Washington University in St. Louis

School of Engineering and Applied Science

Department of Computer Science and Engineering

Project Examination Committee:
Dr. Chenyang Lu, Computer Science and Engineering
Dr. Arye Nehorai, Electrical and Systems Engineering
Dr. Chris Gill, Computer Science and Engineering

Data transport system

by

Dor Rahav, MSCE

A project presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

Master of Science

Dec. 2014
Saint Louis, Missouri

Contents

ABSTRACT OF THE PROJECT

Data transport system

by

Dor Rahav

Master of Science in Computer Science

Washington University in St. Louis, Dec. 2014

Research Advisor: Dr. Chenyang Lu

To facilitate the WU Smart Home research [21] we built a system that collects data from sensors and uploads the data to the cloud. The system supports data collection from multiple locations (typically apartments) that are independent from each other, endowing the system with two benefit: distributed data collection and alleviating privacy concerns. Each location is managed by a local micro-server (μServer) that is responsible for receiving data packets from sensors and managing their transient storage. Periodically the μServer triggers a data transport process that moves the data to a cloud server where it is stored in a centralized database.

In the cloud, the receiving subsystem verifies that the data is coherent and if so, inserts it to the database for permanent storage. Errors are detected and stored in a designated table for review and correction when possible. The database is relational and referential integrity is defined over the schema to force data coherency. The system was designed to support dynamic module loading allowing system deployers to author their own sensor specific module to validate the data in any way they want.

Important features of the system are configured without code changes. For example, users can define sensor identification rules by editing a configuration file. There is full support for logging. Input-Output and other expressions that can fail are enclosed within a *try-except* clauses and an overarching error handling module controls critical events.

The system is written in Python 2.7.5 [12] and its runs on Linux (on both the μServers and the cloud server) and the embedded code is written in nesC [18] and runs on TinyOS 2.1.2 [15]. The source code is well documented; each module has introduction section, descriptive variable names and CONSTANTS are used, when appropriate inline (expression level) comments are provided. *README.txt* files are provided for each module. Excluding Python, the system uses the latest versions of every tool or OS that it rely on.

Chapter 1

Overall system design

The system we developed is designed to transport data from distributed locations into a centralized cloud server. **The system core service is reliable data transport**. In essence it is an enabler of other applications[1] (apps) that use the central database to guide their functionality and for researchers from the various academic departments in the project to perform their analysis and design their algorithms.

In the Smart Home initiative the system collects data about power-consumption and time-of-use from electric appliances connected to ACme [19, 6] power meters (such devices are often referred to as motes). Other sensors can be supported without changes in the system as long they use the IEEE 802.15.4 network [20] and the system modular design should allow for additional networks to be added with minimal coding.

To support distributed data collection we designed the system to be composed of a few subsystems: The embedded module is installed on multiple ACme motes. Within each location (apartment) these motes communicate with the apartment subsystem. Such apartment subsystem can be deployed in multiple locations. If information from the building itself or other location (e.g. environmental sensing external to the system) is of interest, then the location-subsystem can be deployed for it as well. In essence, as long as Internet connection is available nothing in the system design preclude the location-subsystem from being deployed in other sites of interest. Lastly, the cloud subsystem is installed once in the cloud or on a physical computer running Linux. In the Smart Home project we choose to implement it on Amazon EC2 [1] on which we installed an Ubuntu Linux server [17] created from Amazon Machine Image (AMI). The system high level component diagram is shown in figure 1.1.

[1] What these apps do is beyond the scope of this project.

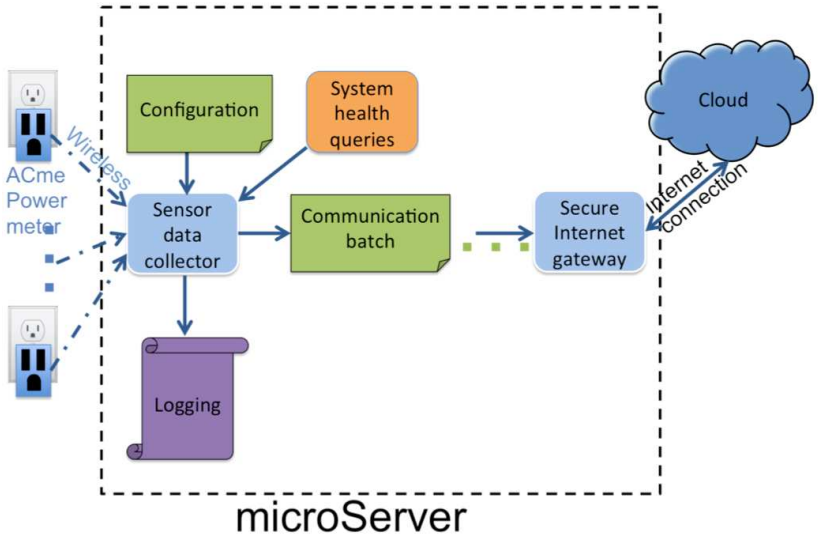

Figure 1.1: The system high level components. In each apartment, ACme power meters send usage data to a local microServer (μServer). Multiple such apartments or locations can be deployed. All apartments upload their data to a single cloud server.

Data from the wireless power meters is collated at each apartment and transported to the centralized cloud server. The apartment subsystem does not depend on building level services, or the cloud subsystem and thus inherently supports privacy between apartments or locations. To further isolate one apartment wireless data from another we specify the use of different wireless channel for each apartment. Installers of the system should appropriately perform spacial design in order to not reuse a channel in nearby apartments. With the indoors range of the 802.15.4 network being about 30 meters and the availability of 16 channels such a layout should be easily attainable. This is of no concern for the WU Smart Home project which has only six apartments.

The data in the cloud can be used by apps for any desired objective such as algorithms for smart control of the apartments, or smartphones based surveys that use the data as basis for driving their functionality. Researchers in the project can use the ready functionality of the SQL server to query the data for any purpose they may wish.

1.1 Programming languages

We used Python 2.7 for programming on the μServers and the cloud server. We made efforts to use the most modern Python expressions without resorting to programming in Python 3.x constructs[2]. We used nesC for programming the motes, and PHP for generating dynamic content for web pages.

1.2 Operating systems

TinyOS runs on the motes, Linux Darwin version when the μServers is an Apple OS-X laptop, and Ubuntu Linux server on the cloud server.

1.3 Modular design

The following sub-systems comprise the overall system.

Embedded subsystem There are two parts to this subsystem. One running on ACme motes that is responsible for activating the sensors at desired intervals and sending the data packets (by wireless network) towards the μServer. The second part is running on a BaseStation [16] mote attached to a USB port on the μServer. Its responsibility is to receive the packets and pass them up from the PHY layer.

Location subsystem The Listen [16] app of TinyOS work with BaseStation to make the packets available at the application layer. The system bundles the packets into files which are sealed with appropriate header and footer. The header is later used to identify the source of the data and the footer is used to verify data completeness. The operation of the module

[2]We used Python 2.x because we felt that 3.x might run into problems on some μServers, which are a hot new trend that is still evolving. Using the more modern constructs of 2.x should ease migration to 3.x once future developers of this system feel that sufficiently many μServers are capable of running it.

is configurable externally to the code. In desired intervals the μServer triggers an encrypted upload of new data files to the cloud server.

Cloud subsystem A Linux server we built runs in the cloud (as our cloud provider we choose Amazon EC2) and hosts the centralized modules of the system. The cloud module is responsible for parsing, verifying, and inserting the sensory data into the database.

Chapter 2

Software repository

All of the system artifacts (code, configuration files, documentations, and so on) are maintained in a private (by invitation only) git repository hosted on `https://github.com`. This cloud repository provides us with code and artifacts revision management, backup, and an excellent way of being able to pull down code to the diverse number of development environment that this project demanded (TinyOS, laptops, μServers, Cloud). The repository address is `https://github.com/rahav/SmartHome` and itcontains the following sub-projects (directories).

BaseStation Contains the original TinyOS basestation software and related toolchain such as the Java Listen app. The basestation software should be installed on a TelosB [14] (or any other compatible) mote that will be connected to the μServer. It is the combination of the basestation mote and Listen that makes the packets available to the rest of the system.

CloudServer Artifacts for the software that runs on the cloud server. Importantly you can find here: (1) createDB.txt with instructions how to create and give access rights to the centralized database; (2) DBLoader.py is the module that the server runs periodically to insert new incoming data files into the database; (3) ParseACme.py which is a specific parser for ACme packets.

Libs Artifacts and libraries that are useful for all modules of the system (for example a library that makes error handling uniform across the system).

berkeley Contains the original ACme software and our modifications to it. The modifications are identified by directory names having the .WU postfix. We kept the original source code for any future need.

uServer Contains artifacts for the system software that runs on each location μServer. Importantly PacketReceiver.py which is responsible for accepting packets from the basestation mote and collating them into files is here.

A few things are worth paying attention to:

The **ACMeterAppC.nc** in `SmartHome/berkeley/acme/apps/ACMeterApp.WU/` is the app to install on all ACme motes. This directory also holds our annotated ACme instructions. The ACme original packet receiver can be found here, but it is used only as a guidance to the ACme packet structure which was not documented elsewhere (this code is written in Perl).

The **uServer** and **BaseStation** sub-projects, together, hold the artifacts that needs to be installed on any μServer and its attached mote respectively.

As long as the directory structure suggested by the repository is maintained the Python libraries in **Libs** do not need to be copied to the directory where a Python app that use a library is running. This is achieved by adding the expression *sys.path.append("../Libs")* in any Python code that wants to use a library.

The directory structure under the **berkeley** sub-project is needed for *make* dependencies.

Chapter 3

Development environment

It turns out that in our modern world, with all of its advanced techology where is it so easy to develop a lot by composing, or meshing (interfacing) existing services – it is actually complicated to design and maintain a real system. The complexity originates from the diverse platforms which today's systems need to integrate into an orchestrated whole. In our project for example, which in essence is not large at all, we had to develop software for motes which required TinyOS. In turn TinyOS is best developed these days on Linux, which requires either a local (because motes need to be connected to it) Linux server, or a virtual Linux server running on a local machine. We also had to develop software on μServers, which we first wanted to check on our laptops. When developing the cloud subsystem we want to have reliable way of having access, from the cloud, to the artifacts developed on other platforms in the project. We see that even in this puny project we have four development environments (namely embedded devices, laptops, μServers, and cloud servers), multiple OS, a diverse set of display capabilities.

One of the main challenges is not simply to support the various hardware you develop on, but also to support the different (Integrated Development Environments) IDEs and editors. Of prime importance is to make sure that as we switch from one development environment to the other the most recent source code is being edited.

There are three decision milestones that we made and aided much of this complexity. The first was the decision to use Virtual Machines (running on our local development laptops) to any extent possible. ACme development for example was highly sensitive to how TinyOS was installed. TinyOS in and of itself was hard to install with no problems. Installing Linux on a local virtual machine, and then installing TinyOS on Linux allowed us to incrementally

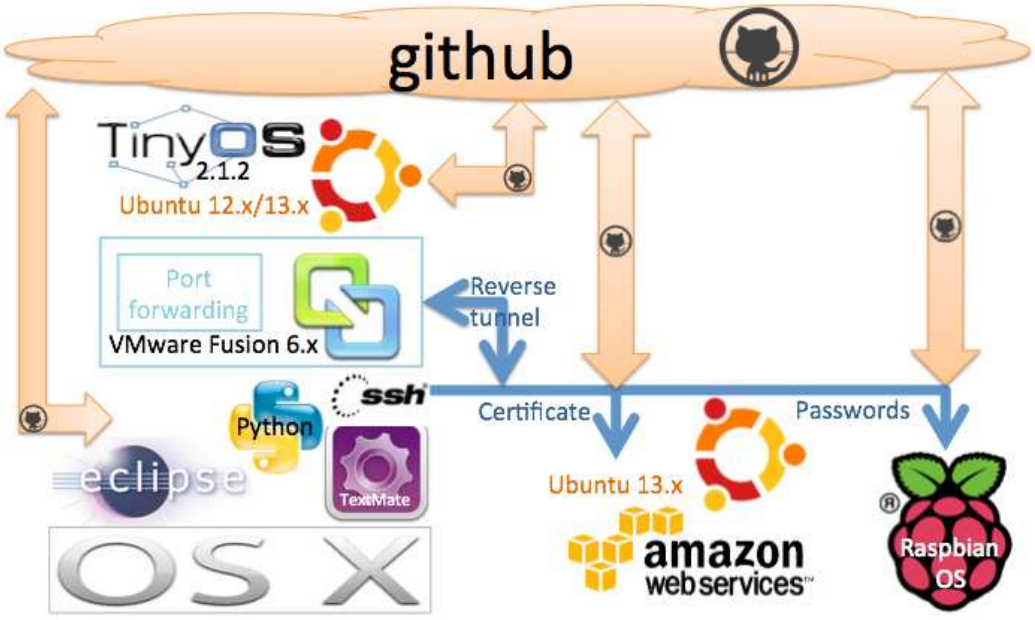

Figure 3.1: The overall system development environment support distributed development on multiple and diverse hardware platforms, using many desired IDEs and editors.

install this specific development environment, from the right sources, and the right cross-compilers. Each time we had a properly installed environment is was easy to save the virtual machine as a backup to which we could revert to if the next step introduced errors.

The second milestones was the decision to use the git repository system, which was designed for distributed development. Git made it easy to work in any of the development environment, at the end of the work session *commit* and *push* to the git cloud, and then switch to any of the other development environments and *pull* the latest code.

The last decision may seem trivial but taken together yield a well covered solution for authoring source code and other software artifacts. It is the decision of using vim (other editors will do equally well) as our editor of choice within ssh login shells, TextMate when the editing session is a bit more involved (note that TextMate also has an excellent solution for providing full WYSIWYG editing during an ssh login, as well as local editing), and Eclipse full IDE was desired. The overall development environment design is shown in figure 3.1.

Chapter 4

Added value services

To provide a complete turn-key system we also installed[3] on the server a handful of additional services:

- MySQL [7] as the research's centralized SQL database.
- OpenSSH server [9] for remote server administration and access for other needs.
- Apache web server [2] to serve web content, including dynamic content generated by PHP.
- PHP [11] for writing web server-side code.
- vim for a better editing than what the default vi provides.

[3]The best installation process we are aware of can be found in `http://research.engineering.wustl.edu/~todd/cse330/index.html`, just follow first few modules under the course calendar.

Chapter 5

Location (apartment) subsystem

The system defines a location as any physical site within which all sensors have some logical relationship to each other. In the case of the smart home each apartment is a location. There are a few artifacts in a location: One or more ACme motes send their packets to a basestation mote attached to a μServer. The basestation mote forward the packets from the PHY layer to the application layer. At the application layer (in fact at the OS shell) the Listen app consume the packets and makes them available to the rest of the system, in our case this would be PacketListener.py which bundles the packets into files that the μServer will periodically upload to the cloud.

Note that there are two parts to the embedded software, one running on ACme motes and the other running on the basestation mote. Both motes run the TinyOS operating system. The basestation mote can be any mote that has an IEEE 802.15.4 PHY and can connect to a USB port. We have tested the system with the TelosB motes.

To protect the privacy within each location (apartment) the ACme motes and basestation mote of each location are made to communicate on their own network channel. This is done by editing the *Makefile* of both apps and modifying the *nn* value on the following line before compiling and installing these apps for each location.

PFLAGS += –DCC2420_DEF_CHANNEL=*nn*

5.1 ACme software

The ACme embedded software is completely based on the code provided by the Berkeley research group that developed the ACme hardware. When needed, the original directory was copied to a new directory (in the same parent directory) and given the same basename with an added .WU postfix. In the git repository this code reside under the **berkeley** directory, which we recommend to keep in its original structure for Make dependencies. The ACme code runs on the TinyOS operating system.

The app to be installed on ACme motes is discussed in section 2. ACme provides installation instructions for this code but they are wrong in a few places places, so we recommend to use the annotated *acme_quickstart.pdf* that we saved in the repository and not the one available from `www.moteware.com` site.

5.2 Packet receiving

Receiving packets in a Wireless Sensor Network follows a common pattern. Packets are sent in a agreed upon structure that TinyOS basestations adhere to. Moreover, TinyOS provide tools to automatically generates Java or Python code to handle the incoming packets, or use ready made listeners. In this system we used the original TinyOS basestation app and the Java based Listen app with no modifications.

Listen forwards its packets using a Linux pipe to our PacketReceiver.py app, which process and bundles the packets into a file for transmission to the cloud.

5.3 Packet bundling

We wanted to separate concerns between receiving packets and communicating with the cloud. Therefore PacketReceiver core functionality is to receive packets and periodically (the period is user configurable) bundle them into a file. Asynchronous to the bundling period the μServer which runs Linux wakes up an OS processes that upload all ready files to

the cloud. It is left to the system deployers to decide whether the files are kept locally after they are successfully uploaded to the cloud. A simple change in a shell script can work one way or another. For disk space conservation purposes we recommend deleting the files after a successful upload.

A configuration file accompanies this program. All modules support this approach with the configuration file having the same basename as the app and the *.cfg* extension. In each apartment the system deployers need to define the apartment name, the desired bundling period, decide the free disk space below a warning will be issued, and whether the subsystem works in interactive mode or not. The interactive mode displays messages on the μServer monitor and allows a human operator to see what is going on.

The system logs interesting events. All modules do that with the log file having the same basename as the app and the *.log* extension. The system was designed to be as autonomous as possible and that led us to the problem of dealing with critical errors. We did not want errors to compound after a first critical error occurred. Therefore all modules use the *ErrorHandler.py* library we authored, which provides functions to prevent any work following such an error. Such a shutdown event is logged. The system maintainers can then view all the information in the log that led to this error condition, fix the root cause, and turn the system back ON. To support this re-ignition we made *ErrorHandler.py* by able to run from the command line.

Chapter 6

Cloud subsystem

The cloud subsystem is the locus point for all things research. The core service of our system is to provide data transport from multiple locations to the cloud and deposit valid data into a centralized research database.

To support high quality research data the cloud system provide coherency at various levels. The system deployers can author their own data verification modules and configure the system to use them without changing the core service of the system – data transport. A fully implemented such module, named *ParseACme.py* is provided. It processes ACme packets and should be used as a template for adding new sensors. Each packet can be identified by rules that deployers author by editing the configuration file. The database is relational and referential integrity between the tables is defined. Every batch of data packets is contained in a file and a transaction is maintained over the file. Any critical error in processing will lead to a rollback to the previously coherent state of the database. Lastly, I/O and other operations that can fail are enclosed in a *try-except* clause and critical errors will cause the system to stop further processing to avoid compounding errors. Processing errors that the system categorize as possibly recoverable do not trigger the transaction rollback and are inserted into an *reading_error* table from which they may be examined and moved to the *reading* table after attending to the error; In this case the valid packets from this file will be inserted to *reading* table.

The cloud provider we choose for this project is Amazon EC2. The instance is named *SmartHome* and its instance ID is *i-97331db7*. We used EC2 Elastic IP (an EC2 service for keeping a permanent IP address). The server address is `54.83.14.180`. The firewall is provided by EC2 as part of their hosting, for which we defined the *WUSTL Green Home*

security group. We configured the security group to allow SSH (port 22), HTTP (port 80), and MySQL remote access (port 3306). Other than that all ports are closed to reduce the risk of malicious attacks.

On the cloud space provided by EC2 we installed a Ubuntu 64 bit Linux server version 13.x (Ubuntu codename saucy). The source for this OS is an Amazon AMI. On this OS we installed SSH to allow remote administration and access to the server, VIM for easier (that VI) editing, Apache to enable any researcher access to the system status[4]. MySQL version 5.x is the database server we installed; it is the permanent store of all of the sensory data send from all locations. For additional security we configured the SQL server to not accept anonymous users. In this project PHP is our language of choice for responding to web service requests, so PHP version 5.x was installed.

Additional utilities that were installed is the zip package, gedit to support some developers that prefer that over vim, and apparmor-utils which eventually was not needed.

6.1 Cloud sub-system architecture

The architecture is shown in figure 6.1. Data files arrive to the cloud server asynchronously from the various locations. They are deposited in a designated directory that can either be protected in a Linux jail or other restricted access schemes, for example the μServers can be given a restricted certificate to access the server, limiting their ssh access to a particular directory and precluding them from establishing a login shell. Other schemes are also possible and is left to the preference of the implementors. Periodically a cron process wakes up *DBLoader.py* module which will check if files are ready in the incoming data directory. When files exist this module will process them according to the configuration parameters and log interesting events. Files with packets that adhere to all of the coherency rules will be inserted into the *reading* table in the database. Packets parsed with errors or any other malfunction, but does not present a critical error will be inserted into the *reading_error* table. Critical errors will cause all inserts from this file to the database to rollback. The system

[4]What web pages or services will be authored is beyond the scope of this project, our scope was to enable such provision (in fact there was already another MS project that used the centralized database over apache to provide access to the database and sensors status).

will stop processing further files until the root cause error is solved. While processing, the system can check the general health of the server itself by invoking Linux system calls. For example we already implemented a service that monitors if the server has sufficient disk space because the storage requirements are expected to grow and we wanted to be judicious with our EC2 expense.

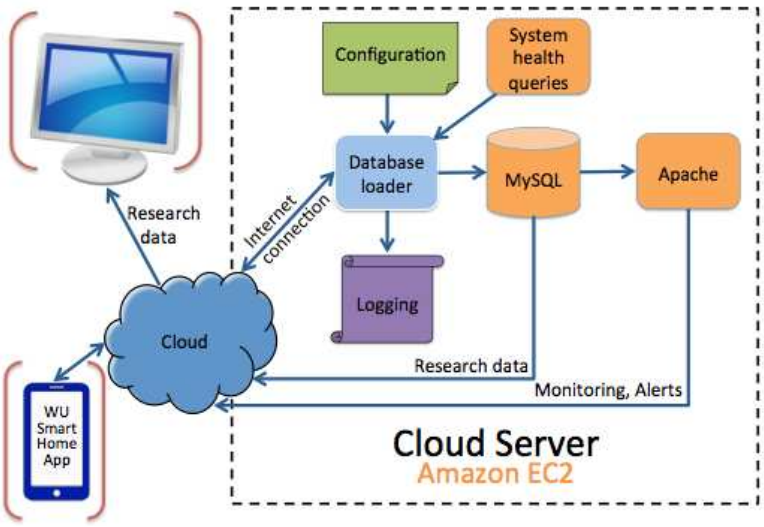

Figure 6.1: The cloud system components (the red brackets denote apps that can be developed but are beyond the scope of this project). Data arrives to EC2 from distributed μServers over the Internet cloud. The subsystem parse the data, verifies that it is coherent, and insert it to the MySQL database.

6.2 Database

MySQL database server version 5.5 was installed on the server. Access from Python is achieved by using the Connector/Python [8] by Oracle. Moreover, to further simplify access to the database we developed a Library with common idioms, which can interchangeably be used along with using direct calls to Connector/Python.

The database schema is designed to be fully rational, but in addition (with this being a school project and wanting to support new SQL developers) a single, non-normalized table is maintained. Its name is *reading_nn_acme* and it stores only valid readings. In all tables appropriate column types were used and only reasonable values are allowed at the schema

level (so future coders are forced to write appropriate code). Referential integrity is defined over the tables and the relationships between them. Security wise users and applications are restricted to access only the tables they need and are allowed to do only the operations that are reasonable for their role. The database Entity Relational Diagram (ERD) is shown in figure 6.2.

6.3 Server added-value services

Apache, via PHP, can access the database for any desired purpose, such as editing the database tables via web UI, developing an app for viewing processing errors, or give researchers status about the system conditions. Such apps were beyond the scope of this project, but another MS student already developed a very nice system `http://54.83.14.180/SmartHome/` for accessing the database to configure buildings, apartments, install and manage sensors, as well as see if the sensors are working properly by determining if their valid sensory data arrives to the server.

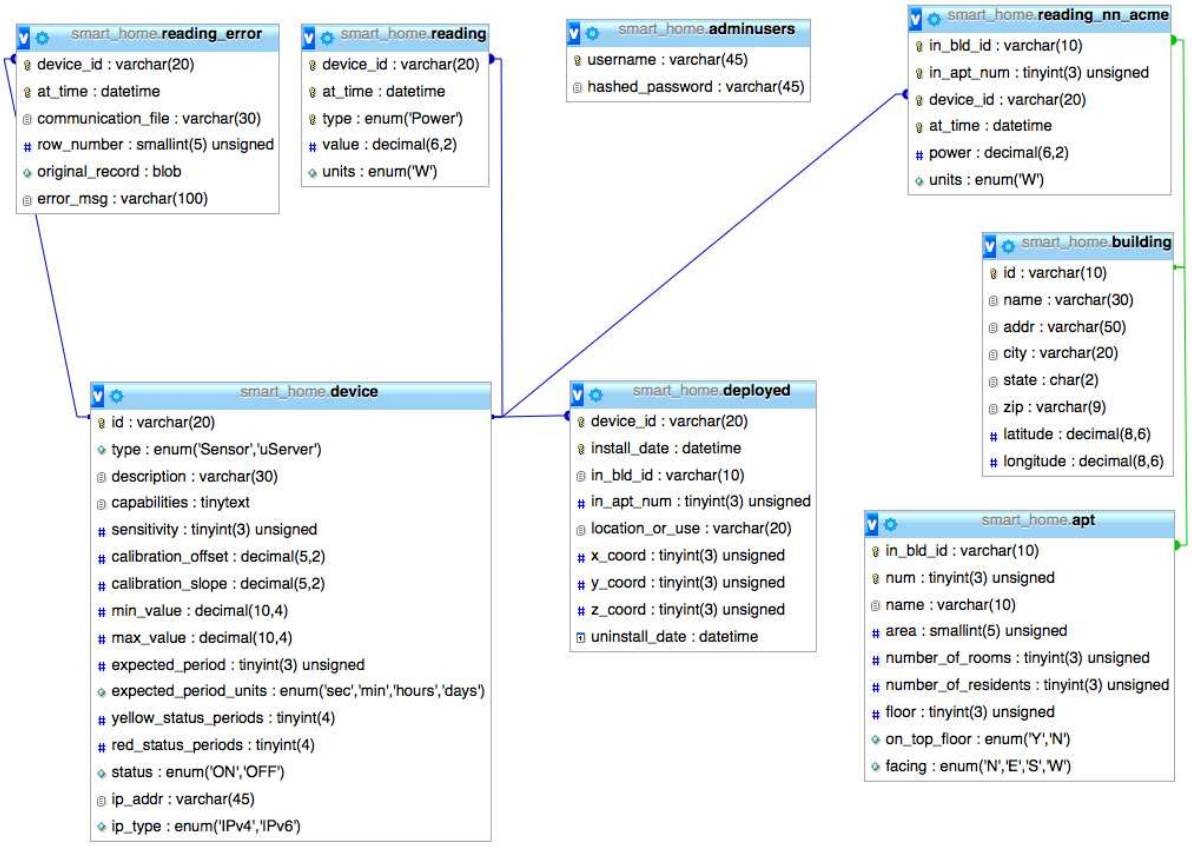

Figure 6.2: Entity Relational Diagram (ERD) for the research project database. The *building* table gathers one or more apartments defined in the *appt* table. Each sensor is defined once in the *device* table and then can be installed by using the *deployed* table, a relationships that allow the system maintainers to move sensors from location to location while keeping their audit trail. Deployed sensors send sensory data that is stored in the *reading* table if it is valid; otherwise it is stored in the *reading_error* table. The *reading_nn_acme* is a None-Normalized table that store only valid ACme packets and can be used by novice programmers, while still having access to all valid sensory data.

Chapter 7

Future work

In this last chapter we summarize what could be the next steps of the system. Starting our discussion at the μServer, the current system was deployed on a laptop and it should be deployed on a Raspberry Pi [13] class of μServers. While the system itself would not change this could be a nice project for an undergraduate independent study, taking an holistic approach of deploying an end-to-end system. The key tasks will be: considering how Linux scripting is kept generic across operating systems, possible issues of the TinyOS Listen app, buffering compatibility when piping between Java (Listen app) and Python (PacketReceiver app), and configuring Linux schedulers. As part of this project we also recommend that once Python 3.x is verified to work well on this class of μServers (e.g. the Raspberry Pi mentioned above, BeagleBoard [4], CubieBoard [5], pcDuino [10]) the switch to Python 3.x be made across the system.

There could be benefits to replacing the Listen app with SerialForwarder. The subsystem running on the μServer could be made to run as a daemon and listen on a port instead of getting its input by Linux pipe. When using ports more than one module can consume the packets.

Completely corrupted packets (often these are not even packets, just error messages) are currently not sent to the cloud. With the core service of the system being reliable transport of well-formed packets, we did not pay much attention to how many packets are not even formed. Currently, visibility into how often malformed packets occur can be calculated by counting how many packets are missing from the centralized database and knowing a priory the expected sensor period. It may be worth while to consider an improvement to how this kind of errors are accounted for.

The infrastructure that flows alerts to the Twitter module is in place but messages are being printed instead of twitted. It should be an small effort to activate a Twitter account and Tweet the message instead of printing it to the console.

Currently the cloud server enjoys a static IP address by using EC2 Elastic IP service. It would be nicer to give it a domain name.

In development we did not need to backup the cloud database as our important artifacts where source code related, and their backup is likely taken care of by git. We do recommend that backup of the project data will be configured in production deployments of the system.

Finally, since the system development started, Ubuntu version 14.x was released and the next system administrator should consider switching to this release.

References

[1] Amazon elastic compute cloud (amazon EC2). http://aws.amazon.com/ec2/, July 2014.

[2] Apache. https://httpd.apache.org/, July 2014.

[3] BaseStation and net.tinyos.tools.Listen, July 2014.

[4] Beagleboard embedded computer. http://beagleboard.org/, July 2014.

[5] Cubieboard a series of open source hardware. http://cubieboard.org/, July 2014.

[6] moteware. moteware.com, July 2014.

[7] MySQL. http://www.mysql.com/, July 2014.

[8] Mysql connector/python developer guide. http://dev.mysql.com/doc/connector-python/en/index.html, July 2014.

[9] Openssh. http://www.openssh.com/, July 2014.

[10] pcDuino – Mini PC + Arduino. http://www.pcduino.com/, July 2014.

[11] php. http://php.net/, July 2014.

[12] Python programming language. https://www.python.org/, July 2014.

[13] Raspberry Pi. http://www.raspberrypi.org/resources/learn/, July 2014.

[14] Telosb by memsic. www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf, July 2014.

[15] *TinyOS*, 2.1.2 edition, June 2014.

[16] TinyOS BaseStation, July 2014.

[17] Ubuntu 13.10 (saucy). http://releases.ubuntu.com/saucy/, July 2014.

[18] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, number 11 in PLDI '03, pages 1 – 11, New York, NY, USA, 2003. ACM.

[19] Jiang, Xiaofan and Dawson-Haggerty, Stephen and Dutta, Prabal and Culler, David. Design and Implementation of a High-fidelity AC Metering Network. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, number 12 in IPSN '09, pages 253–264, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Pat Kinney and Phil Jamieson. IEEE 802.15 WPAN task group 4 (TG4). http://www.ieee802.org/15/pub/TG4.html, July 2014.

[21] Arye Nehorai. Energy consumption profiling and occupant behavior learning for efficient energy use in buildings. https://icares.wustl.edu/research/Documents/Nehorai_Abstract.pdf, April 2013.