CLM AS A SMART HOME MIDDLEWARE

By

JAMES KUSZNIR

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTERS OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2010

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of JAMES KUSZNIR find it satisfactory and recommend that it be accepted.

_____
Diane Cook, Chair

_____
Dave Bakken

_____
Carl Hauser

CLM AS A SMART HOME MIDDLEWARE

Abstract

by James Kusznir M.S.C.S.
Washington State University
May 2010

Chair: Diane Cook

This work details the design of middleware for smart environments that is lightweight, flexible, fast, and easily extensible. We hypothesize that this goal can be met by leveraging the power and simplicity of eXtensible Messaging and Presence Protocol (XMPP) through a publish-subscribe design paradigm. In this thesis we provide evidence to support this hypothesis by implementing PubSub-based middleware for the Center for Advanced Studies in Adaptive Systems (CASAS) smart environment software architecture. We detail the requirements for our middleware and describe how design decisions were made to meet these requirements. We demonstrate how this middleware, which we call "CASAS Lightweight Middleware (CLM)", was successfully used to integrate multiple heterogeneous software and hardware components in CASAS.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

1 INTRODUCTION

1.1 Background

Modern smart homes use a collection of sensors, processors, and control devices to allow a home or other space to interact with the user in a productive manor. This has been a hot area of research for some time, yet designing integrated smart homes that are easy to deploy still eludes researchers. One of the most common problems with these smart homes is the integration of the many varied components that support the smart home. Usually, a smart home consists of at least a few purchased hardware systems from different vendors and often some special-design hardware that must all inter-operate with one or more pieces of custom software to achieve the goals. Yet, there is no ready-made common language, protocol, or other means of plugging them in and communicating.

The smart home research field is far from new. Some claim that the endeavor to make homes smarter started with the field of Engineering; ever since, engineers and inventors have been working on ways to make life easier by making the home smarter. Now, it is possible for someone to contract with a firm to retrofit or build "smarts" into their new home. Such firms exist throughout the world, and more are popping up every day. Furthermore, more and more companies are making smart home hardware; anyone can go to websites such as www.smarthome.com and buy hardware to automate their home. It appears that smart homes are well outside the research arena now; we have arrived.

And yet, smart homes are still a major focus of research. Now, decades later, the same problems keep coming up even as we attempt to forge forward and address new challenges. One of the largest challenges remains the smart home central communications or integration system. For smaller, more focused deployments, this is less of a problem. For example, smarthome.com [1] sells Insteon [2] products, which have their own communication infrastructure (Power Line Carrier). As long as you use only Insteon or similar products, the communication/integration problem is solved. Unfortunately, it does not take long to outgrow Insteon product offerings. Furthermore,

Insteon is focused on control; there are few sensors. This requires the addition of at least a second infrastructure and vendor to sense the environment using sensors not available in the Insteon product line. Most smart home deployments will probably require a third, or even a fourth to get the full complement of sensing, control, and logging required. Now there is a problem: integrating these platforms so they can inter-operate.

This problem is not unique; nearly every research smart home encounters the same problem. When a smart home reaches a level of complexity in which multiple sensor and control infrastructures are used, a communication and integration infrastructure must be created. These days, the most common approach is the use of a "middleware", computer software that connects software components or applications [3]. Usually middleware is used to mask the differences in systems. For example, look at a large corporate or government purchasing problem. If you want to buy something, you first have to identify a source or vendor for the item, then determine how to communicate with that vendor and order the item. Each vendor uses a different procedure, requiring differing amounts of information, and supplying different contact information. Furthermore, many institutions have internal requirements for documenting and ensuring adherence to applicable regulations. To ease the pain this inflicts on users within the institution, many large institutions have a dedicated department for purchasing. Employees at this institution would use the institution's standard request method to request the procurement of an item. The staff in the purchasing department find the item, and do whatever they need to purchase it, or return a failure if it is not possible to do so. When the item arrives, the employee need not even know where the item was procured.

In the smart home world, middleware would provide a common means of communicating between all the components of the smart home. Each different smart home component would have its own middleware agent or interface that would serve as an adapter of sorts between that component's communication protocol and the standard interface decided for that installation (See Figure 1.1). This allows each component of the smart home to speak to other components, or to have a central "brain" of the smart home get data to and from all the components in the smart

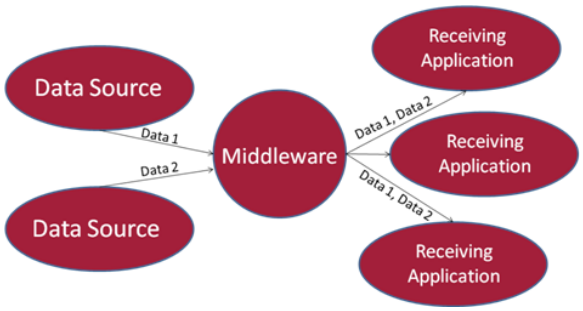

Figure 1.1: Where middleware fits.

home.

Our goal is to design middleware for smart environments that is lightweight, flexible, fast, and easily extensible. We hypothesize that this goal can be met by leveraging the power and simplicity of eXtensible Messaging and Presence Protocol (XMPP) through a publish-subscribe design paradigm. In this thesis we provide evidence to support this hypothesis by implementing PubSub-based middleware for the Center for Advanced Studies in Adaptive Systems (CASAS) smart environment software architecture. We detail the requirements for our middleware and describe how design decisions were made to meet these requirements. We demonstrate how this middleware, which we call "CASAS Lightweight Middleware (CLM)", was successfully used to integrate multiple heterogeneous software and hardware components in CASAS. We will also provide empirical analysis of various performance aspects of the CLM middleware.

1.2 Related Work

Unfortunately, few smart home projects publish details about their middleware or communication infrastructure. While this frustrates surveys of smart home middleware approaches across projects, it makes sense. Good middleware serves to simplify a project and enable focus on the projects' main goals. As such, researchers focus on their smart home projects, not their middleware. Below are some projects that at least mention some detail from which it is possible to extrapolate their approach.

The University of Deusto SMARTLAB [4] utilizes OSGi as their middleware framework. This technology is middleware framework that aids in creating the actual middleware. The authors of the middleware must code in their application logic, utilizing features from the framework to handle much of the actual communications. OSGi [5] is a Java-specific implementation that has no intrinsic distributed communication. As such, all entities of the smart home must live on a single machine, be written in Java, and run under a single instance of the JVM. The OSGi spec does provide for gateways into other communications mechanisms, such as HTTP, but at that point one leaves the OSGi realm, which results in a much smaller subset of functionality.

UTA's MavHome [6], like many other smart homes, uses CORBA [7]. CORBA is perhaps the best-known middleware framework. It is well established, and has implementations in most modern languages. CORBA was designed from the ground up to provide a distributed communications infrastructure, so it easily supports communication across computers over a variety of networks (although most development efforts are focused on IP). It has a rich feature set, and a number of vendors and open source project implementations. All of this make it an attractive middleware framework for many projects, including smart homes. Like all middleware frameworks, CORBA provides tools to aid the developer in building their middleware. SOAP (Simple Object Access Protocol) [8] is a similar technology to CORBA. It aims to provide a means of making remote calls to objects on remote servers and returning the results. Unlike CORBA, these messages are in XML, which is a plain-text encoding with all data marked up with labels. In addition, SOAP does not specify a transport, but often is either pure RPC (Remote Procedure Call) or HTTP (Hyper-Text Transport Protocol, what web servers use for web page serving). SOAP is not well suited to smart homes, as it is a heavyweight, tightly coupled system much like CORBA, where methods on remote objects are called and results returned.

Ritsau's work [9] focuses on a different approach, one similar to that which we take in our environment. In his work, they design a publish-subscribe framework. Doing this results in middleware that is lightweight, as it simply passes messages. It also decouples the agents from each

other, allowing more flexibility in the overall environment – pieces of the environment may be replaced without having to retrofit the entire environment. In all, publish-subscribe middleware are well suited for smart homes. However, Ritsau's work does not appear to be as flexible, lightweight, or extensible as CLM.

Moving data around between the various components of the smart home is the largest problem and need of smart homes, but it only solves part of the problem. In order to move data between components of the smart home, one needs to know what components are there and what data they need or have available. Early and basic smart homes hard-coded this information; the components were told by programmers what was available and how to find other components. Unfortunately, as smart homes grow, this becomes less and less maintainable. Today's smart homes are often far more complex with many more components, with new components coming online and old ones being retired. Keeping up with this can rapidly become a programmer's full-time job.

To solve this, modern smart home middleware designs have put significant effort into various methods of finding and coordinating components dynamically. Done well, this would allow the smart home maintainers to replace, retire, or add components to the smart home without having to modify the unchanged components. In fact, some even allow this to happen without shutting down or restarting the smart home; when the new component comes on-line, it announces its presence and negotiates with other components to get or provide data.

Many of the current middleware frameworks have extensions or support for this features. For example, Java has the Jini Framework [10], which has been extended with OSGi. Others have created uPnP [11], and most major distributed system architectures have their own. In addition, there are many standalone software packages that have been created to attempt to do the same thing a different way: make an easy way to locate and integrate services on different computers (or processes on the same computer).

1.3 The CASAS Environment

We test our hypothesis via an implementation centered around the context of the CASAS smart home environment [12]. The goal of the CASAS project is to design smart environments that act as intelligent agents [13], perceiving the state of the environment using sensors and acting on the environment. Current CASAS applications discover patterns in resident behavior, predict upcoming sensor events, recognize activity, and automate manual interaction with the environment [12, 14].

The current CASAS environment consists primarily of motion sensors connected via a Dallas OneWire bus [15] (Figures 1.2, 1.3). In the CASAS smart homes, we deploy a large number of motion detectors modified to connect to the one wire bus, which is bridged into our middleware via the OneWireAgent. CASAS utilizes some additional sensors on the OneWire bus such as door open/close sensors, item coasters, analog measurement sensors for measuring light, water flow, or the like, and temperature sensors. Some other sensors are either stand alone sensors connected through some custom way (serial, USB, nor network typically), or have their own bus such as Insteon, our brand of addressable power switches. In any case, each sensor interface has an associated agent that encapsulates any driver and sensor specific knowledge (Figure 1.4). The sensor agent is responsible for running that sensor infrastructure and passing the information on as messages to the middleware. Some of these sensor infrastructures are more than just sensors; devices such as our addressable power switches allow the control of lights or other powered devices. In this case, the agent for that sensor infrastructure is also responsible for managing the control functions of that infrastructure.

To integrate all these sensors, actuators, and consumers clearly calls for middleware. As previously presented, many smart homes before this one have encountered and solved the problem; however, most of these are large and complex, and optimized for their specific smart home project, and not conducive to use in the CASAS project due to their complexity. As a result, we set out to

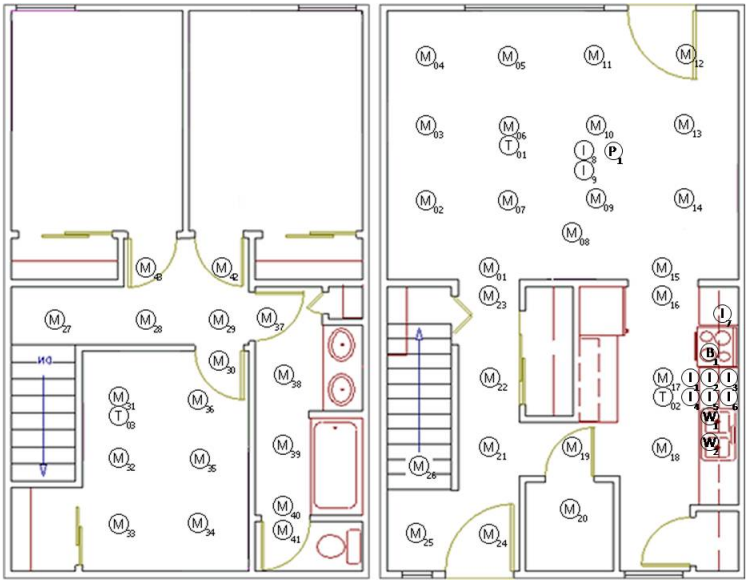

Figure 1.2: Diagram of our first living environment and sensor locations. Circles with M inside them represent a ceiling-mounted motion detector, T's are temperature sensors, and I are item sensors. The subscript refers to its local id number.

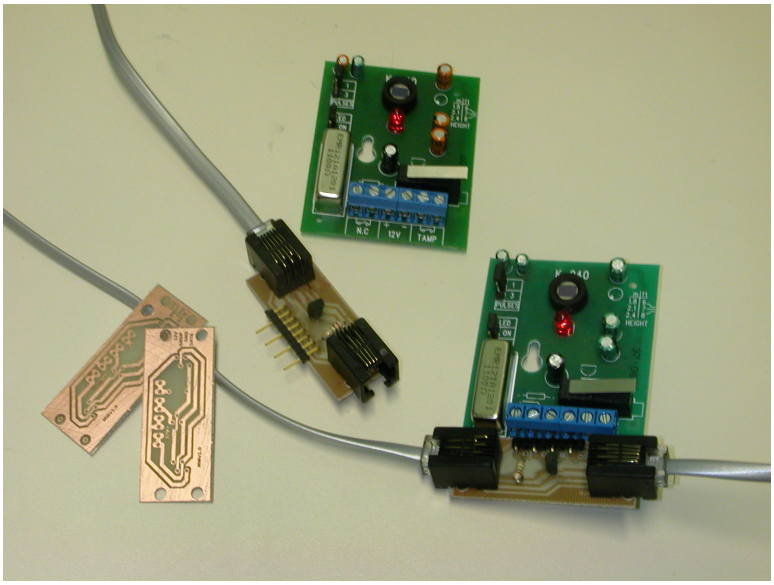

Figure 1.3: Early prototype OneWire converter board with production motion detector boards.

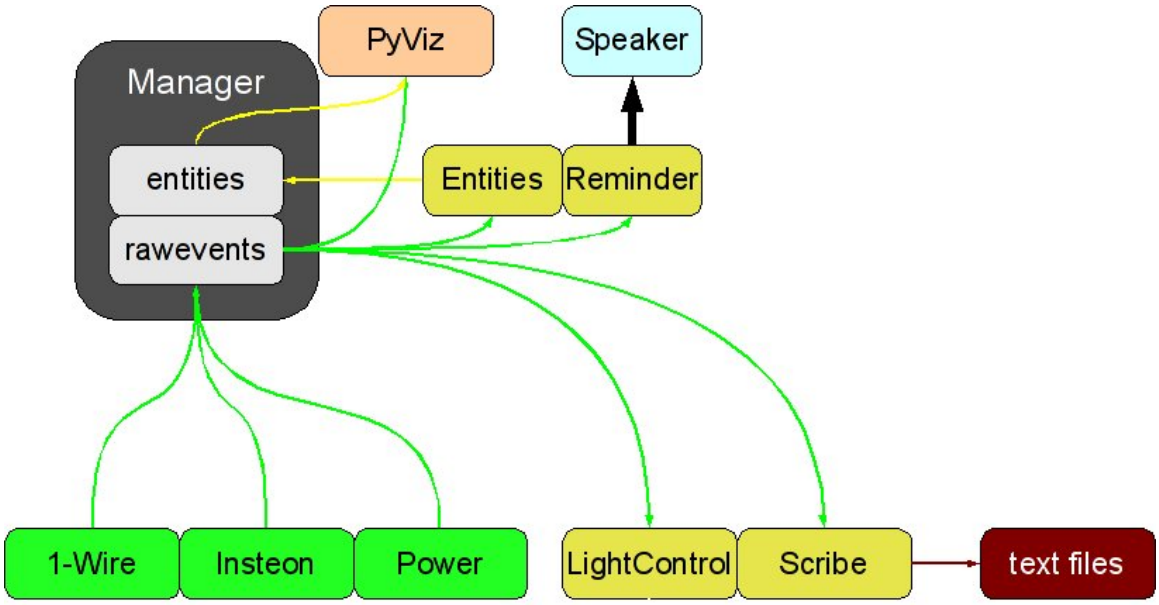

Figure 1.4: Diagram of one of our sites showing all of its agents. Note that the Entities AI agent publishes data back for other agents to use. PyViz is our smart home visualizer. More information on these agents will be given later.

create a simple middleware to meet the needs of CASAS, which are identified in Table 1.1.

1.4 Middleware Design Requirements

Occam's Razor [16] states, in essence, that the simplest solution is often the best. When applied to the CASAS project, Occam's Razor supports the removal of the many layers of indirection and protocol conversions, point to point connections, and other non-essential features from the existing middleware presented, replacing them with a simple, lightweight solution. This solution has few, if any, special conditions or assumptions; there is one set of rules that applies to all project components, both known and future. Throughout the design, the mantra "Simpler is Better" was sung; every rule was carefully pondered for its necessity; anything not explicitly required was thrown out. Out of this came CLM, which has proven through use to be solid and flexible, extensible, and still very simple.

Some of the naturally evident features of middleware designed in this way include lightweight

Table 1.1 CLM's core goals

- Simple
- Light Weight
- Capable of accommodating both known and unknown data
- Language and operating system independent
- Easy for current and future researchers, designers, and users to integrate into their own design and implementation
- Highly reliable
- Maintainable over the long term

(so we can run it on embedded computers as required), easily extensible to add devices and device types without requiring middleware expertise, capable of delivering messages with little delay to multiple subscribers, and secureable without difficulty. In addition, all this is ideally provided through a single, unified interface and simple data delivery middleware that hides the differences of the various data sources. Furthermore, such an approach would also not be platform or language specific.

For example, in the CASAS application we have a few data sources (sensor hardware), and several data sinks (Online AI agents, data archives, and state visualizers). The middleware serves as the central connector. Another important advantage of a configuration such as this comes into play if we should desire that one of the consumers start publishing its own data to other software components (such as an AI agent that is producing useful conclusions about what is happening in the space). These data may be of interest to other AI components. By using middleware with a unified interface, it becomes simple for another data source (the AI agent) to be integrated and other AI agents and data consumers to begin receiving that information.

Before getting into more details about our project, let us examine the specific requirements of our application with regard to how the requirements map into the distributed computing space. The

smart home problem requires an event passing infrastructure. In their simplest form, events consist of an originator (such as a sensor) and a message. For example, Motion-19 (sensor) ON (message). In this form, nearly any sensor type can be accommodated, as the sensor ID and Message fields have no specific format requirements. The middleware need only receive and re-transmit this data to all interested entities.

Middleware that meets the specific requirements described in this section provides the necessary information while still remaining small and accommodating future data sources. We have data sources (sensor data, for example) and data sinks (the historian and AI algorithms). Since the message is a free-form text field, this allows great future flexibility and removes any requirement from the middleware to know / decode the message's content. By employing publish-subscribe middleware to implement the data dissemination software, we have an interconnect capable of allowing the smart home components to inter-operate.

2 DESIGN GOALS AND DECISIONS

The CASAS project needs a means by which to pass messages between parts of the smart home. By breaking the project into parts, it allows for much greater ease in development. This allows different researchers to work on their piece of the project without having to tightly coordinate with others. It also (if implemented correctly) would allow for restarting pieces of the smart home (or tolerating crashes of pieces) without taking down the entire smart home. In addition, it must be possible for the modules to run on physically separate computers. In some cases this will be for computational power reasons (some AI algorithms can be very computationally-intense), in other cases the information may just be needed somewhere else (such as various state visualizers).

2.1 Modules

We have broken the overall infrastructure into modules (agents) plus middleware "hub". These agents may be publishers, subscribers, or both. Publishers have information to provide to others, subscribers consume that information. It is possible for an entity to both publish and subscribe to information. The middleware is the piece that connects the publishers and subscribers. By dividing in this way, all information specific to a task or sensor is contained in a single component, with communication between components happening in a standardized way. In software engineering, this is referred to as cohesion: all pieces of software associated to a responsibility are together in a single module. Our standardized communication protocol allows modules to be worked on individually, and does not require any specific knowledge of the other modules – they all adhere to a communication specification. This is referred to in software engineering as "low coupling". Modern software engineering practices stress that good design accomplishes both of these principals and makes for a far more manageable project [17], [18].

In the basic case, the CASAS minimal environment currently consists of at least one publisher (a sensor agent), and at least one subscriber, usually a data archiver. This provides a data source

and a data consumer. CLM can accommodate multiple additional data sources, consumers, or even combination source/consumers. Having this modular flexibility allows us to bring online any combination of sources and consumers as required, such as launching a site visualizer on a laptop to show real-time data coming through in a human-readable manner or updating and restarting an agent. It also allows us to easily customize a site deployment.

To date, all of the CASAS deployments have had a One Wire agent, principally for motion detectors (but also for door sensors, cabinet sensors, item presence sensors, and some custom analog sensors), and a data archiver to permanently record all the collected data. Several sites have also utilized the Insteon power line controllers for lighting control (data source/sync, as it is both a sensor and a control output), and a power usage meter. CASAS will frequently use visualizers to display data in real-time intermittently at the sites. Also, it is not uncommon for a site to have a requirement to run a custom agent of some sort.

In a well-done modular implementation, any application logic specific to a particular smart home component (such as OneWire – see chapter 5) will be handled entirely within the module for that component. The core of the middleware will not need to know anything about the actual source of the data, other than simply tagging it with where the data originated. The module for the component will completely encapsulate that subsystem, handling all requirements for it and presenting to the middleware a standard interface. This is the true power of the CLM design.

With a well-designed middleware infrastructure, the possibilities for expansion are limitless. We envision that using the CLM middleware, AI researchers can implement machine learning or decision making agents as subscribers (getting raw data from sensors), and publishing their decisions if appropriate. Sensor or network designers can integrate new data sources as publishers and even design bidirectional user interfaces as appropriate.

2.2 Message Format

As this is message passing middleware, a standard message type and format needs to be selected. There are many options, all with trade-offs. For example, one option is to invent our own binary structure for messages. This would allow for very small messages that could be quickly processed and extracted. However, using binary formats would require a completely custom implementation of the marshaling and unmarshaling technology (algorithm to convert binary data, especially container objects, into a format that can be transmitted over a communication link to other software components), parsers, and possibly wire formats. Such a software project is likely to be programmer-labor intensive and require a lot of effort (comparatively) to move between languages and platforms.

Custom structures or binary formats in general have a serious limitation: the need for a "decoder" to see what's going on. This is especially true during debugging. In order to investigate what was actually sent or why messages are not being properly handled, one would need to spend some time investigating the message in a general-purpose dump-style debugger where one can attempt to count bytes and decode what is in what part of the structure. Alternatively, special debugging tools can be made to attempt to do this automatically (although they too may need to be debugged using the manual method).

Custom structures and binary formats do have the advantage of being much more compact. In a well-designed wire format, only the bare minimum of data required to convey the intended information need be transmitted. Unfortunately, this also results in making a less-flexible format that is likely to need to be changed or redefined more frequently as the requirements of the middleware evolve with the research project. In our case, we have plentiful bandwidth and instead want something easier to debug and process by humans.

Using a plain-text message format fixes this problem: any network capture or standard debugger will clearly show the message. No custom tools need to be created. In addition, it makes it

significantly easier for researchers to construct messages by hand to inject into the middleware for testing. Furthermore, many libraries for text parsing exist; these may be leveraged to further reduce programmer effort.

Based on this, CLM uses XML as its message format. XML is a plain text format that is very easy to parse (and nearly every modern computer language has an XML library). Its text tags make it relatively easy for humans to extract the data. Perhaps most importantly, any XML program compliant to the XML specs will simply ignore any XML tag it does not understand. This allows for future growth and change of the communication protocol without having to alter the middleware or older software modules. Finally, XML is very open and flexible; any project or application of XML simply defines in its schema what XML tags will be used, and this can be used as the communications specification.

2.3 XML Schema

Keeping to the mantra "Simpler is better", the spec needs to include as little as possible. With XML, this is easy, as it allows agents to include additional information as required, and the middleware will pass it along without needing to understand the additional information. The important part of the spec therefore was the minimum that was required for a useful sensor message; any less than this would not meet the needs of the space. Table 2.1 captures what the CASAS personnel identified, which should generalize to other spaces, as the bare minimum to describe an event.

Our XML message format can be seen in Figures 2.2 and 2.3. This was derived directly from the requirements in Table 2.1. Due to the flexibility of XML, additional XML tags may be included, and they will be passed to the middleware; however, the above represents all of the XML tags currently known and expected in messages. Agents that do not know about those tags will simply ignore them. The main difference between Figure 2.2 and 2.3 is with time. Each message that passes through the manager has an XML field epoch added or replaced. In order to ensure that all events are properly orderable by their timestamp, we decided to use a single clock to timestamp

Table 2.1 Minimum requirements to describe an event

1. Date/Time in computer-processable format
2. Device ID
3. Location (device ID mapping)
4. Message type
5. Message
6. Originating agent

Table 2.2 XML Message from an agent to the manager.

```
<publish>
  <channel>rawevents</channel>
  <data>
    <event>
      <serial>12C67946000000DB</serial>
      <location>M003</location>
      <message>OFF</message>
      <type>motion</type>
      <by>OneWireAgent</by>
      <category>entity</category>
    </event>
  </data>
</publish>
```

Table 2.3 XML Message from the manager to all subscribers.

```
<publish>
  <channel>rawevents</channel>
  <data>
    <event>
     <serial>12C67946000000DB</serial>
      <location>M003</location>
      <message>OFF</message>
      <epoch>1271192051.34</epoch>
      <type>motion</type>
      <by>OneWireAgent</by>
      <category>entity</category>
      </event>
  </data>
</publish>
```

all messages. This eliminates the need to ensure clocks are synchronized across the system, as this has caused us problems in the past. Publishers (such as the sensor agents) are free to timestamp the event themselves (due to the requirements of XML), but the middleware will add or replace the field <epoch> with a timestamp of when the middleware processed it. This decision was influenced by specific characteristics of the CLM environment: all of our sensor agents have good responsiveness and are able to pass events to the middleware with low latency and in order, and most of our sensors are embedded systems with no clocks. As such, it was most reliable to timestamp messages in the middleware core. Other sensor infrastructures may not provide the timeliness this would require, but this requirement is easy to alter or remove.

Representing time in a message format presents additional challenges. A simple method of representing time in the message is to use a human-readable format such as `YYMMDDhhmmss` in keeping with the goal of easily debugable human readable message format. Unfortunately, this results in "gaps" in time. For example 081231235959 is followed by 090101000000, which represents a "gap" of 8869764041. These time gaps can in turn create a number of data processing problems. This gap problem began to cause a number of data processing problems. For this reason,

CLM makes use of the UNIX epoch time format, which is the number of seconds since "the epoch". This format does not contain any gaps, so it may easily be graphed and processed as linear events. While this is a compromise on our human-readable standard, it is standard and common enough that every language we have used to date has had built-in functions for converting between epoch and human readable formats. In addition, there are numerous pre-existing standalone tools for performing the conversion if necessary during debugging.

Every device, entity, or agent must have a globally-unique ID. We call this a "device ID". Most sensor types provide a unique sensor ID; on the OneWire sensors, this is a guaranteed globally unique ID number factory-lasered into each OneWire component as its manufactured. Insteon also provides its own device IDs. In the case that a new sensor type is introduced where the individual sensor does not provide a unique ID, it will be the job of the sensor's agent to create / assign a unique ID for it. The format of the IDs are quite different, but this is unimportant to the middleware; the ID is simply handled as a text string that must be unique for each sensor, with string comparisons done as required.

Similar to device ID, the spec also includes "location". This is a mapping from a physical location to a logical position, which is typically provided as a lookup table by the publishing agent. This mapping allows for sensor failure and replacement on sensors with a non-settable ID, such as our OneWire sensors. If this is not appropriate (a device that uses a user-settable ID as its unique ID), the agent for that device can set both the ID and location field the same. However, it may still be preferred to show a difference in ID if the sensor is changed, as this provides a historic record that can be used to explain any differences in the sensor or its calibration.

In a smart home where AI agents are watching and acting on a space, it is important to capture the reason for the message. CASAS has several sensor types that are tripped exclusively by humans (motion detectors, door sensors, etc). Yet smart homes are likely to have other sensed values that are just reported periodically (such as ambient temperature or light level). While humans can affect the temperature or light level, they usually did not directly cause those messages to be generated.

A human may have opened up the blinds letting in more light, but that may also have been the sun rising. This is different than a motion event, which means a human or other entity is moving at that location at that instant (i.e., the sensor event directly represents a human or other entity behavior). Furthermore, the AI agents may be controlling things in the space, such as lights. It is important for other AI entities (or lab personnel analyzing data) to know whether a light was turned on by a human/entity in the space, an AI Agent, or something else. Finally, when a sensor agent starts up, it needs to inform the middleware of the state of its sensors by publishing an event for all of its sensors. As these do not reflect an action, but a state, it is important to capture this as well. As such, the spec includes a "category" field in event messages. This currently consists of *entity*, *state*, *control*, or *system*. Human (or other entity such as a pet)-caused messages are tagged as *entity*. Messages that are more of a "status update" are tagged as *state* (such as temperature or light level updates). If an agent ordered or created the event (such as a light turning on as a result of an agent turning it on), this would be categorized as *control*. (Note that even if an agent turned on a light, a light level reading would still be *state*: the agent did not necessarily cause the light level event to be generated right then). This distinction is of great importance to AI algorithms. This also allows some flexibility as more control is exacted on the space to further isolate and identify how messages were originated. The final state is *system*, which is for agents to log information about the system itself (such as an agent starting up).

Finally, the spec includes the message itself. This is a free-form text field; there are no explicit requirements for its formatting at the middleware level. This is perhaps one of the most important features, as this allows the addition of new sensor types with data types we haven't even thought of yet. It won't break any properly-written subscriber (it will just ignore what it doesn't understand).

2.4 Message Transport

With XML chosen as the message format, that leaves the transport. These messages that are being exchanged are similar to the human concept of sending an "instant message" on the Internet

(such as through Google Talk). As such, we decided to look at instant message clients. Due to the open-source nature of our work to this point, we preferred to use an open source instant messaging platform, and one that is friendly to XML. Based on this, XMPP (Extensible Messaging and Presence Protocol) [19] (also known as "jabber", the protocol used by gtalk, jabber.org, and growing numbers of instant messaging providers) is an obvious choice, as it is one of the most familiar, most developed and widely used open source instant message infrastructure, and it is based around a loose variant of XML. In addition, many XMPP client libraries already exist for most programming languages, and a selection of XMPP servers are available free of charge with good performance and efficiency features.

Due to the wide adoption of XMPP, there are many professional packages already out there, such as ejabberd [20], Openfire XMPP server [21], and a whole host of others [22]. These server packages have extremely large deployments, such as running jabber.org's XMPP infrastructure, and as such, receive a lot of attention and support. With the growth of XMPP in the instant messaging arena (many of the non-XMPP based instant messaging providers are either slowly disappearing or switching to XMPP), we can be assured it will be around for quite some time, and a considerable number of people have worked on its engineering. Combining this with the open development and open source, this implies that this software is high quality (and this can be verified by anyone wishing to inspect the source). Its specifications are well written and freely available at [19], the XMPP website. And, XMPP is supported under virtually every modern programming and scripting language. All of this means we, the CLM designers, need not work on an actual network-layer message passing infrastructure, nor do we need to investigate and understand the protocol at a deep level. Instead, we can leverage the work of experts in that area and instead focus our attention on the remainder of CLM that is unique to our problem and within our expertise.

2.5 Security

In this day and age, all software projects, and especially those that involve communication, need to at least assess their security, and all would benefit from building in security at the beginning. The CLM design choices allow for a wide variety of security. For CASAS, one of our requirements is reliability. One consequence of poor security is a reduction in reliability. While not presently a need, there is also a strong potential for future work to require privacy and access controls enforced in the middleware.

Specifically, by choosing to use XMPP, one has a selection of XMPP servers, all with some form of security implemented already. Furthermore, most production-grade XMPP servers have undergone a security study, and the authors are security-conscious (and if the one in use has not been considered, or a major security problem is found and not promptly fixed, it is easy to switch to a different server that is). Furthermore, XMPP employs (at least optionally) various forms of SSL and password protection.

CLM builds its core around existing software to the greatest extent possible without compromising on the design goals. In this way, we leverage programmers expert in their area rather than trying to be jacks of all trades ourselves. This strategy also reduces the amount of source code belonging exclusively to CLM, and thus the bugs and security flaws present in our work. Those who write the portions we use (most notably the XMPP server) take care of their own software, reducing the burden on CLM programmers.

In CASAS's deployments to date, we have used minimal security measures within the protocol, usually limited to simple passwords. As CASAS places the data collection services behind a tight firewall and uses only encrypted services for moving data between sites, this level of security meets the current CASAS requirements for reliability. Privacy isn't a huge concern, as most data sets are posted on the CASAS website for free download, as all of our participants sign agreements allowing us to do so. However, CLM has designed in the capability to secure the infrastructure if/when

additional security is required. Thus, as the requirements evolve, additional security features may be enabled.

2.6 Commercial Implications

Smart home middleware design decisions also have some interesting implications for future homeowners if CLM is used in commercial deployments. As it uses all open standards that are not that much different from what many people are already familiar with, it allows hobbyist smart home owners (or commercial end users with appropriate programming staff) to easily integrate their own programming into their smart home. It also allows them to easily monitor their smart home while simultaneously not requiring that they do so. In addition, by using open source products at the core, such as the XMPP server, users and homeowners can inspect the code themselves for any back doors or other vulnerabilities. The CASAS project promotes leveraging open source software and standards. Our hope is that commercial deployments will do the same.

3 CLM VERSION 1 AND XEP-0060

The XMPP specification [19] includes a publish-subscribe specification as an optional enhancement, specifically XEP-0060 [23]. This enhancement adds to an XMPP server the ability to directly manage publish-subscribe message decimation. In essence, this would be the core of the middleware just designed. Furthermore, it already exists and is written by domain experts who focused simply on the efficient, reliable, and effective delivery of those messages.

CASAS' requirements for throughput and speed are fairly light; currently 10 messages per second is more than sufficient; a design goal of supporting 20-30 is generous. CLM will be running primarily over a LAN with occasional remote subscribers over high-speed Internet connections. As such, bandwidth is not of much concern (which supports our decision to use XML as our core data format).

Publishers and subscribers are referred to as "agents" in CLM. They publish or consume events on channels. It is helpful to have the ability publish and subscribe events to or from different channels. For example, we put all of our raw sensor events into the channel "rawevents", and allow other agents to create channels for publishing any derived data or decisions about the space. This allows agents to subscribe only to the channels it needs to get the types of data it wants.

Occasionally, it is necessary for an entity to send a non-broadcast message to another specific entity. This is most often needed for control data (turning a light on or off, for example), but may also be used in the future for state query or other purposes. Our Chumby agent (discussed later in this section) also makes use of this method. By supporting point-to-point connections, it is also easy to implement security (and prevent unauthorized entities from controlling things, for example).

In this implementation, the ejabberd XMPP server [20] was used, but any jabber (XMPP) server supporting XEP-0060 (Publish-Subscribe extensions) will work, such as the OpenFire server mentioned previously. As our existing code base was all written in Perl, our "clients" are also

written in Perl. Initially, this posed somewhat of a problem, as while the excellent Net::XMPP2 [24] XMPP Client module existed, it did not yet have XEP-0060 (PubSub support) implemented. We wrote a beta version of this implementation and contributed it back to the author in the spirit of Open Source; soon the standard distribution of Net::XMPP2 should include support for XEP-0060 by default.

In the XEP-0060 spec, a channel must exist before an agent can subscribe to it. Therefore, CLMv1 has a tool that creates and maintains the initial channels. Our publishers were then free to begin publishing, and subscribers to subscribe without race conditions, and have a more robust implementation capable of tolerating agent failures. In the typical CASAS deployment, all the agents, the XMPP server, and the channel creation agent ran on the same computer due to their small load requirements. XMPP allows for future expansion in capacity simply by using multiple computers; the XMPP protocol is a network protocol after all. This flexibility and expandability is an important design criteria, and allowed for many specialized agents (see chapter 5).

3.1 Experiments on CLM v1

Given the nature of our project goals, it is very difficult to quantitatively evaluate our success in the stated categories (ease of use, ease of programming). On the other hand, there are aspects of our implemented middleware that can be evaluated and compared with alternative approaches. To this end, we evaluate our middleware approach based on three performance dimensions: throughput, bandwidth, and scalability. While not related to the project directly, we do include measurements from a basic implementation in CORBA for comparison purposes.

3.1.1 Throughput

The first and perhaps most important question was throughput. As previously stated, our middleware needs to process 10 messages per second minimum, with 20-30 preferred. These tests were carried out on a single machine to remove network effects, although network effects should be trivial, as CLM typically runs on 100base-T Ethernet.

Table 3.1 Throughput Results (1000 messages, times in seconds)

Service	Total Time	Messages per Second
CORBA	3.87	258.26
XMPP XEP-0060	21.87	45.73

Table 3.2 Bandwidth Results (1000 messages)

Service	Total Transfer (KB)	Bytes per Message
CORBA	337	337
XMPP XEP-0060	1004	1004

The results, seen in Table 3.1, show both middleware implementations easily met the requirements. As the CORBA middleware utilizes a binary message format with binary-optimized message passing algorithms, it is faster. Still, the slowness of the XMPP was somewhat surprising. Further investigation revealed that the XMPP server settings were not optimized, but optimized settings were not determined in time to rerun the experiments. Also, as we continued to work on and improve CLMv1, we discovered many efficiency problems in the XEP-0060 implementation in Perl that were outside our abilities to fix. Due to this, further experiments were not performed. System statistics during the run showed that the computer was not heavily loaded during XMPP experiments, which further establishes that we have not hit a true ceiling, and the XMPP-based middleware could easily still handle other traffic.

3.1.2 Bandwidth

One of the key purposes of implementing the middleware was to allow agents to operate on separate computers and pass their messages over a network. Consequently, the question of bandwidth use is relevant. The XMPP middleware should in theory require more bandwidth than a dense binary format due to its use of XML. For this test, we ran the publisher and middleware core on one computer, and a subscriber on a second. Wireshark [25] was used to collect network statistics, and was started after the publishers and subscribers were exchanging data to ensure we did not capture any start-up traffic.

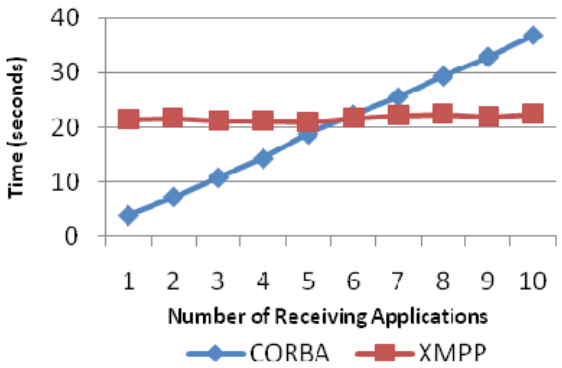

Figure 3.1: Scalability: CORBA vs XMPP

The results are in Table 3.2. CORBA was significantly smaller due to its binary protocol, but also posed many debugging problems over the course of the work. The XMPP middleware, while using more bandwidth, was easily debugable using a simple network packet capture and still used only 1004 bytes per message, an amount easily supportable with its intended deployment (even over low bandwidth WAN interconnects such as 256kbps DSL). Conversely, the CORBA implementation rendered wireshark debugging nearly useless, and often required writing a separate program, or using runtime debuggers to diagnose problems. CLM had none of these difficulties; it was easy to see what was being passed over the network and diagnose problems; we did not need to write any special debugging tools.

3.1.3 Scalability

The final metric measured was Scalability. In this test, we had one publisher and differing numbers of subscribers, and measured the time taken to deliver 1000 messages using code instrumentation as in the throughput test. The number of subscribers was incremented and the test rerun.

As the results in Figure 3.1 show, XMPP had stable (constant) delivery times while CORBA grew linearly. Again, we tested only one XMPP server and one CORBA implementation; results likely would vary with others. From this, one can deduce that the XEP-0060 implementation in

ejabberd was well-written and highly scalable (as is supported by some of its large-scale deployments on major jabber networks), whereas the CORBA orb we used on Perl was a relatively small open source project of which the message passing implementation was only a small part and not optimized for multiple recipients.

3.2 Observations on CLM v1

As implied by the section title, this was a first implementation. Due to some major problems encountered, a second implementation was required to achieve all of our operational goals. XMPP with XEP-0060 proved useful and capable, and supported rapid implementation and deployment of several additional agents without distributed system programming knowledge or understanding (See chapter 5 for details). Debugging also proved easier than CORBA due to the plain text data that was sent over the network. In contrast, debugging CORBA data as it passed over the network proved to be a constant challenge. Over the course of about 3 months, we ran this middleware at two smart homes, where a total of about a million events were handled.

Unfortunately, several limitations came up making further work difficult. XEP-0060, while well-supported in ejabberd, lacked much client support, and the support that does exist is not well tested or supported. As mentioned above, we had to write and contribute Perl's Net::XMPP2 support of XEP-0060; few languages' XMPP libraries included XEP-0060. As CLM v1 ran for prolonged periods of time, memory leaks appeared and other coding bugs impeded reliability that were tracked down to somewhere inside Net::XMPP2, usually some interaction between our contributed code and the original author's code. As few users are using XEP-0060, the code specific to it is not as well tested. Furthermore, improving XEP-0060's performance is not a high priority for the community at large, and we were not equipped or prepared to do it ourselves. Doing so would have put CASAS into the position we were trying to avoid: writing and maintaining a custom middleware system that would not be used outside our lab.

In addition, we were unable to find any pre-existing end-user chat applications supporting XEP-0060. This is significant, as one of the goals of using XMPP was to take advantage of pre-existing user software to allow debugging and interaction with the environment. Due to the light adoption of XEP-0060, there is little demand for such applications. This also hindered development with other programming languages, as few XMPP client programming stacks offer XEP-0060 support natively. While our effort was successful in its ability to pass messages, it did not fulfill all of our expectations. As a result, we created CLM2.0 to address these shortcomings and difficulties encountered by CLM1.0.

4 CLM VERSION 2

While XEP-0060 was invented to solve problems such as ours, its implementation is not yet mature or widespread enough to make the software as usable and programming language independent as we would like. For this reason, we adopted another approach: use XMPP in its original, core use case and have a piece of software of our own creation handle the publish and subscription side of things. This version consists primarily of a *manager* daemon that listens on a known address for XML commands and data. It manages a list of subscribers for different *channels*. When a publisher publishes something to a channel, it simply sends the data to the manager. The manager then re-sends it out to all subscribers on that channel. All messages are passed as "standard" XMPP messages (the same as a human user sending a message to a friend via XMPP). This is likely to be less efficient than the first implementation, but we hypothesized that it would be easier to use and more universally compatible.

Programming for this implementation turned out to be easier than expected, and far easier than implementing XEP-0060-based middleware. The first alpha version was written over a single weekend, and immediately was able to take over middleware duties at our development smart home. This initial version, written completely in Perl and making use of many resource-intensive functions throughout, still managed to meet all but our most demanding needs. Over the course of time, improvements were made to improve efficiency, and ultimately, the manager was ported to Python. The end result allowed us to run our entire middleware suite (including ejabberd) on a sheeva plug computing device [26] – an embedded computer the size of a "wall-wart" style power converter that consumes a miserly 4W of electricity.

In addition, as eliminating dependence on XEP-0060 made us fully compliant with the standard XMPP messages, we were now able to use existing clients throughout. This meant that not only were there ready-to-use library implementations in virtually every modern programming and

scripting language, but we could now also use regular end-user clients such as pidgin [27] to manually interact with and debug the system. With automatic subscription management, we could now subscribe to presence messages ("add a buddy" in instant message terminology; normally it allows one to see the online/offline/busy status of individuals on their "buddy list") and tell at a glance the status of all the agents. Figure 4.1 shows a sample of what this looks like.

Modern instant messaging systems allow users to set a "presence message" which is essentially a status message often displayed with the buddy name in the list. In this figure, you can see that the agent "dbloader" has logged 1,515,757 messages since it was started, and this status was last updated Thu, Dec 11 10;15:43 2008. We are still in the process of adding additional output to other agents, such as a load indicator for the manager representing how many messages per second it averaged over the last 5 minutes, and the last event received for the Insteon and OneWire agents. This allows us to see at a glance that everything is working well.

The Manager accepts some XML-like commands that allow a debugging operator to receive raw events in real time, check the list of subscribers, or identify the last known state of all sensors it has heard since it started. This is the normal suite of commands used by all the agents, but as it is all done in plain-text XML, and all these commands are normally carried through standard XMPP instant messages, the manager does not know nor care whether it is a human typing the commands in a messaging client or an agent. This makes debugging and testing significantly easier; on many occasions, the ability to receive all messages in real-time on a wireless laptop has proven extremely valuable.

4.1 Performance

Since the first implementation performance measures did not measure our goals for CLM, we did not repeat them here. Furthermore, due to the extensive dependence of external projects for critical portions of the middleware (as was our design goal), overall performance depends heavily on portions not written by or under our control, further reducing the usefulness of these metrics.

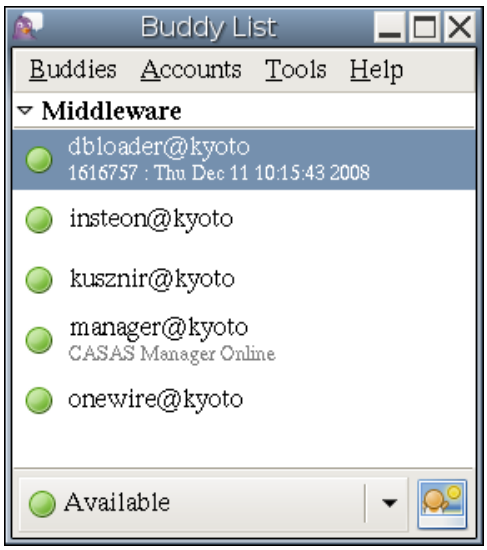

Figure 4.1: Buddy List with early revision manager, two publishers, and one subscriber running.

We have deployed a number of sites and processed several million events under CLM v2. It has been been operating continuously for over two years at our primary two sites, as well as at several other sites for shorter term experiments.

CLM has been running on a variety of lightweight and embedded computers, with the Sheeva Plug being our device of choice these days. Even on these meager computing resources, CLM running under load still is easily handled by this hardware. Its dependencies are also lightweight, making it possible to run the entire suite for a standard smart home on a single, small device while still keeping up with real-time message delivery.

The ease of expandability and programmer-friendliness has come through in the many additional application components that have been created in the short time we have been using CLM. Some examples include two different power meter interfaces (OneMeter and TED), two different visualizers, and even an online AI agent that not only subscribed to events, but performed its computations on them and published its own conclusions on a separate channel. We have also leveraged the flexibility to replace core agents such as our database logger. This transition was clean and seamless , requiring no downtime or modifications to any other agents. For more details

on the agents used with CLM to date, see chapter 5.

5 DESCRIPTION OF AGENTS USING CLM

One of the core goals of CLM is to allow easy extension through the creation of CLM agents. In software design terms, agents are the modules that use CLM as their message passing infrastructure. Some agents are hardware interfaces, allowing information to enter CLM or commands to be executed that affect the world CLM runs in. Others are various means of processing, displaying, or recording the information passing through CLM. In any case, CLM was designed to to make creating agents a rapid and easy process, and the following list of agents created by CASAS programmers speaks to that goal.

5.1 OneWire

This is the first of the CLM agents, and still is one of two core agents that are needed in any deployment. It uses the OneWire Filesystem (OWFS) project [28] for the low level hardware drivers, and manages the OneWire bus. This agent is a publish-only agent; presently we do not have any controller hardware that sits on the OneWire bus. We have some potential plans to add controllers to the OneWire bus, at which point this agent would need to be modified to supply some kind of control interface.

This agent has been through many major revisions. Originally written in Perl, this agent was able to run our two production smart homes, but began to encounter execution bottleneck issues with lots of activity. It was revised for efficiency, and finally ported to Python, where it runs today on our embedded hardware.

5.2 DBLoader

Our other core agent is a means of recording the data received. This agent is a subscriber only, and logs data to a local PostgreSQL database. In our original deployment design, each smart home had a local PostgreSQL database that was copied to our lab every 24 hours. Having the local database allowed local tools to query it for more information, but as this feature was rarely used,

we replaced it with the Scribe (Section 5.4) for a more efficient, more frequently syncing system. This agent was written in Perl and used in all CASAS smart home deployments until late 2009. It is now retired.

5.3 Insteon

Our two primary deployments make use of Insteon [2] [1] light controllers that provide our first and primary control system. This agent is a publisher with an XMPP control interface. It listens using an Insteon PLC, which is a USB to Insteon Power Line Carrier bus interface. This device captures messages sent by the light switches over the power lines, and passes them to a monitoring tool. The Insteon agent then converts and publishes these into the CLM. It also listens for direct XMPP messages for control commands to turn on or off lights.

The Insteon agent was written in Perl and has been used in three CASAS facilities. We have retired Insteon devices from one of our production facilities and no longer install it in new facilities due to limitations with the Insteon technology itself that resulted in unreliable operation. It is still operating in one facility, and has not yet been replaced (a project slated for summer 2010).

5.4 Scribe

The replacement for DBLoader, the Scribe is a Python-implemented agent that logs all messages to a text file in single-line format with one event per line. By default, every 15 minutes it rotates the text file and moves it to a directory for transfer. Additional scripts then transfer the file to the data warehouse at the CASAS central facility and load the events directly into the production PostgreSQL database. These support scripts have been engineered to ensure good, clean transfers and to ensure that the data load successfully into the database. The design is robust in the event of a network connectivity failure between the remote deployment and our central facilities. This means that no data are lost, only delayed until connectivity is restored. As a result of this system, data from our facilities typically end up in the production database no more than 15 minutes after

being generated.

5.5 ASCII Visualizer

Our first visualizer, this agent used “ASCII Art” to draw a layout of a CASAS smart home to the screen. This means text characters are used to represent the space and sensor activity. It would either connect via XMPP to the middleware to show events in real time or take file input for playback. For a couple years, this was the only visualization platform and is still one of the most portable and efficient. It was quickly written in Perl as a debugging tool.

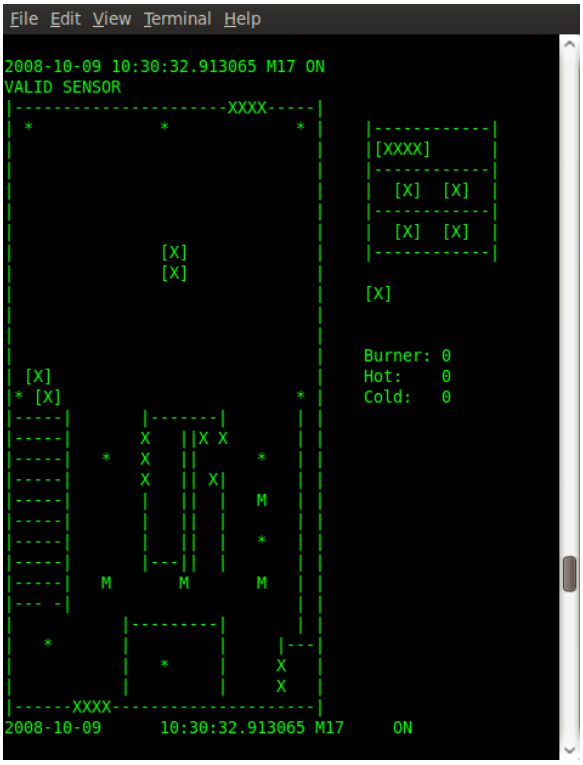

Figure 5.1: AsciiViz screenshot.

5.6 SecondLife Visualizer

Visualizing our data in an appealing way has always been challenging. In an attempt to satisfy many requests for a 3d visualizer, several students began constructing a visualizer of our two

primary facilities using SecondLife [29] (See Figure 5.2). Using SecondLife allowed us to use the capabilities and libraries for 3d display, and allowed us to just focus on the sensor and environment model. Even so, this project was a major undertaking; much effort went to building a 3d model in an OpenSimulator [30] server. OpenSimulator is an open-source implementation of a SecondLife server. This was chosen because it was cost-prohibitive to do so in the Linden Lab's servers. Simply building the 3d model in the simulation took two people nearly a month per smart home.

A variety of scripts were used to control the sensor state within the virtual space based on real-time data from CLM. While running, this proved to be a very high visual quality visualizer. However, real-world studies showed it was far less useful [31] than the ASCII visualizer, or even looking at raw data. Persons processing the data found the 3d interface to be too visually overloading, and the control interface to be too inefficient. All of the lab personnel at the time preferred the ASCII Visualizer over this one. In addition, considering the server resources needed to run it as well as the low reliability of the system overall, the SecondLife Visualizer was eventually shut down three months after bringing it online. In short, it looked good but was far too people-time-intensive to keep operating. As a result, work began on a less ambitious visualization project: PyViz.

5.7 PyViz

PyViz is the current CASAS production visualizer. Written in Python, it uses GTK and other GUI elements to render sensor events on a graphical floor plan of a smart home installation. The floor plan is provided in Scalable Vector Graphics (SVG) file. The sensor IDs are encoded in metadata within the SVG file. PyViz is able to connect to the master database, CLM, or read from a file for its data. It has multimedia-like play-rewind-fastfoward buttons, a user-adjustable speed indicator, and attempts to play events back respective to the relative time stamps within the data. Instead of playing X events per second, it takes into account the time at which the events occurred and plays them back at the same rate.

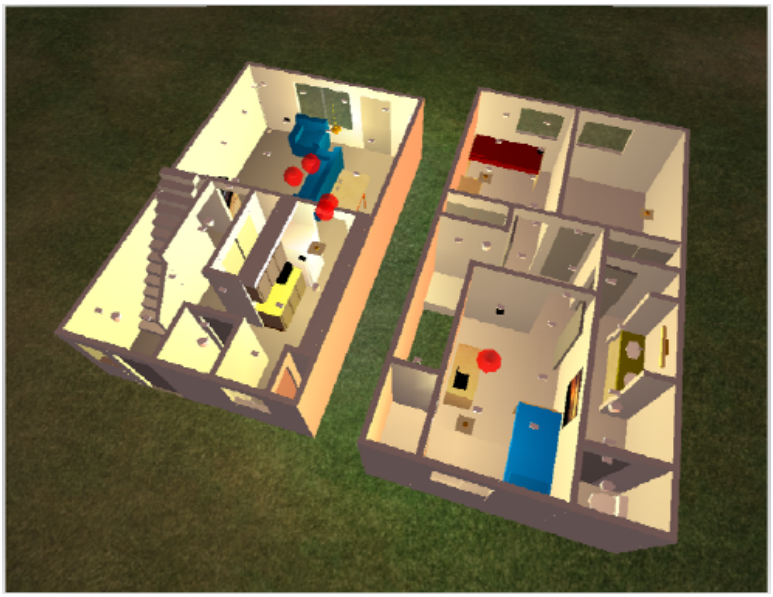

Figure 5.2: Screenshot of the SecondLife-based visualizer. White dots are motion sensors, red spheres indicate active (on) motion detectors.

PyViz is currently the most visible presence of CASAS deployments. It is used daily by the people who process and annotate the data collected from our deployments as well as for presentations and real-time status of our on-campus lab. For privacy reasons, we do not show live data from other facilities except when on-site at that facility.

5.8 Chumby

CLM also enabled us to add another device known as a "Chumby" (see Figure 5.4), a small embedded computer with a touchscreen. This device provides another way for the smart home to interact with the user through verbal alerts and touchscreen controls. The Chumby uses a micro-Linux distribution on an ARM processor, for which a Perl distribution is maintained. The CASAS group build a CLM agent for the Chumby, created a CLM channel, and now the Chumbys are an extension of the CASAS smart home.

The Chumbys are used to provide localized auditory output capability to the smart home. As we tested with two different Chumbys, we needed a means of controlling and managing them.

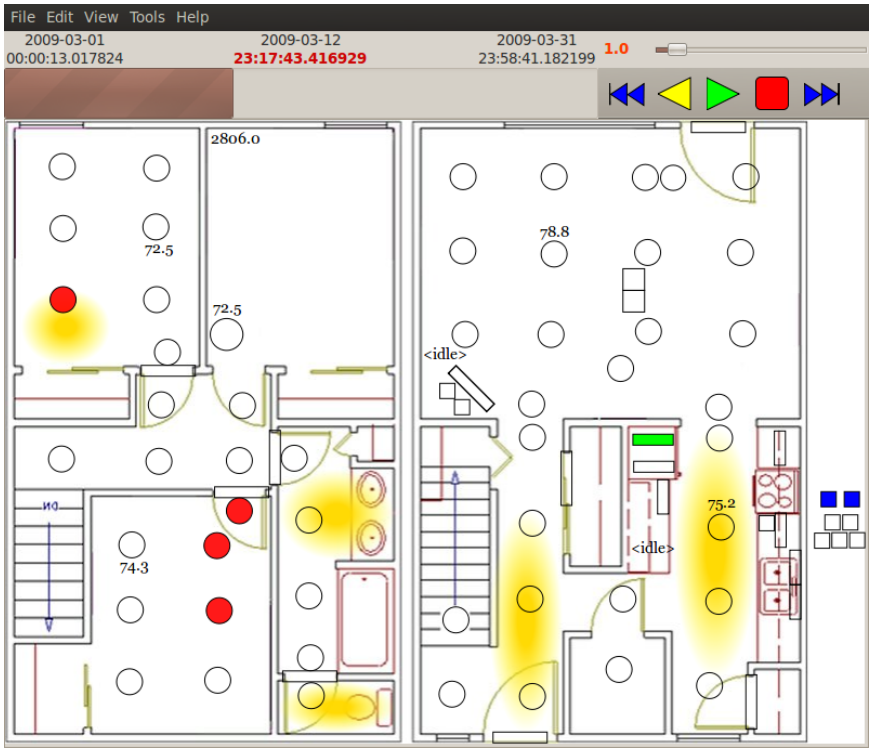

Figure 5.3: PyViz screenshot.

Thus, we had a Chumby-manager agent that listed to the raw events stream for whatever trigger events we programmed in, and then sends messages via the Chumby channel to the Chumbys. The messages were sometimes to a specific Chumby, other times they were broadcast and intended to be received by all Chumbys. The instructions usually consisted of an audio file to play. Due to the light processing capability on the Chumby, the Chumby-manager also ran a text to speech software. Upon detecting an appropriate event on rawevents, the chumby manager would utilize the text to speech software to generate an audio file with the spoken message on it, then send a message via CLM to the Chumby to play that audio file.

5.9 Occupancy and Time-based Lighting Control Agent

This agent provided basic lighting automation to our lab facility as a precursor to implementing an artificial intelligence method. By doing so, it helped us understand the limitations and social

Figure 5.4: 2007 Chumby as used in CASAS Testbed.

issues with lighting control. The lessons learned with this tool were taken under consideration for future control designs.

The agent was a subscriber and controller; it listened to the raw event stream from the manager. We configured it with lighting zones, which lights were controlled for that zone, and the motion detectors that belonged to that zone. When it detected motion in a zone it would turn on the light(s) affiliated with that zone by sending a control message to the Insteon agent directly, and start an internal timer. Each time more motion was detected, the timer was reset. When the timer elapsed, an off command was sent. To account for users' preferences a user could turn a light back off, and that would establish a timer that prevented further control of that light for its duration.

This system showed that even with good motion detector coverage, motion-based timed automated lighting is not very useful. While the occupants quickly adjusted to not needing to turn on lights in a space, it also showed how often people generated no events while working. This led to the lights turning off as people sat still and worked quietly. Even with idle timers of an hour, lights would still often be turned off while people still occupied that zone. This frustration helped establish the criteria for version two.

5.10 Stateful Tracker Lighting Controller

One of the CASAS areas of research is entity detection. This encompasses the counting and following entities in a space, be they people, pets, or others. By tracking entities, the system can "remember" that someone is at a certain location even if no motion was detected. For example, if an entity is tracked walking across the room and stopping at a location, and none of the adjacent

motion detectors are tripped, then its reasonable to assume the entity is still there. While this concept sounds trivial, there are many details that make it not so, but these are beyond the scope of this work.

For this agent, we married the lighting controller from V1, but added entity awareness. Unfortunately ”pure” entity detection was not sufficient, as entity detection and tracking was not perfect. As such some timers are still needed, but now each zone had two states: occupied and unoccupied. When one or more entities were in a zone, then ”long timers” were utilized, often three or more hours. This normally prevented lights from turning off while a space was occupied, but also recovered from faulty entities (“ghost” entities). When a space is unoccupied, then a relatively short timer (about 5 minutes) would control the lighting turn-off. This would allow for people just walking out of the lighting zone momentarily, then coming back. Also, as Insteon supported dimming, when a timer expires and a light is to be turned off, it would dim the light for one minute before turning it all the way off. This way, if the system made a mistake, it was less disruptive to users of the space.

5.11 Entity Detector

As part of the research to build the Stateful Tracker, we wrote a standalone entity detector agent. This agent publishes information about where it believes entities to be on its own channel. PyViz was modified to also subscribe to this channel and show this data when available.

5.12 Experimenter Reminder Control (ERC)

For a CASAS research study involving cognitively-impaired adults, a system for allowing psychology graduate students to issue prompts to the study participants was required. For the purposes of this study, we designed a system that used laptops in the study space to play video and/or audio clips upon demand. An agent was created to accept commands via the middleware and play the requested prompt via a media player. To control this, a basic user interface was created that acted

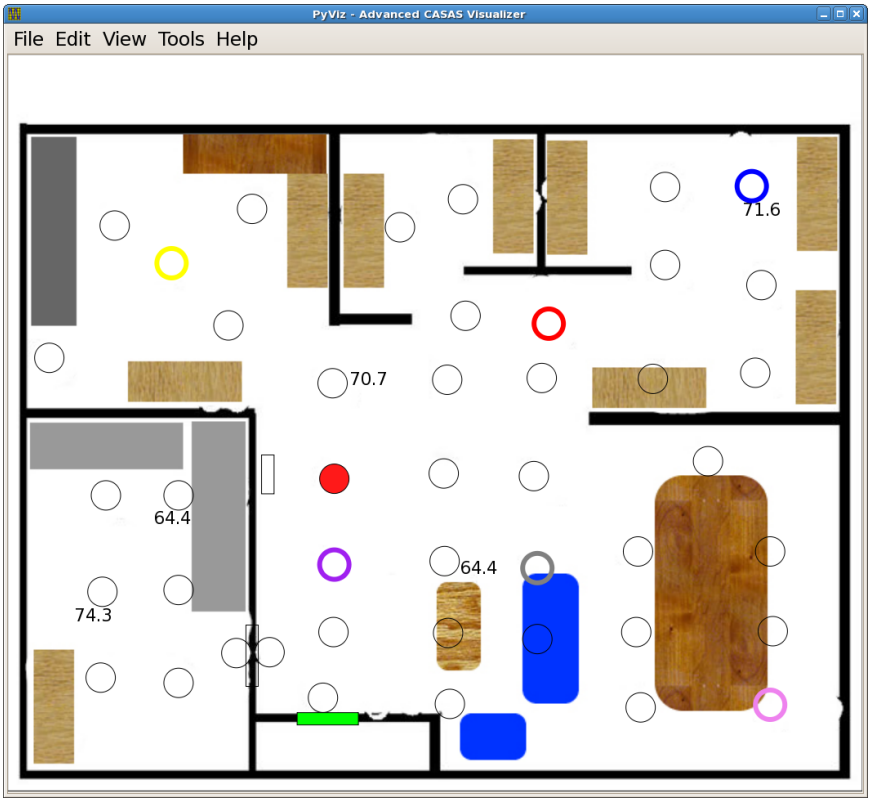

Figure 5.5: PyViz showing entities, colored rings show location of entities.

as a publishing agent to send the commands through the middleware.

We opted to send these messages through the main middleware channel so as to ensure they also got logged along with the rest of the collected data. This comprehensive data can then be used to train a machine learning algorithm as an automated means for issuing these prompts.

5.13 Prompting System v2

CASAS also conducts some in-home studies with volunteers with cognitive impairments. At these deployments we also deploy a basic prompting system. This prompting system is much simpler than the above to provide some assistance to the participant(s) by an automated means. To date these have been Asus EEE-Top touchscreen systems that are used for issuing audio and pictorial prompts. They then accept input from the participant through the touchscreen as a response.

For software, we wrote middleware agent that watches the events and waits for an appropriate set of events to trigger a prompt. It then publishes the fact it saw the trigger and issued a prompt, and then issues an event when the user responds, or when the prompt ”times out”. This agent is still in the early stages of development, and will probably be split into two (or more) agents for better adherence to good programming practices.

5.14 OneMeter

CASAS has begun to take steps to integrate energy efficiency into our automated systems. The first step to energy efficiency is monitoring energy usage. To that end, CASAS acquired a OneMeter [32], which is a circuit breaker panel-level power consumption meter with a RS-232 interface.

This agent is written in Perl and polls the power meter once every 6 seconds, and if the current instantaneous wattage changed by more than a configured value, an event is published with the current power consumption in watts. The deployed system uses a threshold of 11 watts, as this amount is enough to catch real events such as efficient lamps turning on or off, while reducing noise due to power fluctuations from the utility. PyViz was then modified to watch for and graph

Figure 5.6: OneMeter kit.

this value in real-time, which resulted in some very interesting and useful graphs.

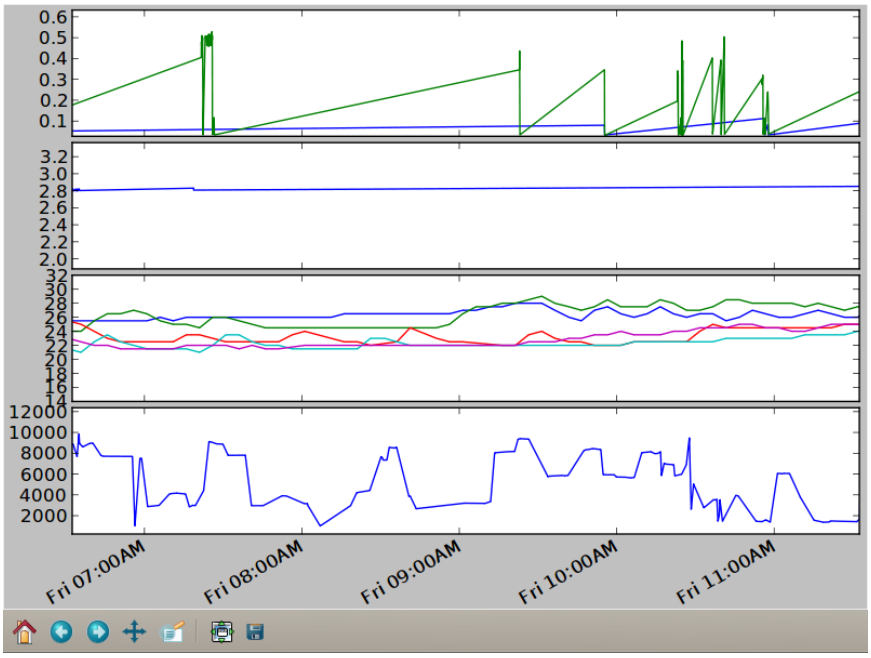

Figure 5.7: PyViz graphing screenshot showing (top) Water usage, Stove Usage, Temperature, and Power (bottom) data.

5.15 TED

After our first facility was equipped with the OneMeter, we found out about The Energy Detective (T.E.D) [33] power meters. The hardware is more in line with what we needed at a much lower cost, so we switched to it. This meter simply takes a reading and puts it out on a serial port once

Figure 5.8: T.E.D. meter display and interface unit.

a second. It is also much easier to wire in, and does not need to be polled as the OneMeter does. Much of the OneMeter Agent Perl code was reused for this project. Once the data is published through CLM, there is no perceivable difference in the resulting data between these two different power meters, or any other power meters we may use in the future.

5.16 SysWuzzup

SysWuzzup is a Perl agent that registers with the publicly available jabber.org servers to present system status information from any site to CASAS personnel from any jabber client/system. This does not connect to CLM, but is an "out of band" management option connecting to public jabber.org servers. It reports the system load average in the status message, and updates every 5 minutes. This allows us to tell at a glance what the site's status is, and detect potential problems. As we've made some changes to CLM, the usefulness of this agent is decreasing, and will probably be retired in the near future.

6 CONCLUSIONS

We set out to create a lightweight, flexible, fast, and easily extensible middleware for smart environments. CLM demonstrates that using XMPP as the centerpiece of the middleware, all of that can be accomplished. Not only is CLM theoretically useful, it is in very active use at the Center for Advanced Studies in Adaptive Systems (CASAS), where it has virtually limitless uptime (current uptime record is over 200 days – and that's with 24/7 data collection in a lived-in space). Our researchers have also found it easy to develop to its interface, and so far it has already adapted to several new data types since its inception with no modifications necessary. All things considered, CLM is a real-world success.

6.1 Future Work

Of course, there are always ways to improve something, and CLM is no exception. To date, we have done little work on executing control of the space. For control, the publish-subscribe model does not fit well. Control messages are a one to one message, not a one to many that publish-subscribe is well-suited to handle. This results in a different subsystem to handle control efficiently. As mentioned above, we have done some planning, but current needs in CASAS have not led to the installation of control devices in our space. Recent research needs are changing this somewhat, so we are beginning to enter into this arena, and we have two first passes on a control interface. We expect more work in this arena in the future.

In any network-facing system, security is important. As mentioned above, CLM does offer security through XMPP's implementation of it. To date, CASAS research has focused on other aspects, and the security capabilities of CLM have not be studied or tested. We know its possible, but that is where we left it. There is room for evaluating the security of the infrastructure and the development of a "best practices" guide to address security in the smart home using CLM.

Another interesting question exists in "linking" smart spaces. We currently have three facilities

operational. Currently each space is treated as a separate entity; however, there have been some questions into the ability to link them and make them interactive. For example, can someone's events at work (the SmartLab) impact the Smart Apartment where they live? Should they? How would that work? What would be required in CLM to support this, and how would that impact reliably and existing models?

Finally, there have been some recent explorations in the concurrent programming language and runtime *erlang* [34]. Erlang was designed to handle distributed and concurrent systems efficiently and effectively; threads are built-in. In addition, many other aspects of the language suggest that it would be a relatively simple undertaking to port many of our agents over to erlang, and significantly reduce their resource footprint while simultaneously improving capacity and performance by orders of magnitude. Future directions for this research include conversion of the middleware for this language and analysis of the effect on performance.

BIBLIOGRAPHY

[1] (2010) Smarthome.com online vendor. [Online]. Available: htp://www.smarthome.com

[2] (2010) Insteon technology homepage. [Online]. Available: htp://www.insteon.net

[3] S. Krakowiak. (2003) What is middleware. [Online]. Available: http://middleware.objectweb.org/

[4] J. I. V. D Lopez-de Ipina, A Barbier, "Dynamic discovery and semantic reasoning for next generation intelligent enviornments," in *Proc. IE'08*, 2008.

[5] (2010) Osgi alliance. [Online]. Available: http://www.osgi.org/Main/HomePage

[6] G. M. Youngblood and D. Cook, "Data mining for hierarchical model creation," *IEEE Trans. Syst., Man, Cybern. C*, vol. 37(4), pp. 571–572, 2007.

[7] (2010) Object management group, coordinating body of corba homepage. [Online]. Available: http://www.omg.org/

[8] (2010) Soap specification. [Online]. Available: http://www.w3.org/TR/soap/

[9] H. Ristau, "Publish/process/subscribe: Message based communication for smart environments," in *Proc. IE'08*, 2008.

[10] (2009) Jini project homepage. [Online]. Available: http://www.jini.org/wiki/Main_Page

[11] (2009) upnp forum homepage. [Online]. Available: http://www.upnp.org/

[12] P. Rashidi and D. Cook, "An adaptive sensor mining model for pervasive computing applications," in *Proc. of the KDD Workshop on Knowledge Discovery from Sensor Data*, 2008.

[13] D. Cook and e. S. Das, *Smart Environments: Technologies, Protocols and Applications*. Wiley, 2004.

[14] D. Cook and M. Schmitter-Edgecombe, "Activity profiling using pervasive sensing in smart homes," *IEEE Trans. Inf. Technol. Biomed.*, to appear.

[15] (2010) Dallas/maxim one wire bus. [Online]. Available: http://www.maxim-ic.com/products/1-wire/

[16] (2010) Occam's razor (dictionary). [Online]. Available: http://dictionary.reference.com/browse/occam%27s+razor

[17] R. S. Pressman, *Software Engineering: A Practitioner's Approach 6th Ed*. McGraw-Hill, 2005.

[18] S. L. Pfleeger and J. M. Atlee, *Software Engineering: Theory and Practice 4th Ed*. Prentice Hall, 2010.

[19] (2008) XMPP homepage. [Online]. Available: http://www.xmpp.org/

[20] (2008) ejabberd homepage. [Online]. Available: http://www.ejabberd.im/

[21] (2010) Openfire homepage. [Online]. Available: http://www.igniterealtime.org/projects/openfire/index.jsp

[22] (2010) Xmpp.org list of xmpp servers. [Online]. Available: http://xmpp.org/software/servers.shtml

[23] (2008) XEP-0060 specfication. [Online]. Available: http://www.xmpp.org/extensions/xep-0060.html

[24] (2008) Net::xmpp2 homepage. [Online]. Available: http://search.cpan.org/dist/Net-XMPP2/lib/Net/XMPP2.pm

[25] (2010) Wireshark homepage. [Online]. Available: http://www.wireshark.org/

[26] (2010) Sheeva plug dev kit homepage. [Online]. Available: http://www.globalscaletechnologies.com/p-22-sheevaplug-dev-kit-us.aspx

[27] (2010) Pidgin, an instant messaging client homepage. [Online]. Available: http://www.pidgin.im/

[28] (2010) One wire file system project homepage. [Online]. Available: http://owfs.org/

[29] (2010) Secondlife, the virtual world. [Online]. Available: http://secondlife.com/?v=1.2

[30] (2010) Opensimulator homepage. [Online]. Available: http://opensimulator.org/wiki/Main_Page

[31] B. M. S. Szewcyzk, K. Dwan, "Annotating smart environment sensor data for activity learning," *Technology and Health Care, special issue on Smart Environments: Technology to support health care*, vol. 17, pp. 161–169, 2009.

[32] (2010) Onemeter whole house power meter. [Online]. Available: http://www.brandelectronics.com/onemeter.html

[33] (2010) T.e.d. power meter. [Online]. Available: http://www.theenergydetective.com/ted-1000-overview.html

[34] (2010) Erlang programming language offical homepage. [Online]. Available: http://www.erlang.org/