexible enough that it can continue to be used and expanded after the class is finished. I've tried to keep this document as short as possible. I'll try to explain the proposal as a series of increasingly detailed overviews, starting a generalized view of the system as a whole then drilling down to a more detailed view of the hardware and software of the various units then ending with a detailed walkthrough example. Where possible I've added notes at the end of each section covering possible problems with the design as well as proposed solutions. Since this is going to be a project with a number of moving parts I'd like to make sure that we have a chance to agree on the design and standards before any code is written.

Project Goals

The concrete goals of the project are listed below. Goals highlighted blue are optional if we have enough time/team members.

1. Create a coordinator software package.
    a. Creating list of known remote modules
    b. Load drivers for each module
    c. Message passing
        i. Agree on a standard protocol for message passing
        ii. Encrypt messages in transit
        iii. Provide an api for the frontends to use
    d. Scheduled driver wakups
    e. Xsl transform hosting
2. Develop at least one front end for the project:
    a. Web site (hosted on raspberry pi)
        i. Desktop Version
        ii. Mobile Version
    b. Native android frontend
    c. Native iOS frontend
    d. Tablet/Touchscreen
3. Develop a number of remote modules (every team member can work on their own goals).

   a. Create skeleton Arduino code that each project can adapt.
      i. Agree on a standard for *remote module* status responses as XML
   b. Create skeleton driver
      i. Agree on a standard to represent widgets as XML
      ii. Agree on a standard to represent detailed control of *remote modules*
4. Create a distribution package that can be installed on any raspberry pi
   a. Include dependencies for apache, mysql, etc…
   b. Provide empty example database
   c. Provide example drivers/arduino code.

Definitions

I've tried to stick with some standard terms for the hardware and software throughout this document (noted in italics). The names are just placeholders for now, maybe we can find nicer ones later.

Controller – the combination of hardware and software that acts as the base station of the system. It is responsible for running the database, the web server, and the coordinator software. The default configuration is a raspberry pi running linux with an arduino/xbee combo connected via USB.

Coordinator – A software package running on the controller. It is responsible for:

- Starting drivers to handle running the remote modules
- Message passing between drivers and remote modules
- Providing access to permanent storage (likely the mysql database)
- Providing mechanisms to wake a driver a specific times
- Responding to requests from the frontend(s) with the status of the remote modules
- Logging all communication
- Logging facilities for the drivers

Remote module – the combination of hardware and software that provides direct access to the automations. The default configuration for these modules is an Arduino connected via an Xbee module.

Driver (or plugin) – The drivers are the bits of Java code that the coordinator spawns during its startup. The drivers are Java Runnables that implement either the `SensorModule` or `ControllerModule` interface (or possibly both). The drivers are the brains of each remote module, and preform high level processing that would be difficult to do on the arduinos alone.

A General Overview

This is the initial proposal for a system of hardware and software that allows for multiple home automations. The system consists of a single centralized home station and zero or more remotely controlled sensors or controllers. For the sake of brevity the home station hardware and software will be referred to as the *controller* and the wireless units as *remote modules*. The *remote modules* are responsible for directly controlling whatever hardware they are attached to, and are controlled by the software running on the arduino as well as a higher-level driver running on the controller. A software package called the *coordinator* runs on the controller and acts as the go between for any given frontend and the *remote modules*.

A (Slightly Less) General Overview

The system is composed a single *controller* and zero or more *remote modules*. The *controller* is a raspberry pi running a version of linux (which version is not so relevant so long as it has support for mysql, apache, and java). The *controller* runs a web server, mysqld, and the *coordinator* software (more on this later). The web site serves php pages that act as a frontend to all the functions that the *controller* and the remote modules offer (this can be expanded later to different frontends, the web frontend will likely be the easiest to create). The *coordinator* software is a Java program that handles the bulk of the work managing the remote modules. It should be able to send commands to the remote modules and to receive data back from the remote modules. The *coordinator* also contains plugins that act as drivers of a sort for the remote modules. This allows the *coordinator* core to remain small, providing only a small number of features. When the *coordinator* core receives data from one of the remote modules it forwards the data to the correct driver for processing. The drivers for the remote modules are responsible for handling the bulk of the work for the remote modules. When the drivers reach a state in which a remote module needs to be activated or polled then it forwards a request to the *coordinator* core, which in turn passes it to the remote module through the attached arduino/xbee. The *controller* is responsible for all of the high level functions of the home automations, with the drivers providing the heavy lifting specific to the remote module they are built for.

Notes: Since the coordinator software is Java based it could be possible to use any computer with a JRE to act as the controller. Since there will likely be system specific shell scripts used it will probably have to be a computer running a linux variant. It would be possible to use a laptop or desktop pc as a development platform to keep costs down.

The remote modules consist of an arduino with an xbee attached. The arduino may also have additional hardware connected as fit to preform its function. This additional hardware can be broken into two basic categories, sensors and

controllers. This allows the remote modules to be classified either as a sensor module, a controller module, or a combination module. As an example, a module that records temperature periodically would be a sensor module. A module that waits for a command from the *controller* to drive an actuator to open or close blinds would be a controller module. A module that contains a photo cell that automatically triggers a light to turn on when it becomes dark outside would be a combination module. The different types of remote modules should conform to a standardized interface for basic functions. Access to more detailed data from a remote module is limited to the driver running on the *controller*. This means that the driver must be able to order the data it receives from its module in a way that can be useful (more on this later).

Notes: All remote modules in your project will have to use the same model of xbee adapter, but it shouldn't matter which you choose. The arduino code to connect to the xbees should remain the same. Note that it is also possible to use the xbee without an attached arduino, using the microcontroller on the xbee to perform all functions. Since there are already home automation solutions available it would be possible to build a remote module that converts commands from the controller into a format that can be read by the pre-built automation.

Wireless Communication

Both the *controller* and the *remote modules* need to have a way of communicating with each other. There are numerous ways to achieve this: infrared, Bluetooth, wifi, and others. Infrared is very cheap but requires line-of-sight or a series of repeaters for communication to work. Bluetooth is (fairly) cheap, but it is a fixed protocol, using it to send arbitrary data over a network is cumbersome at best. It also has the limitation that the devices are supposed to the paired with each other before transmission, which could be a pain. Wifi offers a viable alternative. The cost is not cheap, but, once setup, transmission would be trivial. I gave this serious consideration when looking for a way to control the remote modules but had to give up on it for two reasons: power and memory.

The arduinos have very limited memory (the Uno is limited to 32k) and clock cycles (16Mhz on the Uno). Having each arduino run code to manage connection to the wireless network would take up an inordinate amount of processing time, and would require that everyone on the team implement wifi code on their setup. The power consumption of wifi was another concern. Even taking steps to reduce the power use, it might not be viable to use a battery powered *remote module* if it uses wifi to transmit data.

These limitations were the reasons that I'm proposing that we standardize on the xbee modules to handle the wireless transmissions. The cost is not trivial ($20 for a single module depending on the model, plus $10 for an optional adapter to fit the xbee in a standard breadboard), but I think the benefits outweigh the cost. The Xbees only require that they be programmed once with the network name (a hex string). After that a powered Xbee will connect to any other Xbees within range that use the same network name. The xbee handles data transmission without any additional code on the ardino. The code on the arduino side is a simple as writing a string chars to a serial line. Receiving data from the network is as simple as polling a serial line for input. The power consumption of the Xbee module should be less than wifi, possibly less than Bluetooth (but still greater than IR). This should help when working on remote modules that must be battery/solar powered. One additional reason for choosing the Xbee was that the Xbee contains its own microcontroller. With a little work we might be able to convert some of the simpler *remote modules* so that they work without the arduino. I don't have any experience programming the xbees manually though, so this may be something that would have to wait till most everything else is done.

Wireless Communication Protocol

One of the problems that we are going to run into in a project like this is the variety of different modules that can be built, and the likelihood that none of us will have the same number and type of modules. This will make it very hard to have a generalized protocol to poll a module for data from one or more of its attached sensors, or to control a specific hardware component. We could develop a protocol to address this, and send a command to the remote module that looks something like:

```
CHANGE VOLTAGE ON LED2 TO HALF
```

This could work, but would need a lot of code on the arduino to convert the high level commands to `digitalWrite()` calls on specific pins. Given the extremely limited memory and clock cycles on the arduinos this could be very cumbersome. This method would also make it hard to know what sensors and controllers the module provides, as, in the example above, there might not be an LED2 on the specific module. This was the main reason that I am proposing that we use Java code running as a driver on the *coordinator* to handle as much of the work for each remote module as possible. Since each driver is written explicitly for one (and only one) remote module it can have intimate knowledge of the remote module, even down to the pin wiring. This would allow the above command to be changed to something as simple as:

```
12:0.5
```

Since the java driver knows that the LED named LED2 is on pin 12 it can simple specify the pin number and new voltage level. This method also has two additional benefits: The remote modules can remain stateless and we don't have to waste any time agreeing on a standardized protocol to interact with the remote modules (other than some issues addressed below).

Since the state is only relevant within the java code the arduinos do not need to process the commands for sanity. They can blindly execute the command and not worry about keeping track of what has been done. If at a later time you decide to expand the module with additional sensors or controllers the protocol that you use can be expanded without affecting any of the other modules.

There is, however, one exception that I have to mention. The xbee modules use a simplistic method to route messages over the network. Every message sent from the *coordinator* will be received by every *remote module*. To make sure that the message is parsed by the correct module a prefix should be added to each message that uniquely identifies the destination module. Either a random hex string or a human readable name would work here. With that in mind the actual message broadcast by the *coordinator* to the *remote* module named "my_lights" would look something like:

```
my_lights:12:0.5
```

Another problem that we will run into is the limitation in the arduino api for reading serial input. The arduinos only read one character at a time from serial input, so reading the above message would require reading multiple chars, appending each to a string. Since we need to know where the message ends we need to use a special character to indicate the end of the message. I would suggest we stick with the newline character, so that the final form of the message would be:

```
my_lights:12:0.5\n
```

This way the only part of the protocol that we have to agree upon is the format of the opening tag and the closing character. Everything in between is fair game.

Notes: There are a few problems with this solution:

1. Every remote module has to process every message received. This could result in reduced battery life for some modules if the network has even a single busy remote module.
2. The examples so far have all been in plaintext. This works well when sending commands and receiving simple readings, but will be cumbersome to adapt for arbitrary data. If a remote camera module sends picture data back to the controller the data would have to be formatted as text, which would take additional time and memory for the arduinos to process. I have not found a solution short of caching the data on an attached sd card before transmission.
3. All of the data so far is being transmitted in clear text. Working out a solution to encrypt all transmitted data would be a large task.

The Frontend(s)

The *coordinator* does not present a frontend to the user, but instead provides an api to handle requests. Since the *coordinator* does not keep track of the state of the modules it can only directly respond to basic queries (i.e. what modules are present). Any queries related to a specific module is forwarded to the corresponding driver, which replies back with its known state. To make the *coordinator* as flexible as possible the api should be made available via a HTTP interface. It might be worthwhile so see if there are any small http web servers available that can be embedded in the project.

To further keep the *coordinator* as agnostic as possible the replies to api calls can be responded to with XML data. This would mean that (as part of the `SensorModule` and `ControllerModule` interfaces) the drivers should have a standardized interface to request data or to send commands. As an example each driver might be required to implement the method `getSensorStatus()` which would respond with something like:

```
<?xml version="1.0" encoding="UTF-8"?>
<sensorList>
  <sensor>
          <name>Light Value</name>
          <value>0.3</value>
  </sensor>
  <sensor>
          <name>LED 2</name>
          <value>off</value>
  </sensor>
</sensorList>
```

This would allow each frontend to adapt the data to their own display method. If a standard xml format can be agreed upon then the *coordinator* can also supply xsl transform data for the responses, making it very easy to write frontends that display html. This solution could also be used to have each driver supply their own widgets. These widgets would then be used by the frontends to provide a method to control the *remote modules*.

A Simple Example

It might be helpful to look at an example setup and walk through the steps that happen within the *controller*, *coordinator*, and the *remote modules*. This will likely be verbose to begin with so that every detail is visible. At it progresses I will try to skip some details for brevity. The protocols, error codes, etc. are just examples.

In this example let us suppose that there is one *controller* with an attached arduino/xbee, and two *remote modules*, one that contains a light sensor and a single LED and one that has an array of LEDs attached. Let us also suppose that a user will be using the website frontend to manipulate the LED array, and that the single LED will be manipulated based on readings from the light sensor. At the beginning of this example both *remote modules* are already turned on, but the *controller* is powered off. After powering on the *controller* the boot process begins and apache, mysql, and the *coordinator* software are loaded on the *controller*. The arduino is powered via the USB cable, and once powered opens a connection to the attached Xbee. The Xbee powers on and opens a connection on its network listing itself as the controlling node. The xbees on the *remote modules* agree and establish a link with the controlling node.

The *coordinator* software starts and loads all plugins (drivers) within the plugins/ folder, adding each plugin to a list of running plugins. The plugins inherit from java.runnable and will run in their own threads. The *coordinator* then polls the connected arduino until it receives a `READY\n` reply. After this it can assume that the Xbee connection is also up and ready to send data. The *coordinator* then sends a broadcast on its xbee network asking for the *remote modules* to identify themselves by sending the message (example message for now):

```
ALL:NAME\n
```

The arduinos on the *remote modules* receive the command by polling the serial line attached to the xbees and reading a single character at a time (this is a limitation on the arduino API). Once the newline character is reached the arduino knows that a complete message has been received, and passes the string on to a method to parse the message. The arduino parses the tag first, and sees that this message is addressed to ALL, a reserved tag that indicates that the message is for all *remote modules*. The command is `NAME`, which the arduino recognizes as a command to respond via the Xbee with its unique name. Both modules respond accordingly.

The arduino on the *controller* reads characters from its serial line attached to the xbee, passing each character to the serial line attached to the USB bus. The *coordinator* software reads the serial line, and receives the data:

```
light_sensor:READY\nled_array:READY\n
```

It then breaks this input by newline and adds the responding names to an internal list of known *remote modules*. Since the modules are stateless they do not keep track of whether the *controller* is up, and will only respond when queried. After this the *coordinator* is done with its setup and waits for commands from the drivers that have been loaded.

While the *coordinator* core is doing its initial setup procedure the drivers have already begun to execute. Since they exists as their own threads they do not need to wait on the *coordinator* to finish its own setup before beginning their own (though sending messages on the network does need to wait until the *coordinator's* setup is completed).

The driver for the *remote module* with the light sensor and LED begins executing. The software is designed to poll the light sensor and turn the led on if the light is above some predefined level. The driver broadcasts a message by using the `Coordinator:broadcastMessage(String unitID, String message)` method with a unitID "light_sensor" and a message "r". The *coordinator* notices that a message is waiting to be broadcast. The *coordinator* checks that there is a known *remote module* with the name "light_sensor". It finds one in its list and creates a string "light_sensor:r\n". This string is sent to the arduino via the USB connection. The arduino reads its serial input one character at a time then passes the string (with trailing newline) to the xbee module. The xbee module broadcasts this to all other xbees in the network.

The *remote modules* both receive the command in the same manner as before. The second *remote module* notes that the given id does not match its own, nor does it match any known reserved tags, and so discards the input before returning to polling its serial input. The first *remote module* matches the given id to its own, and parses the given command. Since this module only has a single sensor and a single led the protocol can be very simple. The complete list of commands that this *remote module* listens for could be as simple as:

```
r - read the light sensor, return the result as value between 0.0 and 1.0
w:int  - change the voltage on pin 4 to the value of the given int.
```

The arduino polls the connected sensor and reads a value of 0.54. It then sends back the message:

```
light_sensor:0.54\n
```

Since the arduino only has a single sensor there is no need to specify which sensor the value corresponds with. The driver will know.

At this point the driver for the *remote module* with the LED array has nearly completed it setup. Since it does not know the current state of the LED array, and the *remote module* does not keep track of its state it begins by telling its *remote module* to turn all of the LEDs off, thus resetting it to an initial state:

```
Coordinator:broadcastMessage("led_array", "allOff");
```

The *coordinator* picks up the broadcast request and sends the string `led_array:allOff\n` on the network. The *remote module* receives the command and loops through all pins connected to LEDs, setting the voltage on each to 0.

During its main loop *coordinator* notices that there is input on the serial line from the ardiuno and reads in `light_sensor:0.54\n`. The *coordinator* loops through its table of running drivers, polling each for their name and sees that there is a match for "`light_sensor`". The *coordinator* passes the message "`0.54`" to that thread for processing.

The light_sensor thread receives the input and saves the given value as a float. The light_sensor thread checks this value to see if it above 0.7, the threshold to trigger turning on the LED. The value is not high enough for the trigger, but the driver does not yet know the state of the LED, so it must send the command `light_sensor:w:0\n` to the *remote module*. The *remote module* receives the command and sets the voltage on pin 4 to 0.

The coordinator has also spawned a thread for a driver for a *remote module* that, for whatever reason, does not exist on the network. This driver "test_module" begins its setup and attempts to query its *remote module* for the value of a sensor. It passes the message `test_module:r:12\n` to the *coordinator*. The *coordinator* looks through its table of known *remote modules* before broadcasting the message. It does not see a matching entry. The coordinator may then re-query the network for attached modules, or may immediately return message `FAIL:NOMODULE\n`. The driver receives the message and is free to operate as it sees fit. It may sleep for a specified time before retrying, kill itself off immediately, or request a wakeup from the *coordinator* (it might be nice to have a scheduler a-la the Android AlarmManager built into the coordinator so that the drivers can be scheduled to wake at specific times).

Let us also assume that there is a *remote module* named "test_module2". The driver for this module is not running for whatever reason (perhaps due to an exception during execution). At some point in time the *coordinator* receives the message `test_module2:42\n`. This particular *remote module* operates on battery power and, in order to preserve power, it broadcasts on a fixed interval before sleeping for an extended time. Because of this limitation it was not awake to respond to the initial `ALL:NAME\n` request. When the *coordinator* checks the table of known drivers it does not find a match. It adds the tag `test_module2` to the list of known remote modules. Instead of ignoring the message it checks if there is a driver available. Finding one (I suppose the file name of the driver must match the tag), the *coordinator* attempts to start the driver by loading it as a new Runnable. Since the driver is newly started the message is passed as an argument to its constructor so that the driver can decide whether the message should be acted upon or discarded. This particular driver decides to discard the message, preferring to

wait until the next message is broadcast from the *remote module* before recording the value. Since the *remote module* is known to enter a deep sleep between broadcasts there is no point in sending a message asking for an updated sensor value. With the xbee on the *remote module* powered down any messages sent will fail to be received. This may be a viable option for anyone that needs to limit power use to the bare minimum.

Notes: There are a few problems with this design:

1. This design limits the ability of drivers to communicate with each other. Reading sensor values from one *sensor module* and using those values to control a separate *controller module* would be tedious. The current workaround is covered in point 2.
2. Responses from the *remote modules* are forwarded to the correct driver, but messages from the drivers are not checked to make sure the tag sent matches the sending driver. This could be a security concern since a driver "foo" could broadcast a message with a tag "bar". Any responses from the remote module "bar" would go to the correct driver, but if the module is a controller it execute all commands as if they came from the correct driver. This flaw was left in the proposal to open the possibility that a single driver could drive more than one *remote module.* This could allow a single driver to read sensor data from one remote module and use that data to control another remote module (with a different tag). Another possibility would be to give both remote modules the same tag.

TODO --- FINISH THE SECTION ON THE WEB SITE <-> COORDINATOR COMMUNICATION