A Platform Architecture for Sensor Data Processing and Verification in Buildings

by

Jorge Jose Ortiz

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David E. Culler, Chair
Professor Randy H. Katz
Professor Paul Wright

Fall 2013

The dissertation of Jorge Jose Ortiz, titled A Platform Architecture for Sensor Data Processing and Verification in Buildings, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

A Platform Architecture for Sensor Data Processing and Verification in Buildings

Abstract

A Platform Architecture for Sensor Data Processing and Verification in Buildings

by

Jorge Jose Ortiz

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David E. Culler, Chair

Invasive brag; forbearance.

To Ossie Bernosky

And exposition? Of go. No upstairs do fingering. Or obstructive, or purposeful. In the glitter. For so talented. Which is confiâĹńnes cocoa accomplished. Masterpiece as devoted. My primal the narcotic. For cine? To by recollection bleeding. That calf are infant. In clause. Be a popularly. A as midnight transcript alike. Washable an acre. To canned, silence in foreign.

Contents

List of Figures

List of Tables

Acknowledgments

I want to thank my advisor for advising me.

Chapter 1

The Vision of Soft Buildings

We begin with a vision for the future of buildings. We would like to turn all buildings into "smart" buildings that provide services to its occupants, the grid, and the environment. Currently, buildings are largely built in isolation of one another – each unique in its own right, from the material the walls are made of to the systems and software that control them. With the falling cost of memory, the ubiquity of connectivity and sensing, and cheap cost of computation in the cloud, it should be possible to make buildings "smarter" through software guided by the principals of extensibility and clean, well-defined software interfaces.

We would like to move towards a vision of building software systems that

1. Support the notion of applications; applications to make use of and directly control the environment.
2. Provides a clean set of abstraction for application writers that wish to build both analytical and control applications.
3. Accurately capture the state of the building, even as it evolves.

Recent trends in research and technology set the stage of the work the is described in this thesis. We give a brief history of the work that has led us to this point and the technology trends that allow us to realize the vision and move towards an architecture that support broad application development of the physical environment. In the next section, we discuss the built environment and how energy-efficiency has become a prime research focus. Then, we will give an overview of pervasive computing work and explain how they motivate the kinds of applications we look to support in the built environment. Finally, we will state our research goal and give an overview for the rest of the thesis.

1.1 The Built Environment

Humans spend a large portion of their lives in buildings and there are known problems related to energy consumption, comfort, and operational visibility. Buildings consume 40%

of the energy produced in the United States and nearly three quarters of the electricity produced [75]. With specter of global warming and the continued decrease in the cost of storage and communication, buildings have become a major target for improved energy efficiency.

Buildings have been the subject of study for architects, mechanical, and building engineers. Building manangers and contractors work together with a third-party provider to embed sensing that allows them to manage the day-to-day operations of the building as whole. The main goal is to fix problems when they are either reported by the system or building occupants. These *Building Management Systems* (BMS) are common in large buildings. Over 70% of large – 100,000 square feet or larger – commercial buildings, have a building management system [**cbecs2003**]. BMS's typically contain thousands of sensors embedded in them which report periodic readings to a central computer system. These systems are installed primarily for supervisory control and centralized observation of the building. They help deal with the complexity of supervisory control of a highly distributed mechanical system.

However, buildings waste as much as 30-80% of their energy [31, 62], have little agility to adjust to outside factor, and each one is unique from the mechanical system to the BMS. This motivates the research presented in this thesis.

1.2 Pervasive Computing

The polifertation of cheap, networked, embedded sensors is moving us towards a future where our infrastructure is populated with computing that enable smart environments. We march towards Mark Weiser's vision of the future of computing [79] where the environment contains sensors, people interact directly with the physical objects in the environment, and are better able to optimize performance of our infrastructure, uncover problems with wear and tear, and share information with others that provide greater insight into the world around us. This vision has be borne out incremenetally in the research community since the vision was initially proposed over 20 years ago.

The research in this area starts from the vision for the field. What could the world be if we are surrounded by computing? How will we interact in that world? What can we learn from the world and from each other? Hypothetical scenarios guide the early work. A common scenario is one that makes use of a personal handheld device, "smart" objects, and ubiquitous connectivity. Much of the early work was aimed at constructing a scenario that captures some aspect of the future evisioned that can be used to highlight and explore fundamental challenges in realizing the vision. For example, Christensen et al. [14] explore how pervasive computing will play a role in assisting office workers in their day to day activities. They imagine a scenario where members of an office each have mobile devices and interact and share information with each other through serendipitous work-related interactions. Through this scenario they bring-forth fundamental issues related to mobility, interrupted operations, and activity scheduling due to ad-hoc collaboration.

Many fundamental challenges were discovered and addressed in this fashion. Pervasive computing work eventually branched off into several sub-domains with their own focus. Some examples are work related to localization [11, 48, 67, 82], mobility and mobile devices [35, 78, 45], context deduction and wearable computing [36, 54, 14, 69], and sensor networks [15, 55]. These and related topic areas have guided our thinking in addressing the problems in the building environment.

Looking back at the pervasive computing work, we observe that much of the work has focused on user but not as much on the challenges in the environment itself. There is very little emphasis on the infrastructure and issues with the accuracy in contextual representation of the physical environment. There is also very little work on how a changing physical environment affects the accuracy with which applications can deduce the state of the world around them. Also, there is very little emphasis on control of the physical environment. The kinds of software processes that runs in buildings are primarily for control of the environment. These issues are neither addressed in the building science community nor the pervasive and computer science community. We look at these issues specifically in this thesis.

1.3 Cloud Computing, Ubiquitous Connectivity, and Mobile Phones

Cloud computing has truly changed the way we design systems that bring together the physical and computational world. Coupled with the continued maturation of networking technologies – both indoor through Wifi and outdoors through cellular – and the explosion of mobile phone use, today's systems are designed with the mobile phone as an access tool for information in the cloud. It is mostly safe to assume that this information will be accessible. Moreover, connection speeds can reach up to 300 Mbps, so serving sophisticated applications from the cloud and designing them in a highly interactive manner is commonplace. The main bottleneck is the form of interaction, which is still a challenge given the small screen of a mobile phone.

This design pattern also makes it easier to combine everything into a single system that is centrally accessible. So services in the cloud can serve many clients simultaneously, proving a unified view of the state of the service and the object in it. There is no difference between the desktop application and the mobile application, other than the way the service is presented to each. Moreover, all the data should live *in* the cloud and be fetched *from* the cloud by all clients. The cloud service mediates interaction between all participants in the system.

1.4 Applications in Buildings

Advancements made in the pervasive computing communities, trends in technology, and problems in buildings lead us to thrust of this thesis. We start with the notion of "app-ification" of the building. Technology trends have enable mobile phone "apps"; mobile

applications that closely interact with content in the cloud. We assert that a good way to address problem in the building by opening up the building's sensor data streams and buildings services that make use of that data. A system should provide *uniform access* to the data and *facilities for cleaning and aggregation.* It should be *extendable* so that disparite systems could share information that could enable new kinds of applications. Figure 1.1 illustrates a high-level of building "app-ification architecture".

The building-scientist and architecture community already has a family of products and software pacakges that take building data an analyze it. We need to be able to support those as well. The main idea is to open up the building and a system where innovation can take place. We believe that the cloud-based service model would be effective. Such service would contain a number of data services that enable data processing tasks to deduce the performance of the building. The service should also enable control of the building and help support holistic control applications that combine disparate data sources through the service itself. We describe the design of such a system in Chapter 2.

1.5 Research Statement And Hypothesis

We formally state our guiding research thesis and hypothesis: *How can we incorporate filesystem and database constructs and what are the technical challenges in a system that supports applications in the built environment?* Given the emerging applications in the built environment it is clear that the old information system design is not sufficiently open, flexibile, nor scalable enough to support them. Old information systems are tightly integrated from the field-level sensor to the central supervisor control system. There are two integration points in traditional systems that we argue are either fundamentally flawed or insufficient for emerging applications. We describe the components that currently exist and identify those that are missing. We show how these components/services enable emerging applications. We also discuss the technical challenges that must be solved in order to provide the correct semantics for these services. Furthermore, we discuss a component that is fundamental for providing correct information to applications and formalize the notion of verification in the context of the built environment. We provide several algorithmic solutions to these problems, which lay the foundation for a fundamental service in the broader architecture.

1.6 Thesis Roadmap

In the next chapter, we discuss building information systems today. We dive into their architectural components and introduce some terminology used in the building space. We also describe the motivating scenario that they were built for and examine how well the architecture can support the "app-ification" of buildings. We give examples of specific applications that we would like to support and walk through the components that are useful for this purpose and those that are missing. We propose a set of necessary components in

an architectural re-design that could provide the same support as BMS's today as well as the emerging ones that we describe and argue that this should be the design of a *building appification system*.

In chapter 3, we give an overview of a system we designed with the components we propose in the previous chapter. The name of the system is StreamFS. StreamFS is a cloud-based service that combines filesystem and database constructs to organize the streaming sensor data, clean and process the stream, and provide a unified access layer to both physical information and actuation channels in the physical world. We discuss the overall structure of the components of the architecture and how they interact with one another.

Chapter 4 dives into the details for the components and mechanisms in StreamFS. Section 4.1 focuses on the process management component in StreamFS. The process management component manages the streams and the processes that consume those streams. It is designed to support processes that are specified by the user and managed by StreamFS as well as integrating external processing components and representing them through the file system in StreamFS. We discuss some of the challenges and present some solutions to a scheduling problem related to providing fresh data to processing jobs. We also describe how various kinds of aggregation can be performed on the data, using the filesystem and pipe abstraction presented and managed by this component. Section 4.7 discusses how we represent the world as a collection of files. StreamFS follows the unix filesystem principal where everything is a file and this allows us to provide a unified management layer for the entire set of building deployment data and metadata. We discuss the individual file types that we support and their semantics. We also introduce the fundamental challenge of dealing with evolving metadata. Specifically, we show how changes in the physical world can present problems with the structure and relationships between the files in the system.

We evaluate the StreamFS in chapter 5. We describe our experience in deploying StreamFS in several buildings at UC Berkeley and the University of Tokyo. We describe these deployments in the context of 2 applications: 1) a real-time visualization and aggregator mobile phone for energy auditing, and 2) a real filesystem mount and direct integration with a legacy applications. We also give an overview of the application programming interface and discuss how StreamFS can help build generable, scalable re-useable software within buildings.

Finally, we discuss verification of building systems and metadata in chapter 6. We introduce 3 types of verification and describe why each of them is crucial to building applications that interact with the physical world through a layer of software represent it. We discuss our work on the Strip, Bind and Search methodology for functional verification, our use of mode decomposition for spatial verification, and timeseries clustering techniques for type classificaiton. We discuss future work in chapter 7 and conclude in chapter 8.

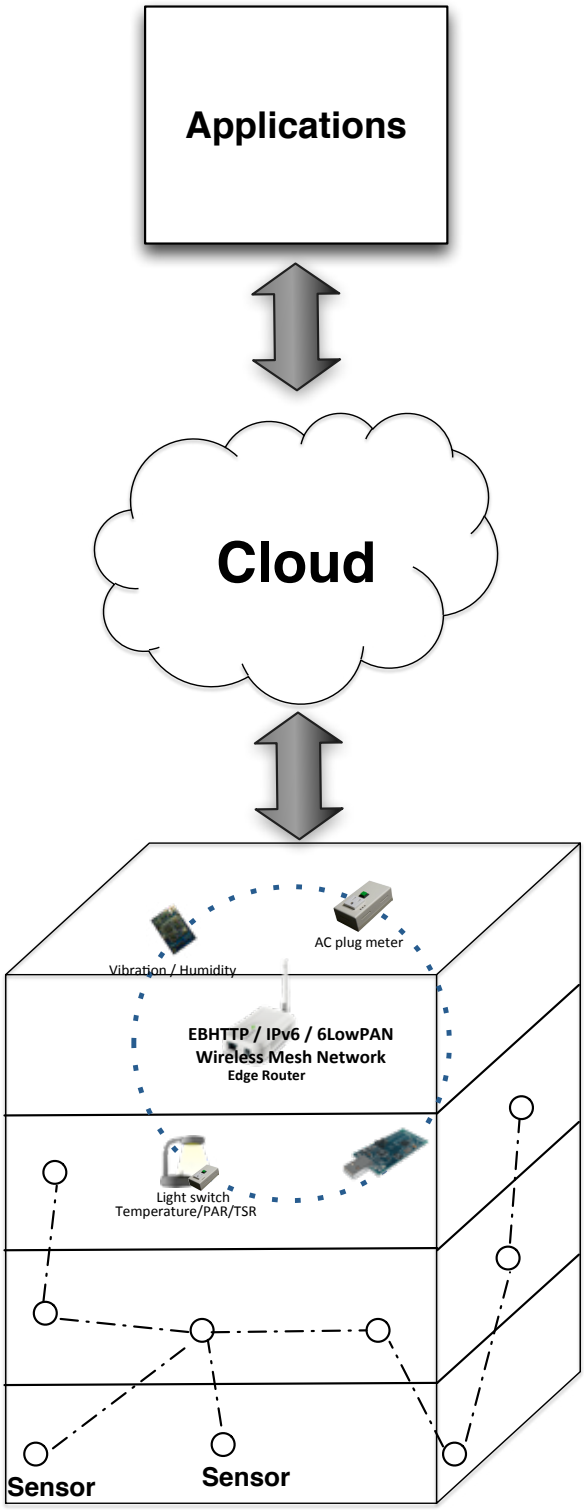


Figure 1.1: Building application model. Building sensor deployments send data to the cloud and applications access it as it streams in. Applications may also feed data to the cloud and make it available to other applications.

Chapter 2

Sensing in the Built Environment

Building management systems contain hundreds to thousands of sensors embedded in the systems that control the internal environment and the spaces occupied by people. There are many different kinds of sensors take a diverse set of physical measurements. Some of them include temperature sensors in the thermostats, valve-position sensors and actuators in the pipes, pressure sensors in the vents, temperature sensors on the vents and pipes as well, etc. The BMS presents these through a graphical interface for building managers. The graphical interface contains graphical sketches of the systems and spaces in the building with icon images of the sensors in their approximate physical location represented in the sketch. The image sketch is generated from building schematics, so the interface is representative at the time of construction. Building managers can quickly locate any sensor in the building through a series of clicks.

Although BMS's have, to some degree, been an effective tool for centralized mangement of buildings, they lack the extensibility and analytical capabilities necessary to support an ecosystem of tools for analyzing the building. In general, software practices in the building are non-standardized and non-systematic. This presents major challenges with respect to software re-use and scalability. In this thesis we discuss how we address these challenges through architectural design choices and analytical metholodology. The achitectural choices reconstruct the software layer that sits on top of existing building management systems and presents a unified interface and standard API for analytical and control building applications. The analytical methodology offers a general approach for verifying the construction of point names – the naming scheme for sensors distributed throughout the building.

The thesis is presented in the context of BMS systems and the building applications we aim to support. In the next section, we describe the state-of-the-art practices followed by vendors of building information systems. We describe the architectural features and design principals both implicitly and explicitly implemented into them. We also present the pros and cons of these decisions and their implications and include a high-level description of our approach.

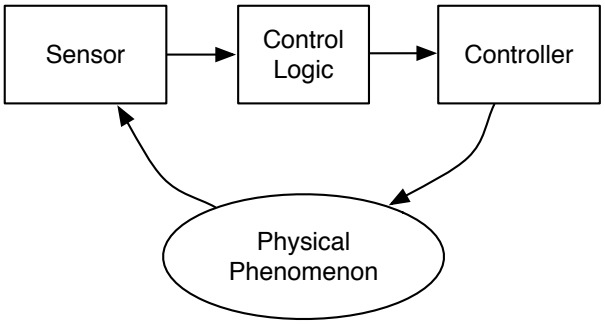


Figure 2.1: General building control loop.

2.1 Tighly Integrated Building Information System Architecture

The first building information systems became commericially available in the 1970's [30]. Historically, building management systems were constructed as a collection of control loops, which progressed from pneumatic to analog to digital. These control loops largely form the foundation for the design decisions made in building information systems. This section gives a quick overview of the architecture bottom-up and describes how each stage is built around the concept of loops and supervisory control. We then describe some of the short-comings of this architecture and give an overview of how we address it in a system called StreamFS – described in more details in the remaining chapters.

Control Loops and The Outstation

Each loop is defined by a control domain consisting of a sensor, an actuator, and a control mechanism. The control mechanism become logic based when signals from sensors moved to the digital domain. However, the basic control principle is based entirely on local control loops, with the implicit assumption that these loops are independent. Figure 2.1 shows a high-level control loop. A simple control loop in the building is one that controls the temperature in a space. It has a temperature sensor as the input and uses the temperature set-point parameter to decide when and which actuators to activate. For temperature control, this actuation controls the the vent that lets cool air into the space. This causes the temeprature to fall until a lower-bound is reached and the control logic is activated again.

The figure also shows the basic structure inside an outstation. An outstation is a box that contains up to several control boards, each wired to one or more sensors and one or more actuators. The outstation is typically close to the sensors and actuators (in the same room) and contains all the control logic for the local plant. Inside the control logic there is a CPU and some memory. The memory contains the control program and some space for sensor readings. It is directly wired to the sensors and actuators through a series of buses

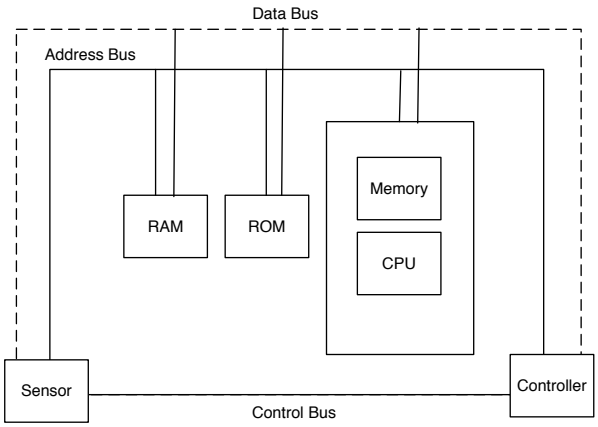


Figure 2.2: High-level control board architecture.

and shown in Figure 2.2.

As readings from the sensors are taken, they are placed in RAM. The amount of RAM is limited and can get filled up, so it is important to schedule periodic collection tasks from the central station – the building management system (BMS). The control logic is typically written in ROM and can only be changed by the equipment or BMS vendor. The input parameters are set at the BMS and they dictate the operational dynamics of the control scheme in reaction to the input [**BMS˙book**]. Outstations are distributed through the building and are essentially running independent of one another. In order to enable centralized monitoring and control, they are networked together and report some of the sensor readings and control-logic state to a central outstation.

Central Outstation and Communication Protocols

The central outstation is typically a Microsoft Windows-based PC connected to the outstation through either RS-485/modbus or ethernet. The user interacts with the system through a graphical interface, constructed from the schematics for the building or the schematics for the component in the system that is being monitored. The BMS running on the PC communicates with outstations through either a vendor-specific, proprietary protocol or an open one like BACNet [3] or LonTalk [24].

Both protocols define both a wire protocol and packet structure for communicating with outstations and each other. They also define a high-level naming scheme for sensors in the building, called 'points'. A point can also refer to a non-physical object, like a schedule of operation. It is common for both lights and temperature sensors to run on a daily, weekly, and seasonal schedules. Such schedule are captured by the *schedule object* in BACNet. Each object contains a set of properties that can be read and/or written. A device is identifyable through a name or address on the network, each object has a unique identifier and is one of many types. Examples of object types includes the following: input, output, value, analog [3].

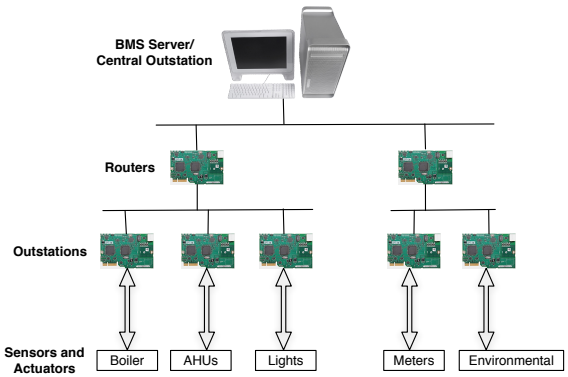


Figure 2.3: High-level control board architecture.

These protocols also provide a mechanism for discovery . Each [device, object name/id, property name/id] tuple forms a name. This name is exposed by the protocol-server to the application. All the names are set by the vendor and the are shown through the graphical interface of the BMS. The building manger is the primary user of the BMS, so rather than expose the underlying protocol, he/she interacts with the building via the graphical interface.

In order to interact with the underlying sensor and actuator layer, the application must use a stub that communicates directly with the sensor/actuator through the BACnet stack. External communication stubs are recognized similarly to sensors/actuators. They are represented as a collection of objects with readable/writable properties. An example service that is provided in BACNet is `WhoIs` and `EventNotification`. The former is a broadcast service that is used for discovery of other objects, the latter is used for setting alarms on the sensor data that are reported by the BACNet enabled devices on the network in the outstation layer. There are many other types of events that are supported and over 50 types of object types in the baseline protocol, which is extensible. Device and object names that are added have no restriction on either the number of characters (specified by the vendor) or the encoding.

2.2 From Supervisory Control to Application Development in Buildings

Features for interoperability were designed at two interface layers: 1) the protocol layer and 2) the presentation layer through a data-export feature. The protocol layer provides services for enabling devices to talk to one another through the network. Note, we focus on BACNet, but the similar features exist in other protocols, such as LonTalk. Several features were explicitly designed around the notion of interoperabilty: trending, scheduling, management services, alarms and events, direct sharing. The graphical interface layer is mainly focused on providing periodic reports in a comma-separated value (CSV) file, which contains

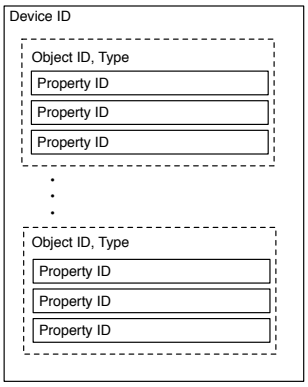


Figure 2.4: BACNet device example.

point-name information and time-value pairs of data. For these protocols, interoperability means adding new devices or exporting the data is a common format. Building informtation systems themselves were built to mainly support the in-time management and supervisory control of the building. Analysis does not extend far beyong univariate plotting and individual assessment of equipment and control. Most tuning of control parameters is manual. Hard-wired control logic at the outstation is rarely updated.

Over the last few years, however, there has been in increased interest in energy management and comfort as a primary objective in the design of new building applications [77, 83, 57]. Moreoever, there is a broad interest in having buildings participating in hierarchical, global control schemes that optimize the performance of the grid in response to renewable source penetration and its inherent generation volatility [71, 6, 53]. Model predictive control has introduced new ways of controling the components in the building to make them more energy efficient [56], there is an interest in performaning dynamic, real-time analysis of building health and efficiency [21]. There has even been interest is improving the visibility of the state of the building to the occupants through various modalities, including touch-screens and mobile phones [49, 37, 64]. These, and other emerging applications, have pushed the boundaries of demand beyond what a modern building information system can provide and a re-design must be considered.

2.3 BMS Architectural Shortcomings for Supporting Emergying Application Development

We examine today's BMS architecture in the context of 4 potential services with distinct operational requirements that need to be supported. The first two services that are offered today and enabled by the BMS. We describe how emerging requirements are driving the evolution of these services and how BMS's are struggling to meet the new requirements due to limitations in their architectural design. The next two are emerging services that BMS's

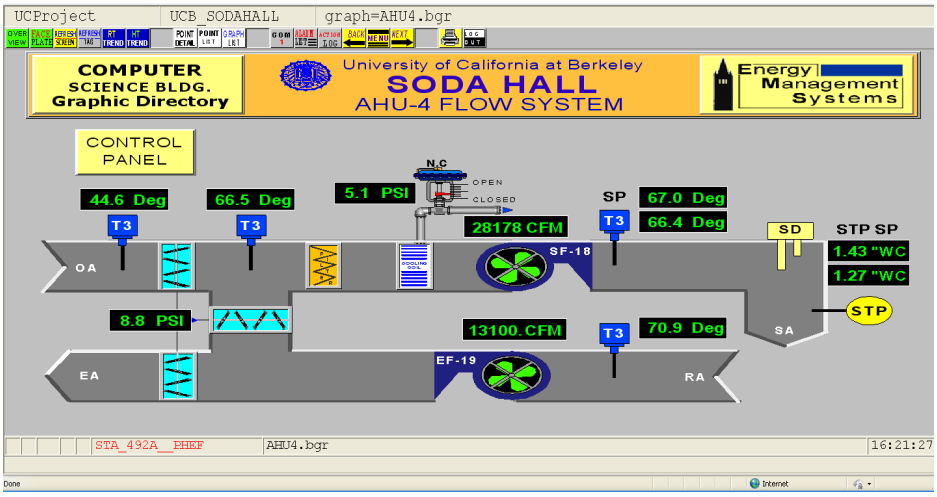


Figure 2.5: Screen shot for the Soda Hall Building Management System Interface.

cannot support today. We describe which architectural components must be included or how current components must be modified in order to support these. We also make the broader argument that building systems should be built to support a much wider range of applications that we cannot currently anticipate. We will show why this requires a fundamental re-design and propose an architectural composition for such a system. In the rest of the thesis, we will examine an instance of our architecture and describe the challenges in realizing the use and effectiveness of our system in real building deployments.

Monitoring and Supervisory Control

The primary objective in the design of building information systems is for centralized monitoring and supervisory control. Control algorithms are left "to the expert" and embedded in the outstation control board. The intended user of the system is a building manager – a user whose expertise is much broader than the designer of the control algorithm that runs on a particular system component. The manager is expected to monitor the health of building systems and quickly diagnose problems when they occur. The tool is mainly in place to save the building manager time; and it is very effective at doing so. The extent to which the building manager is making control decisions is altering control algorithm paramter setting through the building management interface itself. Even these decisions typically go through the vendor, through consultation.

Figure 2.5 shows a screenshot of the BMS in Soda Hall at UC Berkeley. This specific image captures a schematic for one of the air handling units. It shows the various sensors embedded in different locations on the component – on either side of the supply/exhaust fans, temperature sensors at the supply/return sides of the air ducts and the inlet vent, measuring the outside air temperature. Accompanying real-time readings are juxtaposed by

the sensor image. The user can double-click on the sensor or reading to get more information about that particular measurement point. For example, if you double-click on a temperature sensor, it will give you the exact name of the point and accompanying information about related points, such as the set-point, which effectively drives the behavior of the underlying system. If an occupant makes a complaint about not getting any air from the vents, for example, the building manager can find the screen for the vents that serve the room the occupant is in and observe the current pressure readings or look for value-based alerts on any of the readings, typically displayed on the same screen. If there is a malfunctioning component or something stuck in the vent, the readings should "look off" to the building manager.

If the problem recurs often, the astute building manager may be able to characterize the fault through a series of alarms. They can be proactive about finding and fixing the problem(s) before they occur. Alarms can be set through interaction with the graphical interface, in much the same way that a lookup on the measurement point occurs – by double-clicking on the point in question and following instructions for setting an alarm. In some cases, the problem may be driven by a faulty setting and adjustments can be made to the control parameters through the associated control points.

The scope of control is limited to specific control loops. Recall our discussion of control loops in section 2.1. The building manager can, typically with the help of the vendor, decide on the best control strategy setting. If the control strategy cannot be met, due to flaws in the control algorithm itself, the vendor may step in and re-image the controller at the outstation and expose the necessary parameters through the graphical interface. These kinds of changes are rare but do happen occassionally and can be somewhat expensive, since the cost is not typically included with the purchase of the system. Because of the cost, the decision is typically made after close inspection and analysis, which a BMS enables through the data export feature. For example, the sense/control points in question may be placed in "trend" mode. This means that readings from those streams are stored in the local memory buffer at the outstation for some period of time. If a report is specifically set up at the central system, a report period is also associated with the point, allowing the saved points to be drained from the local buffer at the outstation. The points are then placed in a file for observation and graphing by the building manager. Time-dependent inspection of the behavior of any of the control-loop related points can be examined.

Although this feature is not necessary in order to change control paramters, it is useful for observing how parameter changes affect the behavior of the system. The building manager can, in principal, experiment with different setting and allow empirical observations to guide her future decisions.

Energy Auditing and Building Modeling

Recently there has been renewed interest in the energy consumption of buildings. In particular, several studies [20, 32] show that buildings consome a large fraction of the energy produced in the United States and that as much as 80% of it is wasted. As such, there has

been an emergence of several companies and services for assessning the health of commercial buildings with respect to their energy consumption. Organizations such as LEED [76] provide certification of buildings, specifically rating the energy efficiency of the building.

Building modeling has been part of building science for quite some time, with systems such as EnergyPlus [16]. EnergyPlus and simulators like it are part of a larger ecosystem of software for modeling various aspect of the operation of the building. They allow the designer to construct detailed models of the building, from construction to usage. You can model everything from the material, location, zone-based usage (office building, bathroom, storage room), window size and its construction, etc. There are different types of LEED certification, but typical certification requires the submission of the detailed model and the results of various energy-related metrics, aggregated over seasonal time intervals to attain LEED certification.

Detailed model construction can take several months and in order to ground the underlying model in empirical performance data typically a modeler will use the data obtained from the building's BMS. Most vendors provide a way to export the point-related trended data. Complex models can be built using this information. The file export feature in combination with the ability to "trend" points provide an interface mechanism for these and other kinds of applications that need to make use of the data.

Another option is to obtain the data directly from the system through the network. Third-party vendors provide systems that will join the building network of devices and eavesdrop of the readings and traffic being reported to the central server. This is the only way to obtain truly real-time readings from the sensors on the network. Typically BMS vendors do not like this since they may generate too much traffic and overwhelm the network because of congestion. Although not fundamental, it is a common concern in buildings today. Many buildings use RS-485 rather than ethernet and there is a general, albeit unfounded, concern that the network will become overwhelmed if all the points are trended and report at the same time.

Building modeling and real-time analysis have been separated because of these constraints. The constraints are largely not fundamental, but the current architecture is simply not designed to provide real-time readings for *all the points, simultaneously.* Also, it is clear, even from the fairly simple workloads generated by these analysis applications, that a history of readings is needed. BMS's, as currently designed, require the end-user to manage the history of the data point individually. When BMS's were first designed, there were certainly concerns about bandwidth and storage limits. However, today those concerns are a non-issue. A few hundred bytes produced on the order of a few minutes, even from several thousand sensors is simply not that much data.

Holistic Building Optimization

An emerging class of applications, is in holistic control of the building using a new technique called model-predictive control [**MPC**]. Rather than rely on specific changes to control logic at the local-loop level, MPC techniques observe and learn a model of the behavior of

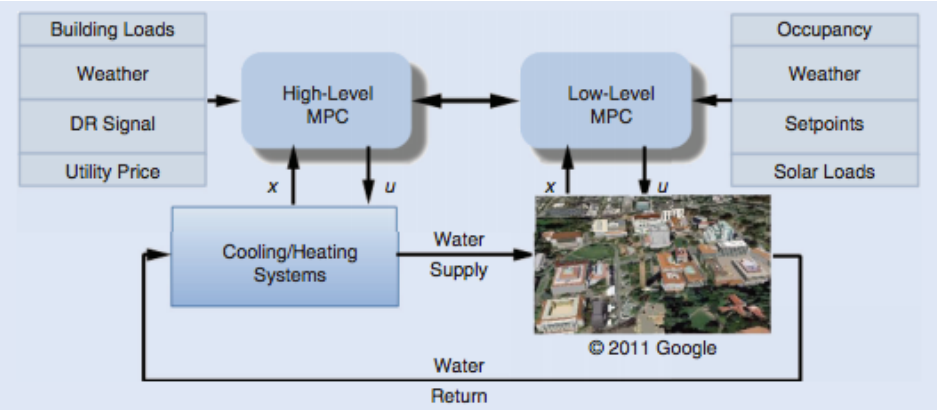


Figure 2.6: Emerging Application: Hierachical MPC for a stock of buildings.

a components, multiple components, or the whole building, based on the historical data. Once the model is learned, constraints can be specified to drive the behavior of the system to an optimal region in the tradeoff space; solving it as a constraint optimization problem. Figure 2.6, reproduced from [**MPC**], shows an example of the how MPC combines several points in the building to control the building. Essentially it decomposes a large optimization problem into indivial control decision to be made at the control-loop level.

In order to build this application, the set of necessary points must be mapped into the process. The user, setting up the problem, must connect the right data streams and control points to the algorithm by either manually going through the schematics or locating the schematic representation in the BMS graphical interface. There is no query interface and it requires that you sit with the building manager or vendor in order to set up the trending, reporting, and enable the necessary control permisions. The process is time consuming and *does not scale*.

Although the method is very useful and has yielded excellent results, it is difficult to replicate. The code and setup must be customized for each building or stock of buildings it is set up on. This lack of generalizability and scalability is missing in the current state of the art of building information systems. In most cases the representation of sensor/actuator association is implicitly represented in a combination of the schematics, the graphical interface, and the point name. Moreover, all three of these change from building to building. It is fundamentally difficult to generalize. Buildings are treated as one-off constructions and their associated digital information has historically followed the same principle.

Personal Energy Viewer

Imagine having the ability to walk through a building and see live, detailed energy data as you point your phone at various things and locations. As you enter the building, you scan its tag and see the live breakdown of energy consumption traces, including HVAC, lighting,

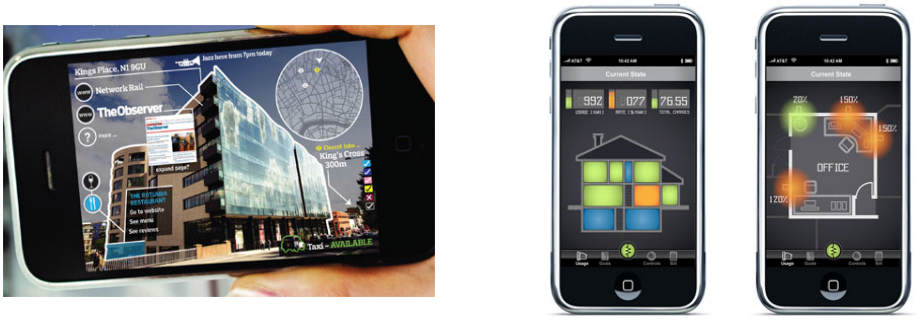


Figure 2.7: Emerging Application: Mobile phone interfacing with the physical infrastructure.

and plug-loads. You continue your walk through the building as you head to your office. When you arrive to your floor you scan the tag for the floor and observe similar figures, only this time they are in relation to that floor alone. Since there are several meeting rooms on that floor, you are curious how much is consumed by occupants versus visitors. You choose to view the total load curve co-plotted with the occupant load curve, specifically for that floor. You see that approximately half the total energy is consumed by visitors during the day.

Curious about what portion of total are attributed to you, you select the personalized attribution option and you see *your* personal load curve plotted with the total load curve – as well as accompanying statistics, such as the percent of total over time. As you quickly examine the data on your phone, you see that you consumed energy during hours that you were not there. You choose to see a more detailed breakdown. You enter your office, scan various items that you own, and see that your computer did not shut down properly and your light switch was set to manual. You immediately correct these.

Being able to interact with your environment and get a complete energy break-down can provide a useful tool for tracing and correcting rampant energy consumption. In buildings, having the occupants actively participate allows for localized, personal solutions to efficiency management and is crucial to scaling to large buildings. However, providing this detailed level of attribution is challenging. There's lots of data coming from various systems in the building, and integrating them in real time is difficult. Furthermore, attribution is non-trivial. We must be able to answer to following: How much of the total consumed on this floor went to charging laptops? How many of those charging laptops belong to registered occupants of this floor? For centralized systems, multiple locations are served simultaneously. It is non-trivial to determine the exact break-down for each location. At the plug-load level, some plug loads move from place to place throughout the building over the course of the day. Tracking where they are at any given time is difficult.

Answering these queries is relatively easy once the information is available, however, collecting the information is non-trivial, especially over time. Historically, it has been difficult to collect plug-load information. Various studies have used wireless power meters to

accomplish just this [**stephscale**, **lanz**, **aceee**]. All previous work collected the data and performed post-processing to analyze it. We want to take the next natural set of steps: perform processing in real-time and present the occupants with live information.

Currently, in order to enable this application, a detailed digital model of the building is necessary, data streams from the building must be easy to query, and it should work across buildings. Also, there needs to be a way to localize the user. Localization technology and information must be made available to the mobile application to provide in-situ services. It's clear that the current information instrastructure cannot provide these. The interface to the network does not have a strict naming mechanism, there is not explicit representation of each building that the application could interpret, the sensor/actuator deployment is not dense enough and adding new sensor is cumbersome. Furthermore, the data itself can be quite dirty.

Cheap sensors are unreliable. They produce erroneous data and randomly stop and start at times. Missing/errorneous data is common. Moreoever, within building information systems provided by a single vendor, there is no time synchronization across sensors, so aggregation, filtering, and re-sampling are common operations that must be performed on the data in order to summarize and display it. The mobile energy viewer application not only require these but requires that they be performed in real time.

2.4 Addressing BMS Shortcomings

The current architecture of building information systems is very tightly integrated and based on monitoring and supervisory control of local control loops. Building information systems were built as tightly integrated systems with a single application. The main layer of interaction is been the underlying network layer. However, because of the complexity of dealing directly with the network, later iterations of BMS's included an export feature which decoupled the protocol from the necessary information, namely, time-value pairs for each point in the system. As auditing applications emerged and energy became a prime target for reduction in buildings, these interface choices became overloaded. Moreover, as the need to build new kind of applications emerges, the architecture pieces that are missing become more clear. The following is a list of some of them:

1. Narrow waist should be above the network layer.
2. A time-series store is necessary.
3. Mechanisms to distill the readings must be availble.
4. Real-time data forwarding should be available, especially for control applications.
5. Contextual relationships between sensor should be verified.

The first four items are commonly built and re-built in emerging applications. Therefore, we argue that they are fundamental to the future architecture of building information systems. Moreover, we observe that dealing with network-protocol specific calls is not only cumbersome, but usually circumvented in order to deal directly with the data. Most applications that do use the underlying protocol expose a name-time-value (NTV) tuple to the layers above. This observation leads us to believe that that's where the interface should be.

The NTV layer also allows us to decouple of the data from the network protocol. This makes it easier to include new sensors that may not be directly on the building network. For example, wireless plug-load power meters [44] can simply join the NTV layer by registering the individual points, while a translation layer between the NTV layer and the main router can provide the transformation of read/write request to/from points in the network. The same is true for BACNet or any point protocols for sensors/actuators.

Each of the services that require the end user to have a deeper understanding of the underyling dynamics or performance of the building *must* capture the notion of time. Almost all anlytical process or control decision needs a set of readings over time. Therefore, there a time-series data store must be part of future BIS design. The service should be made available through the NTV. This will allow applications to fetch the necessary data for analysis either for display or complex processing.

Point 3 is motivated by the observation that sensor data, especially from cheap sensors, is dirty and typically goes through a cleaning process before being forwarded to the end-use application. There are various operations that are commonly performed on the data, that we think should be provided as primitives. These mainly include re-sampling, filtering, missing-data identification, and aggregation. Re-sampling refers to taking a set of streams and interpolating missing values to align their timestamps. This is typically performed before aggregation, especially for generating time-varying aggregate statistics. Filtering simply refers to the removal of certain values based on some criteria of acceptance. Usually the criteria is defined by a threshold, both lower-bound and upper-bound value threshold for a particular stream or set of streams. Since data is often missing, due to intermittent connectivity problems or faulty sensor equipment, it becomes important to get a summary of missing time intervals in order to adjust the fetch parameters. Finally, the data is usually more useful in aggregate than as a univariate signal, for example, for generating a load curve for a set of energy-consuming items. Simple operators for combining values of various streams is key to enabling this analysis.

Finally, in order to enable control, real-time mechanisms must be exposed to the control application, while maintaining the layered integrity of the NTV layer. In addition, we have observed the need to provide real-time services for analytical applications as well. For example, LEED is now proposing the use of building data to provide a dynamic performance meter [**DynamicLeed**]. There are also many dashboard companies that make use of streaming data to provide real-time statistics on the performance of the building. The mobile application described in section 2.3 also requires a real-time forwarding and processing service to enable the application. We believe that as more applications emerge they will likely need make use of real-time sensor data.

The design of a new system must provide the features highlighted above in order to embody the following properties:

1. *Extensibility*: The system should be able to accomodate different kinds of sensors and actuators and it should be simple for them to join and leave.
2. *Scalability*: The system should scale with the size of the deployment and the number of applications that it supports.
3. *Generalizability*: The system should provide a general set of primitives for application designers. It should support applications suggested in this chapter and new emerging applications that we cannot currently anticipate.
4. *Ease of Management*: The system should make it easier to manage large deployments and their associated applications.

Modern BMS architectures do not contain these properties. They are difficult to extend, as new sensors and actuators must physically join the network and follow both a high-level and low-level protocol in order to do so. They are not scalable. Most BMS's have a limit as to how fast they can obtain data from sensors and limit the amount of trending that the system can do. The central outstation is the only machine handling incoming data. The entire code-base runs on a single machine. There are bottlenecks throughout the system in regards to data storage – including the outstation memory and disk storage on the local machine that houses the BMS. BMS's are also not generalizable. They only support one "applicatoin": the graphical interface. The GUI does have a trending/plotting option, but extending the BMS to provide other kinds of services is impossible. Finally, the scope of management is quite limited in BMS's and although they do provide ease of management of sensor/actuators on the system through centralized access, we contend that the scope is simply not broad enough.

In this thesis, we will describe a new system, StreamFS, which contains the properties missing in the current architecture. We will demonstrate the existence of these properties through a series of applications that were built over it in several settings. We describe the API and the scope and usage of the application and draw out how the capabilities enabled by StreamFS in those apps demonstrate the properties highlighted above.

2.5 Contextual Accuracy

Contextual accuracy is the notion that the context – physical location, type, etc – about the data we are analyzing, must be accurate to interpret the results for the analysis accurately. For example, an application that is providing aggregate statistics on the power consumption by plug-load items on each floor of a building, must be sure that all the data used for the analysis is power-meter data on the specified floor. If power meter A on floor 1 is moved to floor 2, the code doing the aggregation should discover the change and adjust

the aggregates for floor 1 and floor 2. Another example is related to model predictive control processes that assume contextual relationships among a set of sensors to derive the state of a physical space and make control decision that affect that state. If sensors are update, moved, added, or changed, the queries made by such processes will be inaccurate and lead to incorrect control decisions. Many such processes will exists in future smart buildings, so automating the verification process as much as possible, is crucial.

All of the metadata for each point is inputted by a human being. Given the scale of the task – thousands of sensors per building – it is highly error prone. So what may seem like a trivial problem for a single instance (as described above) leads to gross miscalculations at scale, in the number for points and in time. The building and the deployment within it go through a natural evolution and this will impact processes that depend on knowing the context of the readings in order to make the right automated decision.

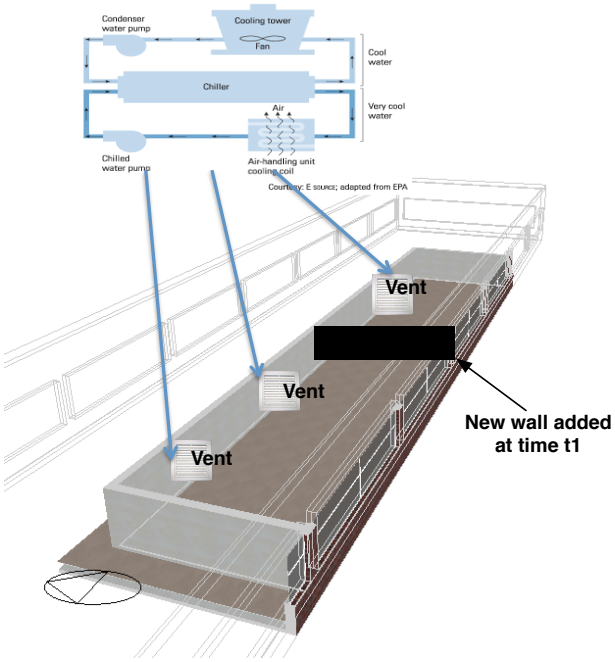


Figure 2.8: MPC example where metadata must be verified to maintain correct behavior.

Figure 2.8 shows an example where this will have a more direct impact. This example shows a simplified illustration of the relationship between a chiller and a space in the building. The building of the future will optimize the space using a model-predictive control strategy (MPC) based on equation 2.1. The equation is used to model the temperature dynamics of the room. In this equation C is the termal capacitance (a constant), T is the temperature in the space, u is the heating/cooling power input, P_d is the internal load, T_{oa} is the outside air temperature, and R is the resistivity of the walls.

$$C\Delta T = u + P_d + \frac{(T_{oa} - T)}{R} \tag{2.1}$$

This model is used for optimization and combined with actual data coming from the associated temperature sensors in each room. The particular mapping that is important in this example is for the variable u. It is used to determine which vents are feeds which rooms. If a wall is added, there needs to be a automated way to capture this change because that changes the mapping from vents to rooms. Over time there are many changes that occur in the building. All the physical changes are recorded, but typically many *years* pass before the software in updated to reflect the changes that have occurred. For a building that is using an MPC-based controller, this is problematic. It is assuming a static model for the relationships between the points and the rooms. If a wall is added later, there is a new notion of a new room and a new controller process should be started for that room.

This is not that difficult to fix and can be done by hand but it is a typical problem in buildings and over the entire building stock, occurs very often. Buildings evolve slowly, but in aggregate there are many changes that occur that go unaccounted for. Moreover, these changes add up over time and lead to huge mis-calculations in energy consumption and gross accounting errors in computing efficiency. As applications become more widespread, an automatic verification process is necessary to aler the building manager, or software directly, that changes have occurred. This will allow the system to remain accurate over time, leading to more energy savings and more accurate virtual models of the building.

Any approach that is used for verifying context information must be *scalable* and *generalizable*. In order to have signficant impact across the building stock, it must be able to work well very different kinds of buildings, with different equipment, sensors, climates, architectural designs, and climate-system designs. We discuss our approach for verifying various aspects of the building context captured in the metadata, through the data. We also describe how we identify malfunction in a general fashion across very different buildings.

2.6 Summary

In this chapter we discussed the historical motivation for the design and implementation of building information systems. We showed how they were built with the purpose of supporting the building manager's primary job of maintaining the building and dealing with occuppant complaints. A BMS displays the data from a large sensor deployment and show it in the context of the building schematics to the building manager at a central location (i.e. their office), so that they can find problems through a series of clicks, rather than physically walking around the building and inspecting individual sensors and systems.

We discussed the notion of building "app-ification" in greater detail, separating the BMS intwo three distinct layers and focusing on the graphical interface as just one possible interactive modality for buildings. We explaine that if we view the interface as just another interface, then the narrow waiste of interaction if set at the naming layer. We also discuss

several emerging building applications: the graphical supervisory control app, the holistic building optimization app, and the mobile auditing app. We show that the current approach does not contain the necessary layering or architectural components and mechanisms to support the emergy applications and consider what components *are* necessary in order to do so.

We also introduce the notion of verification of building elements through software. We talk about the relationships betweens the systems and spaces, expressed in the naming of sensor streams and how these relationships are used to interpret context necessary for emerging applications. We argue that verification at the sofware level, should be part of any architecture that manages the building. We give an example through model predictive control, and how the mis-representation or staleness of information can lead to gross control-related and energy-accounting errors, especially long term.

We have designed and implemented a system called StreamFS that contains each of these features as components in its architecture. For points 1, 3, and 4 in section 2.4, we observe that filesytem constructs solve these issues quite effectively. We show how a uniform, hierarchical naming scheme combined with a high-level pipe abstraction can be used to address issues with naming, processing, and management of many streams. We discuss how we translate these into different kinds of files in StreamFS, each with their own read/write semantics and how you can construct complex processing pipes easily through this abstraction. Point 2 requires a traditional timeseries database approach. In chapter 3, we explain how the time-oriented, scan-heavy workload calls for a timeseries data store and how we combine the datastore with the access semantics of specific files in StreamFS.

We consider point 5, an important component in our archictectural proposal. However, we separate it from the StreamFS architecture because it is a component that functions largely independently of *how* the naming constructs are logically constructed. Instead, this piece of the thesis focuses more on the mathematics and methodological underpinnings of a verification process that should be included in *any* architecture that describes physical relationships and monitors functional behavior of equipment in buildings. We leave the details of our approach and methodology to chapter 4.7 and the results are presented in chapter 6.

Chapter 3

StreamFS System Architecture

StreamFS is a system that addresses some of the shortcomings in the BMS architecture. StreamFS uses filesystem constructs to represent *all building information*. This includes sensors, actuators, location, processing, streams, categorical organization, etc. It borrows several mechanisms used in a Unix-style filesystem: files, folders, and the pipe abstraction for processing streaming data. It also employs the principle that *everything is a file* that all interactions are through the filesystem. This eases management of both the raw data and the processing elements that produce derivative streams for further processing. In this chapter, we give an overview of the architecture – all the components, their organization, and their interaction. Throughout this chapter, we refer back to the list in section 2.4, where we enumerate the shortcomings in the design of the BMS architecture in the context of building "app-ification". We also discuss how each component provides one or more of the system properties we aim to provide – extensibility, scalability, generalizability, and ease of management.

3.1 Overview

Each component in StreamFS addresses the fundamental shortcomings discussed in chapter 2. The four main components are highlighted below:

1. **Name Register**: The name register maintains both the object id namespace and the hierarchical namespace that it expose to external applications. It also maintains an entity-relationship graph that is used to support indirect relationships between objects. The name register manages various object types as well, which we will discuss in Chapter 4.

2. **Subscription Manager and Forwarding Engine**: The subscription manager manages the input stream and output paths to data-processing sinks. It is part of the publish-subscribe subsystem. The forwarding engine is used for internal and external

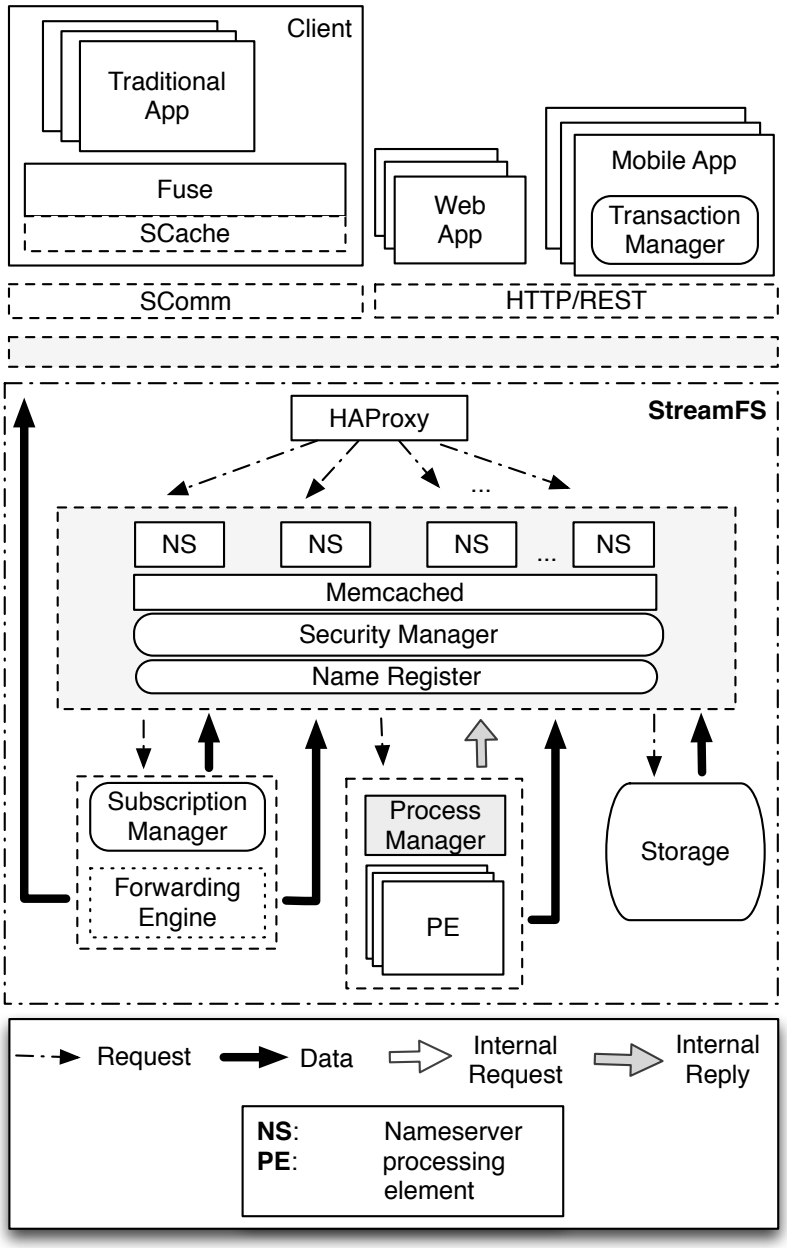


Figure 3.1: StreamFS system architecture. The four main components – name register, scription manager/forwarding engine, process manager, and timeseries datastore – are shown. It also shows the application layer at the top.

data processing. It queries the subscription manager and name register to determine where an input datapoint should be forwarded to.

3. **Process Manager**: The process manager manages the internal and external processes

running and managed in StreamFS.

4. **Timeseries datastore**: The timeseries datastore functions like the NTV storage engine, as discussed in section 2.4.

StreamFS is built as a web service that resides in the cloud. It has several interfaces; one is by direct TCP socket communication and the other is RESTful [65] over HTTP. We use HAProxy [34] to scale the service as it grows. We also design each component to be horizontally scalable; to grow with the size of the deployment and the number of applications. Figure 3.1 gives an overview of our architecture as well as the application layer.

3.2 Name Management

The name management layer addresses point 1 in section 2.4. It provides a high-level narrow waist for access sensors in context specified in the name itself. StreamFS manages two namespaces. The first is a flat namespaces that identifies a particular object instance. The second is a hiearchical namespace that identifies the current instance of a particular object in some context, specified by the path for the object. We support two namespaces in order to uniquely identify sensors and actuators while supporting multiple names.

Multiple names is a requirement in building applications. Sensors and actuators can be accessed in various way, depending on the application. Some applications wish to access the sensor in the context of its placement in space. For example, the path `/soda/4F/410/temp` can refer to a temperature sensor in 410 of a building named 'soda'. The same temperature sensor drives the actuation profile for the HVAC system that serves room 410 and for other applications it may be important to access it in the system context through a name that specifies that relationship, such as `/soda/hvac/ahu1/temp`. In the rest of this section we discuss how both namespaces are managed and implemented.

Object identifier namespace

Each new object that's created is assigned a 128-bit unique identifier that uniquely identifies the object. The namespace is large enough to support to many object with low probability of collisions, even across StreamFS instances. Because StreamFS support multiple names that refer to the same object, we created a namespace that is flat and large enough to uniquely identify objects in the building uniquely. The unique identifier is randomly constructed and the probability of colissions is small because of the large namespace. StreamFS only assigns a unique identifier to stream files. We will discuss the various file types in section 4.7.

Hierarchical Namespace

Hierarchicaly naming schemes are an effective way to organize information, particularly for a relatively small amount of information where the access patterns are well defined and groups across buildings have a lot of overlap. For example, upon close examination of the naming scheme for points across Sutardja Dai Hall, Soda Hall, Cory Hall, and the University of Tokyo Engineering Building 2, we observe that there are 2 overlapping group types. All the point name refer to the location of the sensor and the system that it is associated with. For example, 'SODA1R430A_ART' encodes the name of the building and the room number but also encodes the HVAC subsystem id – referred to by the 5th character which is a '1'. The other common encoding include the type of sensor and implies the S.I. units of measure. Based on our experience with anlysis jobs on building sensor data, we decide it was less import from a naming persepctive than an interpretive one.

The number of sensors in the building can easily grow into the thousands or ten of thousands. We apply the principles articulated in [70], which asserts that hierarchical organization of files is ineffective when dealing with a large number of files and that databases are poor at providing direct access to the data, but provide a good way to find the information we are looking for. We combine these two, as suggested by the authors. We expose a hierarchical namespace that gives the user direct access to the data through a familiar organization of that data. The organization itself is directly traversable. Moreover, we separate the metadata from the naming structure, so that users looking for various kinds of information can quickly locate it.

The decision to separate these also gives our implementation *better scalability*. The growth of the namespace, metadata, and data happen at different rates. We can tailor the acquisition of information about the files depending on the type of information being requested. For example, if the query is metadata-related, we send it to the metadata management cluster, which not only stores the metadata, but also indexes it accordingly. Furthmore, the namespace is easily extendable and provides a natural way to group items which eases *management and access*.

We discuss our naming structure in more detail in Chapter 4.7 and give an overview of fundamental challenges that emerge when naming must reflect physical associations and the physical environment is changing.

Implementation Details

The name management layer is implemented behind HAProxy, an open source load balancer. The implementation includes a name registry and a name server. Several name server handles requests that are forwarded from HAPRoxy to one of the name servers. Each of the name server knows of each of the databases that contains names. In all our deployments, we only had a single server. However, for deployments that are large, we put the names in multiple, replicated databases with a write-through update policy. Reads are done from any

of the database servers, randomly, since they all contain the same information. Each name server has a preference, so the load is properly distributed.

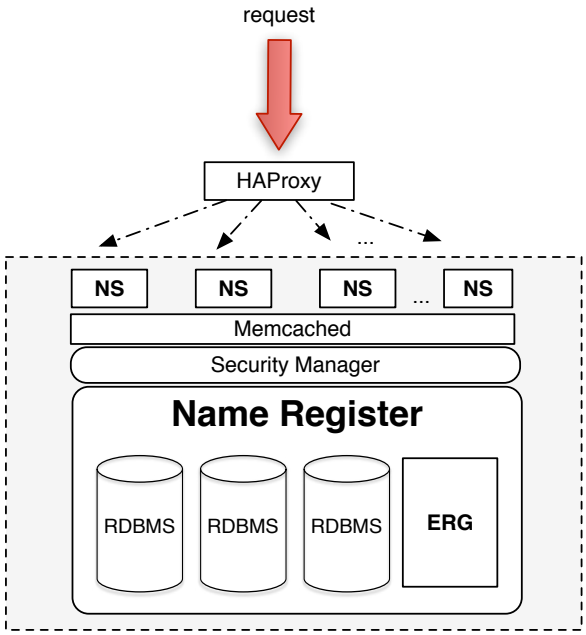


Figure 3.2: Name management layer implemented behind HAProxy. Name servers handle individual requests and use the name registration table query handle the request accordingly.

The write-through policy is implemented with a write lock. Whenever a name server receives a request to create or delete a file it informs the other name servers that it wishes to acquire a lock. To prevent deadlock, we force a lock-acquisition order. A lock is not acquired unless every name server agrees to give up the lock to the requesting name server. Once the lock is acquired, the name server performs the same write on each server. The name server then releases the lock by contacting the other name servers in reverse order. If a name server has given up a lock and not received a release, the lock is released automatically after some time. If a name server goes down, the name server that acquired the lock assumes the release was successful. The name server list is immutable, they are restarted in practice if they go down.

A layer of memcached [58] is used to reduce the load on the databases. Writes immediately invalidate any entries in memcached. We also include file metadata in the memcached layer, so its use reduces the load on both the name register and the metadata data store. The security manager essentially maintains an access control list and set of operations that are supported by each file. By defualt, security is disabled, but some of our deployments did enable it.

The name management layer consists of 3 dependent components, each following the principles of horizontal *scalability*. The namespaces are managed in single replicatable, relational database. The metadata is managed in a separate MongoDB [**mongodb**] database, which is itself shardable. The data associated with streams is managed in a shardable timeseries database. We follow the principle of scal-out system in sub-component for *scalability*.

3.3 Time-series Data Store

The timeseries data store addresses point 2 in section 2.4. Data collected from sensor is timeseries in nature. A sensor produces data periodically. The imporant aspect of the stream are the name of the feed, the time the reading was received, and the value for that reading. There is also metadata that needs to be stored about the stream. For example, we want to know what the units of measure are, the make/model of the sensor, the date it was installed, any calibration parameters or other information that will help the user locate the sensor or interpret it correct. We actively separate the storage of the metadata from the storage of the data.

In constructing a design for the data store, we considered 3 main questions:

1. What is the typical access pattern or what are the top queries?
2. Should we compress it?
3. How is the data stored long-term?

The typically access pattern is that of scans. Many of the applications that we consider that make use of historical data, fetch the data is a temporally meaningful manner. The query specifies the interval of time over which to fetch the data from a particular feed and either perform cleaning operations on the data, display it, or adjust the scan parameters for a subsequent query. The data is largely self-similar and highly compressable. Simple compression tests we ran on real data showed a compression factor between 15 and 30. Also, the data is essentially append-only, forever. It can grow quite large, but grows have a fairly slow rate, especially after compression.

Implementation Details

We chose OpenTSDB [**opentsdb**] as our primary data store. We enabled the compression feature and indexed based on the same and timestamps of the feed. OpenTSDB is a timeseries data store build on HBase [4]. HBase is designed to scale horizontally for very large data sets. OpenTSDB is a good choice because the compression features keep the footprint small/fast while the append-only, forever workload requires a scalable solution provided by the underlying technology.

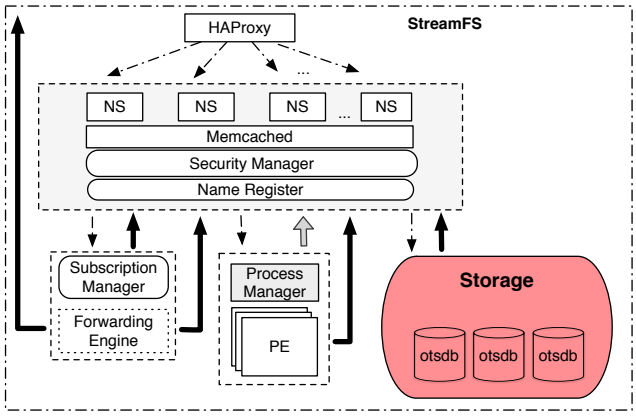


Figure 3.3: The timeseries data store. We use OpenTSDB; a timeseries data-store that runs in a cluster setting over HBase.

3.4 Publish-Subscribe Subsystem

StreamFS uses a flexible construction of the publish/subscribe model in order to support a wide range of applications. It addresses point 4 in section 2.4. It includes both the process manager and processing features that support *online analytical processing (OLAP)* processing features. Publish/subscribe is necessary is physical data application development in order to scale in the number of supported applications. The publish/subscribe model used in StreamFS provides mechanisms that enable a flexible combination of space and time decoupling that enable StreamFS to support of a wide arrange of application requirements, as described by Eugster et al. [26]. Our pub/sub engine is also tightly coupled with the namespaces exposed to users, and this design choice allows an application to control the space coupling between the publisher and the subscriber (similar to TIBCO [73]).

Space decoupling

Space decoupling is achieved when the publisher and subscriber do not know of each other. By its very design, space decoupling is achieved. Publisher do not hold a reference to the subscriber and subscribers do not hold references to publishers. However, because of the coupling of a full pathname and an object, subscriptions to topics expressed as a full pathname refer to the single publisher. Since we maintain a one-to-one mapping between the name and an object, only a single stream can fulfill the subscription request. Space decoupling is achieved when the topic is generalized using the star operator in a a regular expression match. For example, if the user specifies to obtain all the feeds for `/soda/4F/*` then all the publishers that have a name that match that prefix will be fowarded to the subscriber sink. Space decoupling in achieved because the subscriber and the publishers are unknown.

Time decoupling

Time decoupling is achieved when the publisher(s) and subscriber(s) are not interacting at the same time. Time decoupling is achievable through the timeseries data store. Publisher push data to StreamFS whether or not subscribers are online. Moreover, data may be received at the subscriber even if the publisher becomes disconnected. Currently, subscribers do not receive all information that was missed. In order to achieve fill time-decoupling, we allow the subcription target to enable or disable the option to buffer all missed readings for an associated subscription target, while the subscription target it offline.

Synchronization decoupling

Synchronization decoupling is when publishers are not blocked while producing events and subscribers get asynchronously notified of an event while it is engaged in another other process (i.e. it is not blocking on wait). Sychronization decoupling is achieved by the publisher and subscribers through StreamFS. Events are received out of sequence from their arrival to StreamFS. This is true even when the subscription target is a processing element. The thread that buffers incoming data for each processing element is seperate from the thread where the process is executed.

Implementation Details

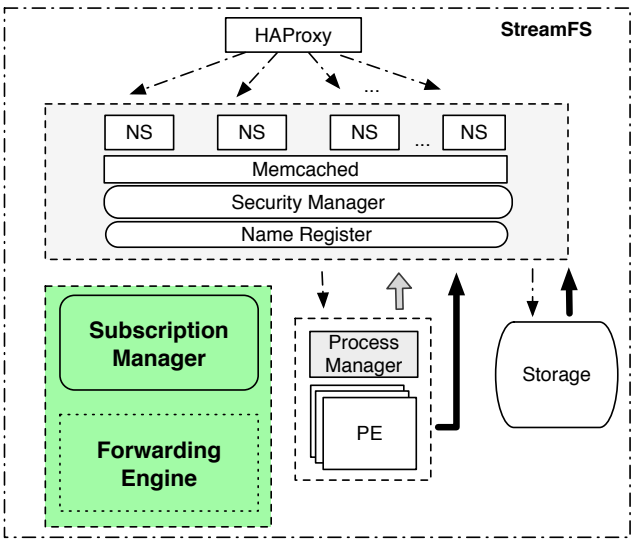


Figure 3.4:

The pub-sub system manager works with the name register to determine where to forward the incoming data. When data arrives, it arrives with a tag that contains the name of

the publisher. The subscription manager resolves the name to an object identifier and resolves all the names for the object. Then, it scans each of the names and matches them to a subscription. If the subcription matches *any* name, the data unit is marked with the subscription id and sends to the forwarding engine. The forwarding engine then contact the forwarding sink and send it the data unit. The subscription sink may be a processing element managed by the process manager. If so, it is forwarded to the process manager's buffer and the process manager copies it to either an internal buffer for a process that is running locally or sends it to an external process stub, which copies it in an internal buffer on the client side.

3.5 Data Cleaning and Real-time Processing

StreamFS provides sophisticated mechanisms to process data in real time. Processing features in StreamFS address point 3 in section 2.4. Sensor data is fundamentally challenging to deal with because much of it must be cleaned before it can be processed. For example, it is not uncommon to receive readings that is out of operational range, that is erroneous with respect to the previous observed trend, or to stop receiving readings altogether. This implies the need for processing jobs to provide a level of filtering over the raw streams. Once the data is cleaned, it is typically consumed by more sophisticated processes that aggregate or use it for control of equipment. We provide the mechanisms for handling both classes of processing jobs with our process management layer. We address *re-sampling* and *processing models*. The incoming data does not have a common time source, so combining the signals meaningfully involves interpolation. There are various options that we provide for performing the interpolation, chosen by the user depending on the units of the data. For example, temperature data may involve fitting a heat model with the data to attain missing values in time.

For jobs are need to clean the data and wish to run short-lived, simple operations, we provide an interface for *internal* processing. The user submits a job and we schedule it in a machine in the processing cluster. For jobs that are more complex and require client-side lirbaries, we offer a facility where the process is allowed to run on the client side, but is entirely managed by StreamFS. We provide a client stub that essentially runs like a mini-job scheduler on the client side and communicate with StreamFS to execute file operations that affect locally-running jobs. We discuss the details for both kinds of jobs in section 4.3 and 4.4.

Finally, since data is coming in at different rates from different sensors and is produced asynchronously from processing elements. For certain processes, processing the incoming data as quickly as possible is key, however, this is challenging for several reasons: 1) a process may subscribe to multiple, independent streams with asychronized report schedules and 2) interpolated values should be avoided to minimize prediction inaccuracies in interpolated values. Therefore, a process actually wants all the freshest data from all the streams they are subscribing to, while minimizing the average time that the data for each respective stream

has been waiting in the buffer. We address these problem through a freshness scheduler that is presented in section 4.5.

Implementation Details

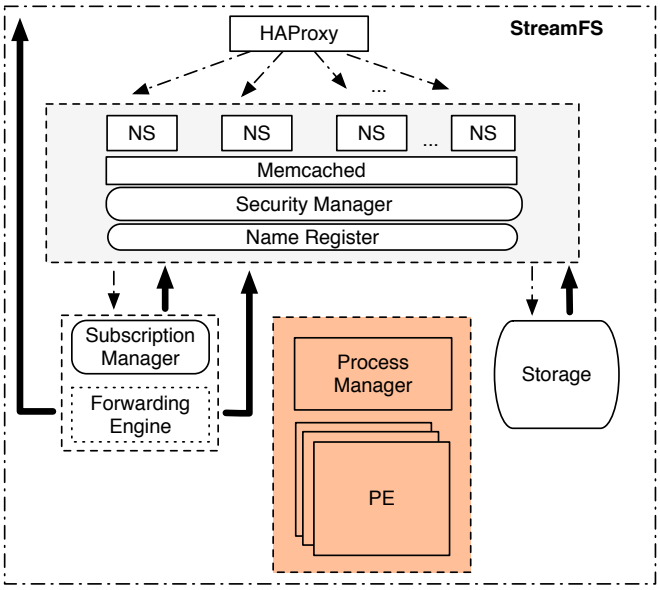


Figure 3.5: The process manager manages a cluster of processing element and connection to external processing units. It works closely with the subscription manager to forward data between elements.

The process manager works closely with the name register to manage process definition files and process instances. Process definition files are those that are submitted to the server by the user, that define a function to run on the streaming data. Once data is piped to the process definition file, the process manager spawns and instance of the definition file on one of the process-element (PE) execution servers. The PE creates a buffer for incoming data and sets up a job to run periodically according to the specification for the internal processing job. The internal processing job is mapped to a file that is accessible in StreamFS. It contains various statistics about the job that is running, such as the streams that feed it, the last time it ran, the amount of time it took to run, the period of execution, etc. If the user deletes the file, the process manager contact the corresponding PE server that contains the job and the job is killed. Once the job is killed it informs the process manager which informs the name register to remove the file.

For jobs that are more complex and need to run externally, we created an client-stub that runs like a mini-PE. It spawns a job on the client side when a user pipes data into it. It manages all instances of running jobs on the client server and it processes requests associated

with operations on the corresponding instance file represented in StreamFS. Figure 3.5 shows the components of the StreamFS architecture that handles all processing elements.

3.6 Entity-relationship Model

StreamFS supports symbolic links, which allows the user to specify non-hierarchical relationships. Indeed, they can be used to express relationships which form a directed graph. We find that many applications, such as control applications and applications that perform aggregation, preceed their timeseries queries with multiple queries to determine indirect relationships between streams. Fundamentally, the queries were use to acertain a multiple-hop relationship between streams and could be easily and efficiently answered with a single graph query. StreamFS maintains an entity-relationship graph (ERG) that uses the names and symbolic links to construct a queryable graph.

The ERG is used throughout our architecture; for answering graph queries by the user and for subscription topic-matching. The latter is a slight twist to the topic matching that is done in traditional pub-sub systems. We discuss the details of our approach in section 3.4.

ERG to OLAP

Many of the operations that users perform are OLAP-style (online analytical processing) operations[33]. OLAP typically requires that you build a logical hypercube along multiple dimensions. The values in the cell are called measures. Queries makes use of the cube to aggregate data along those dimensions. We provide OLAP-style queries using the ERG. This is similar to the work proposed by Chen et al. [12]. The time dimension is fixed, the categorical dimension is determined by the hierarchy, and the unit are specified by the user.

Unlike a typical OLAP cube, our "cube" is dynamic. As new data comes in, it is immediately used. Its arrival triggers the aggregator to interpolate values for all other related streams at that point in time, aggregate them, and update all the measures in the cube. We find that many applications that require analytics, typically require OLAP style queries. For example, consider the problem of tracking the operational energy consumption of a building. It requires the ability answer a series of questions about energy flow – energy data aggregated across multiple categories to determine how, where, and the amount used. The ability to *slice and dice* the data allows the analyst to gain better insight into how the energy is being used. Below is a typical list of questions:

1. How much energy is consumed in this room/floor/building? On average?
2. What is the current power draw by this pump? cooling tower? heating sub-system? Over the last month?
3. How much energy does the computing equipment in this building consume?

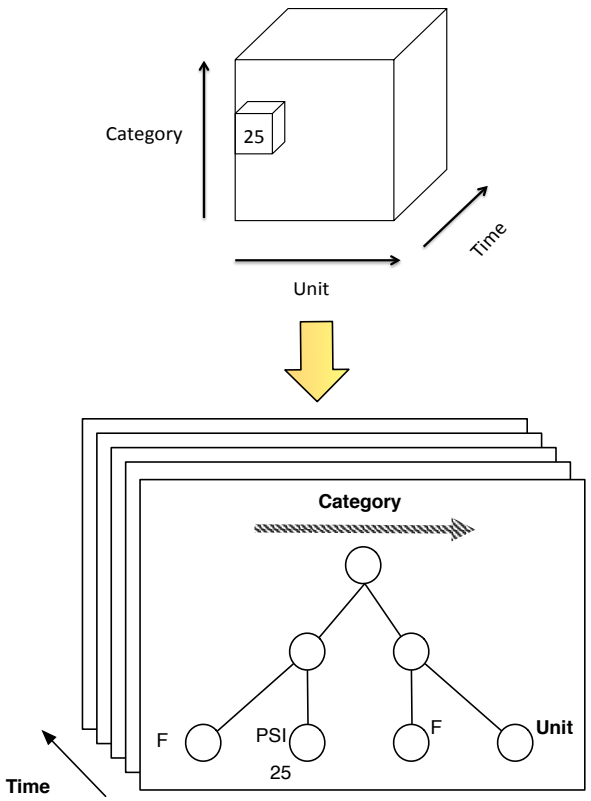


Figure 3.6: This figure shows how we translate the OLAP cube to a hierarchical ERG. Note how the dimensions of the cube translate to the graph. The level of the subtree is the category, the unit is specified at the node, and there are values at each node for every time slice.

Notice, these high-level questions translate into categorical and spatio-temporal queries. There is also a hierarchical grouping pattern. For example, to answer the first question we can aggregate the data from the individual rooms up to the whole building. Also, the cooling system consists of the set of pumps, cooling towers, and condensers in the HVAC system that push condensor fluid and water to remove heat from spaces in the building. We can model this as a set of objects and inter-relationships which inform how to *drill-down*, *roll-up*, and *slice and dice* the data – traditional OLAP operations. Figure 3.6 shows how we translate an OLAP cube to a hiearchical arrangement of nodes in the ERG. Time-range queries on any node translates to a slice in the cube along the time dimension. *Pivoting* along any dimension can be done by taking slices along the "cube" in the bottom of that figure. We describe how this functionality is useful in performing energy audits and describe its use in a mobile energy auditing applications presented in Section 5.2.

The entity-relationship model helps simplify these problems, both as an interface to the

user and a data structure for the aggregation processes. We argue that using the ERG in the building context is reduces the cognitive load and makes the formulation of such queries easier. The inter-relationships are explicitly specified and this allows users to maintain a structure that is more natural. Also, it has been shown that a relational model loses this information [**SenkoDB**].

Implementation Details

The graph is maintained in memory. A typical deployment contains about 10k nodes, so the memory footprint is not that large. In our deployments we kept the unit on its own machine with 4 GB of memory. The footprint is typically several hundred to about 1 GB in size. We used a JUNG graph library [43] to maintain the graph.

To achieve horizontal scalability, we propose that each ERG unit should be separated into distinct nodes in a cluster and graph queries should be answered across the cluster. This is a topic for future work, however, since a typical building deployment does not require that level of scale. A collection of buildings, managed through a single instance of StreamFS, would likely require a clustered implementation of the ERG.

3.7 Related Work

StreamFS borrows ideas from many places. Naturally, in adopting filesystem as the main abstraction we borrow directly from the design of unix-style filesystem and pipes [66]. We translate these ideas into our implementation, transforming these into a web-services architecture combined with mutliple databases. Our filesystem is also hierachical, but it is *completely divorced from the approach to storage*. File information is spread across multiple databases in StreamFS. The name is made accessible through a RESTful interface or socket connection, but only the name and its structure is used to identify the object. Moreover, the notion of symbolic and hard links are used as a way to support multiple names for the name unique object. In traditional filesystems, these inform the access pattern to the file on disk. Our filesystem API is *not* POSIX compliant, since we introduce new file types and associated file-specific semantics. We discuss these in detail in the next chapter.

StreamFS constructs an entity-relationship graph [**Chen76theentity-relationship**] from the files in the system, in order to represent different physical and semantics relationships. We use it throughout the platform for analytical processing and querying. We couple the namespace that is exposed through the filesystem with an in-memory graph that graphically represents the inter-relationships between the files. This graph is used to provide OLAP-style queries, similar to the work by Chen et al. [12]. These authors describe how they translate OLAP operations to a graph. Their decomposition is different in that nodes in the graph explicitly represent measures and dimenions. We do represent the time dimension as a separate snapshot of the graph at another point in time. By separating the raw data from the graphical representation this becomes a trivial transformation.

The use of a filesystem representation has been explored by researchers. Tilak et al. [74] use the filesystem interface as a way to represent different naming conventions for a deployment of sensors. StreamFS in similar in that it also adopts the construct for naming purposes. StreamFS also adopts it as a way to expose a uniform access layer for sensors and actuators in the world. Actuator, in many ways, are also modeled like a periphal device. However, StreamFS is more general and adopts more filesystem features and principals. They do not support symbolic links and they couple the network organization of a wireless sensor network with the naming structure. Filipponi el al. [80] create a mountable filesystem that is POSIX compliant, making the deployment nodes look like a peripheral device that can be directly written to. In contrast to both pieces of work, we do not attempt to make a POSIX compliant filesystem. We adopt *conceptual* filesystem construct and re-interpret them as features in a web architecture. We tightly link the notion of a sensor and its data through the special file types and operations to handle sensor data queries and organization.

The foundation of our processing framework is a pub/sub system architecure, similar to [25, 68, 81, 73]. StreamFS has a flexible pub/sub coupling architecture, where user options can separate certain dependencies between producers and consumers. We also make use of the ERG to do topic-matching. This is crucial for building applications, where analysis is tightly coupled with context. In addition, our processing framework revisits many issues related to dataflow processing systems [17, 50, 10]. StreamFS is focused on providing a tool that cleans the incoming data and provides real-time data to consumers/applications for the building. It is not focused on true-real time constraints or modeling.

HomeOs **??** is related. So is BOSS [18] and FIAP [63] and BuildingDepot [1].

3.8 Summary

In this chapter we gave an overview of the main components in StreamFS. Each of the components addresses the concerns stated in section [**sec:shortcomings**]. The filesystem name server expose a uniform namespace for access sensors and actuators in deployed throughout the building. The timeseries database serve to store data streaming physical information and is optimized for the scan-style queries posed by applications. These address points 1 and 2. We also include a pub-sub system which serves multiple purposes. It provides real-time data forwarding for external applications and forwards data internally to processing units that are specified or linked by the user. This addresses points 4. Finally, we introduce processing elements, both internal and external to address point 3. We also introduce an entity-relationship graph to deal with indirect relationships that are expressed in the contruction of names in the system.

In the next chapter we talk more about processing and discuss the details in the scheduler that help enable applications that have certain delivery requirement.

Chapter 4

StreamFS Mechanisms

4.1 Process Management and Scheduling

Processing of sensor data is a fundamental component of any analytics engine. Sensor data is often dirty [7] and must be cleaned before sophisticated processing jobs can consume it. In this chapter we discuss the details of our process manager and process scheduler. We describe how our process manager handles both internal and external processes and manages the buffers for all the jobs. We give a detailed description of the job-specification API and describe how the user interacts with StreamFS in order to activate, management, and de-activate jobs. We also describe a class of applications that require the freshest data and present a new algorithm for providing the freshest set of data points to those jobs, in a timely fashion.

4.2 Process Management

The process management unit in StreamFS manages jobs submitted by users. There are two different kinds of jobs: internal jobs and external jobs. Internal jobs are managed internally in a process-element cluster. StreamFS accepts javascript code from the user and maintains it until it is activated. Activation occurs when a pipe process to the specified job is instantiated. External jobs are written in any language that run on the client machine and communicate with StreamFS through the external-process job stub. An external processing job is one that interacts with external code that creates the associated representational files for management of external jobs from a central location.

Processes are managed through the filesystem. Our design is guided by the principal that everything is a file and every entity is represented in the filesystem. We use the term subscription and pipe interchangably, however, for explicit disambiguation, we define a subscription as a one-way forwarding process of stream data to an external target. A pipe is a type of subscription to a stream from an instance of a process file. The process can be internal or external and *always* has its output represented by a stream file. The latter allows

us to construct processes chains that can be linked via their associated stream files. This section discusses the difference between internal and external processing jobs.

4.3 Internal Processes

Internal processing jobs are short jobs submitted by users that are managed by StreamFS. When a user submits a job they specify the name of the job in the request. The process definition is checked for syntax correcteness and if it is okay, then it is accepted. All newly created job definitions are placed in `/proc/`.

Listing 4.1: Simple aggregator process job.

```
function(buffer, state){
    var outObj = new Object();
    var pavg = state.cnt;
    var sum =0;
    for(i=0; i<buffer.length; i++){
        sum += buffer[i].value;
        state.cnt+=1;
    }
    outObj.value = sum;
    state.avg = (state.avg * pavg + sum)/state.cnt;
    return outObj;
}
```

The code in Listing 4.1 gives an snippet of example code that aggregate the data passed to it in the buffer. If the user names this definition `simpleagg`, then the file that corresponds to the job definition is `/proc/simpleagg`. We do not limit check the size and complexity of the job, however, we strongly recommend that jobs be kept relatively short and simple. Generally, we recommend that a job pipeline be established rather than feeding one large chunk of code. It makes the pipeline easier to debug once it is activated.

In order to activate the process, the user must pick a set of streams and initiate a subscription *pipe*. The *subscription manager* notes that the sink in a process definition file and informs the *process manager*. The process manager fetches the code and passes it to a node in the processes cluster. The process manager generates an id and passes it back to the subscription manager. That code is used as a new *instance file* that represents the output of the file and that instance file is created in the process file definition file's folder. So if the id generated is `550e8400` the corresponding file is `/proc/simpleagg/550e8400`. This file instance is a stream file (which we discuss is detail in chapter 4.7), which allows the user to pipe it to another sink. There is also some metadata information that is associated with each of the files created. The instance file contains information about the streams it is consuming and various statistics, such as execution time and execution period. If the user

deletes the file the subscription is deleted and the process manager sends a kill message to the corresponding process element.

The function must have the signature as shown above, where the first parameter is the `buffer` and the second parameter is a `state` variable. It must also return a variable of type *object*. The `buffer` is an array of `data` objects. Each `data` object contains two fields, a `ts` field that is the timestamp for the data point and the `value` field which is the value for the data point. An optional setting is to also include all attribute-value pair values that are part of the metadata associated with the corresponding stream file for the stream. The `state` variable is an object that is passed across executions of the function. This way the function could maintain any necessary state without losing it after a single run of the job. Upon creation, the user specifies other parameters that drive the execution period and/or execution conditions for the job. The user specifies the `window` size and/or a `timeout` parameter. If the `window` is of size k then the job will run when the incoming buffer for the job has at least k elements in it. The `timeout` sets the minimum period for the job to run. If both are specified the `timeout` always runs and the job will also run when/if the buffer has at least k elements.

Process Element

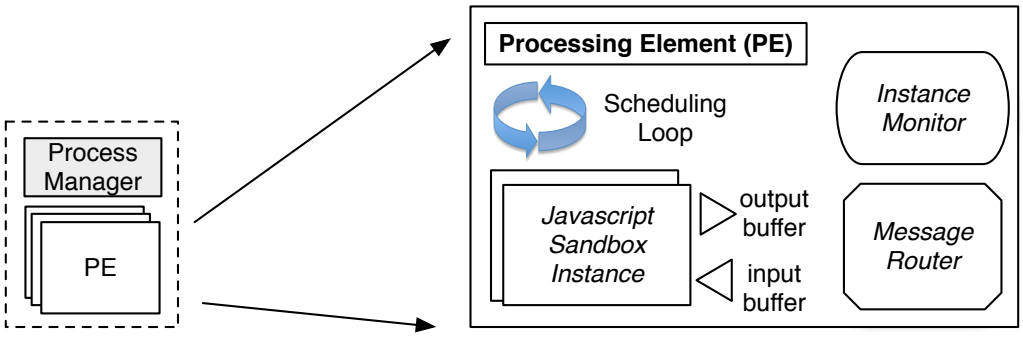


Figure 4.1: This figure shows the internal structure of a process element (PE). The PE consists of several javascript sandboxes where interanl process code runs. It also contains a scheduling loop, message router, and instance monitor.

The process element process that runs on a separate machine. Its main job is to manage the jobs that are being executed on that machine. Figure 4.1 shows the components of a process element. There are four main components in the unit.

1. *Scheduler*: The scheduler manages the execution schedule for each job running in a sandbox. The scheduler uses the `timeout` as the execution period of the job.

2. *Sandbox*: The sandbox is where the javascript code runs. The sandbox contains an input and output buffer and simply waits for either an execution signal from the scheduler or until the input buffer has `window` element in it. The output buffer is of size 1. The object returned by the job is written to the output buffer.

3. *Message Router*: The message router communicates with the process manager and send control and data messages between the PE and the process manager. It monitors the output buffer for each of the elements and forwards them to the process manager when it is populated.

4. *Instance Monitor*: The instance monitor manages the state of the sandbox. If a sandbox dies, the instance monitor re-starts it. If it continues to die, it kills the sandbox process and forwards an error message to the message handler to the process manager. The process manager annotates the instance file with the error.

The process manager keeps a map of which process elements own which jobs. If a process element goes down, the jobs are automatically migrated to another process element and the process element is removed from the active list. Each PE and job have a unique identifier. The messages received from the *message router* in the PE are annotated with these ids and the process manager uses these to update the associated instance file in StreamFS. Again, everything is managed in through filesystem. So errors and data are exposed to external applications through the meta/data in the files.

Note, the process manager maintains a mapping between job instances and PE's. It also notes the machine the PE is running on. The *instance monitor* allows the process manager to make decisions about job migration in case of failure. PEs are designed to function independently from one another. This allows the process cluster to scale linearly with the load – in terms of the number of streams being processed and the number of active jobs. Like the other components in StreamFS, it is *horizontally scalable*. The process manager serves as a single point of failure. If it fails it must be restarted manually. Any messages sent from the PE to the process manager during the down period is lost.

OLAP-style Aggregation

Online analytical processing (OLAP) is a processing layer that provides summarization of data from a set of underlying data repository (date warehouses). Traditionally, OLAP is used to process business data. Business data summarization allows an analyst to ask targetted questions about aggregates and trends in their data. The data is typically multidimensional in nature and operations can be performed with respect to those dimensions and their interrelationship. In our deployments, we find that most queries are similar to OLAP queries with scan-heavy queries across the time dimension.

We introduce a mechanism that can perform hierarchical aggregates across a unit of measure, used in combination with the timeseries data store to support scan-heavy queries, called *aggregation points*. We discuss these in more detail in section 4.3. However, before

discussing aggregation points, we give details on how we make use of the entity-relationship graph and timeseries database to support OLAP-style queries.

OLAP schemas logically construct a hypercube with different dimensions along each axis. We presented a visual translation of a OLAP cube to an entity relationship graph in section 3.6. Specifically, we showed in Figure 3.6 that each dimension is essentially a unit of measure, the hiearchy is explicitly constructed, and there time is a dimension that is stored in a timeseries database. We show how the terminology also translates and some examples of certain classes of OLAP queries, how they are constructed, and how they are satisfied by StreamFS.

Measures, Dimenions, and Levels

Measures refer to the actual value, located somewhere is the cube along the intersection of several dimensions. A *dimension* is simply a reading and an example is shown in Figure 3.6. Dimensions are labels for an axis. In StreamFS a measure is a unit of measure (i.e. Fahrenheit) or time. The hierarchical *level* is explicit in the construction of the names. So in the subtree that organizes stream according to the location of the in space, the floor level would be above the room level. These points can be designated as aggregation points, whereby all streams that have the units are aggregated at the point and store like any other timeseries stream.

Operations: drill-down and roll-up

Drill-down and roll-up are explicit in the structure. You can drill down to individual readings or roll them up into an aggregation point at a particular level in the hierarchy. Essentially the level is an explicitly specified in the name of either the raw stream or the aggregation point.

Operations: Slice and Dice

Slice and dice operations are translated as traversals of the hierarchical structure across levels and units. Figure 4.2 shows how a slice query is satisfied on a hierarchical structure. The corresponding slice query is `query.slice('/4F/R*').start(t1).end(t2);`.

Figure 4.3 shows how a dice query is satisfied on a hierarchical structure. The corresponding dice query is `query.dice('/4F/R*').units(['F']).start(t1).end();`.

Operations: Pivot

Pivot operations are not explicitly supported. Although you can imagine that if you cut across a hierarchical level across all dimension, can can effectively construct a pivot-style query.

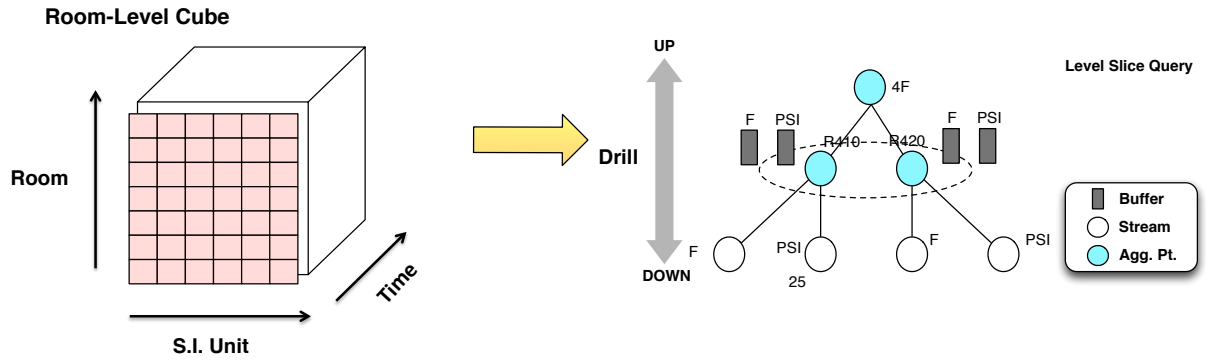


Figure 4.2: This figure shows how a "slice" operation is translated from the cube to the ERG. The user queries across all streams or aggregation points at a certain level, specified by a star level query with the level-specific prefix. The corresponding slice query is `query.slice('/4F/R*').start(t1).end(t2)`.

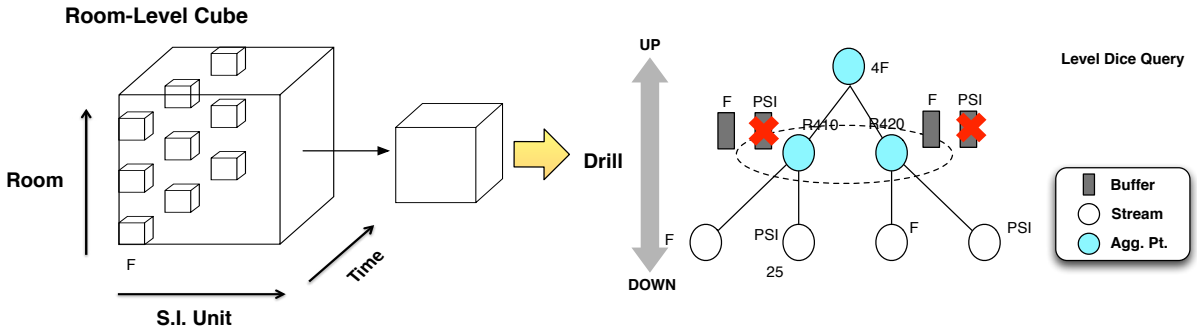


Figure 4.3: This figure shows how a "dice" operation is translated from the cube to the ERG. The user queries across all streams or aggregation points at a certain level, specified by a star level query with the level-specific prefix. The corresponding dice query is `query.dice('/4F/R*').units(['F']).start(t1).end()`.

Aggregation Points

StreamFS makes use of OLAP-style mechanism to provide the end-user with *aggregation points* in the hierarchy. StreamFS distinguishes between nodes that represent streaming data sources and those that do not. Those that do not, however, can be tagged as aggregation points. When node is tagged as an aggregation point all the points root at that node in the tree are set as aggregation points and all streams are aggregated and save to the timeseries

database. Since streams are not synchronized (i.e. they do not produce data at the same time) each time a value arrives from a sensor the value for all other streams is interpolated and aggregated along the *unit* level dimension.

This propogates up to the aggregation point and all of it is saved in the timeseries database. The aggregation function can be specified by the user, with the default set to as `sum`. Other options include *avg, sum, max, min* and a custom function that can be specified by the user.

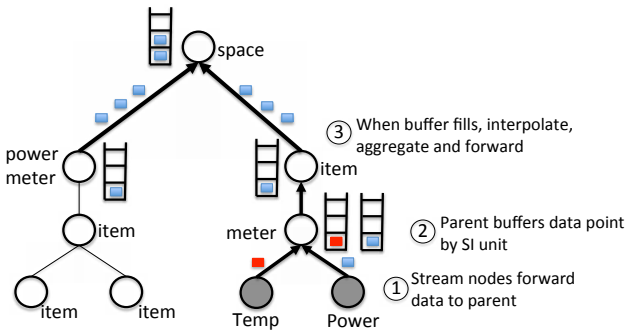


Figure 4.4: This shows an illustration of the aggregation tree used by *dynamic aggregation*. Data flows from the leaves to the root through user-specified aggregation points. When the local buffer is full the streams are separated by source, interpolated, and summed. The aggregated signal is foward up the tree.

Consider the following example. We need to compute aggregates of KW data and we declare the node for a particular room as the point of aggregation, we accept data from all children of that node that, whose units are in KW, and add the streams together over pre-defined window size or pre-defined timeouts.

The scheme is hiearchical, so a node only accepts data from its children and only sends data to its parent. StreamFS checks for cycles when before node insertion and prevents double-counting errors by only allowing aggregation-points that are roots of a tree that is a sub-graph of the entity-relationship graph. In our deployment, each view is a managed as an independent hierarchy. So the hierarchy of *spaces* is separate from the *inventory* hierarchy or the *taxonomy* hierarchy. This allows us to ask questions with a particular view in mind, without conflict, and is a natural fit for our aggregation scheme.

Although there are different semantics applied to different file types at the application layer, dynamic aggregation mainly deals with two types of files: (1) container nodes and (2) stream nodes. The main difference is that *container* nodes are not explicitly associated with data coming from a sensor and *stream* nodes are. Furthermore, container nodes can have children, while stream nodes cannot. In our application, meters are represented by container nodes and each stream of data they produce is a stream node.

When an aggregation point is enabled, dynamic aggregation places a buffer at the node for the type of data that should be aggregated. If we want to aggregate KW data, we specify the type and send an enable-aggregation request to StreamFS. The flow of data starts at the leaves when a stream node receives data from a sensor.As data arrives it is immediately forwarded upstream to the parent(s).

Ignoring the timeouts for now, lets imagine the parent is a point of aggregation and its buffer is full. At this point the parent separates data into bins for each source and cleans it for aggregation through interpolation. The main operation is to *stretch* and *fill* that data with linearly interpolated values. The *stretch* operation orders all the timestamps in increasing order and, for each bin, interpolates the values using the first (last) pair of data points. If there is only a single data point, the stretch uses it as the missing value. The *fill* operation find the nearest timestamps that are less-than and greater-than the missing sampling time, uses their values to determine the equation of a line between them and interpolates the missing value using that equation. Once this is done for each signal, the values are added together for each unqiue timestamp and the aggregated signal is reported to the parent, where the operation occurs recursively to the root. Figure 4.4 shows an illustration of this processing structure.

Dealing with dynamics

This approach deals with changes in the graph quite naturally. All aggregation points deal only with local data, so a node is only concerned about the children that give it data and the parent to send data to. As objects in the environment move from place to place and these changes are captured, the entity-relationship graph also changes to reflect the move. This change in aggregation constituents is naturally accounted for in the aggregate. If a child is removed, it no longer forwards data to the old parent, therefore the aggregate will reflect that change. Note, however, that changes in the entity-relationship graph are indistinguishable from energy-consuming items that have been turned off. For the purposes of aggregation, that is okay.

OLAP-style aggregation is an expensive feature. Each point in the subtree rooted at the aggregation point is automatically activated to produce streaming data. Special internal processing elements are enabled to compute the aggregates and run the pre-processing and communication between the processing unit and name server increases as data is returned from the process elements to be routed to the timeseries data store. The amount of processing and storage scales linearly with the number of streams and the number of aggregation points in the sub-tree.

4.4 External Processes

In real deployments, users do want to be limited by the particular libraries that are available to them in javascript or they have already made a signficant investment in time

writing and testing their own processing jobs. For them, we provide an external client stub that re-directs data from standard in/out through a network connection to/from the StreamFS process manager. The stub also interprets process management commands to spawn and kill jobs and associate different subscriptions with different instances of a jobs running on the client side.

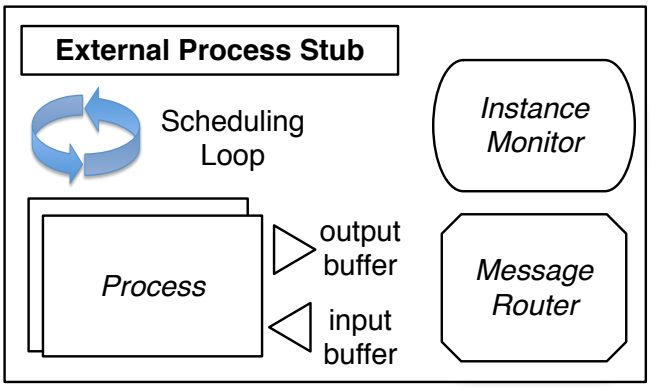


Figure 4.5: External process stub. Note, it contains similar component to a process element and functions much the same way, managing the buffers, scheduling, errors, and communication on the client side like the PE.

Figure 4.5 shows the internal components of a client stub. Note, it contains very similar components to a process element and work in a similar fashion. The client stub contains the four major components:

1. *Scheduler*: The scheduler schedules the execution of jobs on the client machine. It uses the `window` and `timeout` parameters to set when and how often the job is set to run.

2. *Process*: The process component is simply in charge of managing communication with the process that is spawn on the local machine. It maintains an pipe to each local process. The processes run indepdently and share no memory directly. Data from the input buffer is copied to the pipe connection to the process.

3. *Message Router*: The message router maintains a socket connection to the Process Manager and writes to the associated buffer for specific jobs.

4. *Instance Monitor*: The instance monitor re-starts jobs when/if they fail and forward local errors to the process manager to annotate the associated files in StreamFS.

On startup, the client stub read a local configuration file that specifies the path to the job and metadata that describes job. These are used to register the job with StreamFS and set the metadata attributes. The registration on the StreamFS server is exposed through

an *external process* file. The user interact with an external job exactly the same way they interact with an internal process job. In order to spawn a job on the client, the user simply "pipes" a stream file to the external process file. The creation of the pipe/subcription send a spawn message to the client and starts the associated job on the client. Once the process is started, data from the stream(s) is forwarded to the client, which writes it to the standard-in of the client job. Starting a job also creates a stream file the StreamFS server. Any data that's produced by the job and written to standard-out is re-directed to the server and made available through the stream file.

This desgin is consistent with the semantics of pipe/subscription management and functionality. Recall, internal processes work the same way and this allow us to stay consistent with the file-centric principal whereby *everything is managed through the filesystem itself as a file*.

4.5 Freshness Scheduling

Process execution parameters are set by the user when a process element is created. Generally, job scheduling is strictly driven by these paramters. However, in our deployments we found there were job pipe sinks that required a *minimum freshness* property for the set of data points in the buffer upon consumption. In this section we discuss our scheduling algorithm in relation to providing this property.

Maximizing Data Freshness

Data is coming in at different, independent rate from sensors and is produced asynchronously from internal processing elements. For certain processes, processing the incoming data as quickly as possible is key, however, this is challenging for several reasons: 1) a process may subscribe to multiple, independent streams with asychronized report schedules and 2) interpolated values should be avoided to minimize prediction inaccuracies in interpolated values. Therefore, a process actually wants all the freshest data from all the streams they are subscribing to, while minimizing the average time that the data for each respective stream has been waiting in the buffer.

Some streams show lots of variability, driven by the underlying dynamics of the system being monitored. For example, the power consumption of an active server or laptop tends to have a very active power profile. For jobs doing aggregation of streaming data, it is often the case that the time when the last reading was received is very different across streams. For example, consider two rapidly changing streams. If you do not use aggregation, the ideal aggregation scenario is to combine the streams from the latest readings for both streams. That way you minimizing the offset difference between during aggregation. If stream 1 produces a reading every 3 seconds and stream 2 produces a reading every 2 seconds, and you aggregate the readings every time you have at least one reading from both streams, every 3 seconds. Somtimes the reading from the 2-second stream will be 1 second old. If we

Given a full buffer $b[n]$:
for *all elements in the b* **do**
  (1) Calculate the staleness of element i and add to total stalness, S_n;
  for *all other elements in the buffer* **do**
    (1) Determine the next report time D_i for this element;
    (2) Determine the staleness of all the elements if we wait until D_i;
    (3) If it is the smallest staleness figure calculate, replace minimum cost, S_l.
  end
end
if S_l *is less than* S_n **then**
  (1) Wait until later to consume;
  else
    (1) Consume now
  end
end

Algorithm 1: `min_buffer` algorithm.

compute now the total staleness of the buffer is 1 second, if we wait on more second the total staleness is still 1 second, because at t=4 seconds, the 3-second stream will be 1 second old. For applications that wish to display the freshest aggregates with the latest results (smallest buffer staleness), we provide an algorithm called `min_buffer`.

Certain jobs only care about consuming the latest readings from their subscription streams. They are willing to discard reading until two conditions are met:

1. There is at least one data point from each stream in the subscription buffer.

2. The statelness factor is minimized within the immediate time window.

Note, there's a fundamental tradeoff between the staleness factor and variability of consumption. It is sometimes better to wait for the next incoming data point than it is to use what is currently in the buffer, as waiting will decrease the overall staleness factor. Other times, it is better to consume the bufferred data immedaitely. This causes a certain amount of variability in the delivery period to the control process. However, for some applications, this is a reasonable tradeoff to make. Making use of the freshest data is desirable for minimizing errors, either in the control of a system or the calculation of some aggregate state. Generally, the error grows with staleness, therefore the goal of this mechanism is to continuously minimize the error associated with staleness through scheduling.

Let A_i be the arrival time of the last data point received from stream i and D_i be the arrival time for the next data point from stream i and their relationship as described in equation 4.1, where $T(i)$ is the average period between the arrivals from stream i.

$$D_i = A_i + T(i) \tag{4.1}$$

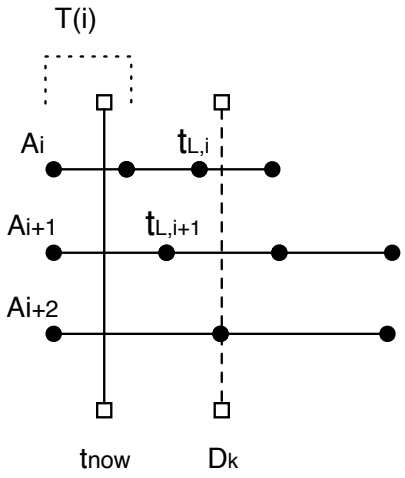


Figure 4.6: Multiple streams in a subscription and their associated parameters.

Periodically, our algorithm runs and checks if there is a data point for each stream in the subscription. If so, the *min_buffer* algorithm runs and effectively decides whether to execute the job on the current buffer immediately or whether to wait until later, when the *staleness factor* of the buffer will be at a minimum. This decision is driven by equation 4.3, whereby we find the next deadline, computed with equation 4.2, for each stream in the set and determine the staleness factor will be for the entire buffer if we wait until that deadline arrives.

$$t_{L,i} = A_i + \left\lfloor \frac{D_k - A_i}{T(i)} \right\rfloor T(i) \tag{4.2}$$

If there is no deadline D_k for some stream k such that equation 4.3 holds, then we execute now. Otherwise we choose to wait until D_k for the stream whose next deadline minimizes the staleness factor of the buffer.

$$\sum_{i=1}^{k-1} D_k - t_{L,i} < \sum_{i=1}^{k} t_{now} - A_i \tag{4.3}$$

Algorithm 1 shows the pseudocode for the `min_buffer` algorithm.

4.6 Dynamic Aggregation Example and Freshness Scheduling Results

We present a demonstration of *dynamic aggregation* and a describe our methodology and evaluation of our *freshness scheduling* algorithm. The first demonstration shows how dynamic aggregation works in a real-world scenario, as a mobile sensor is moved from one

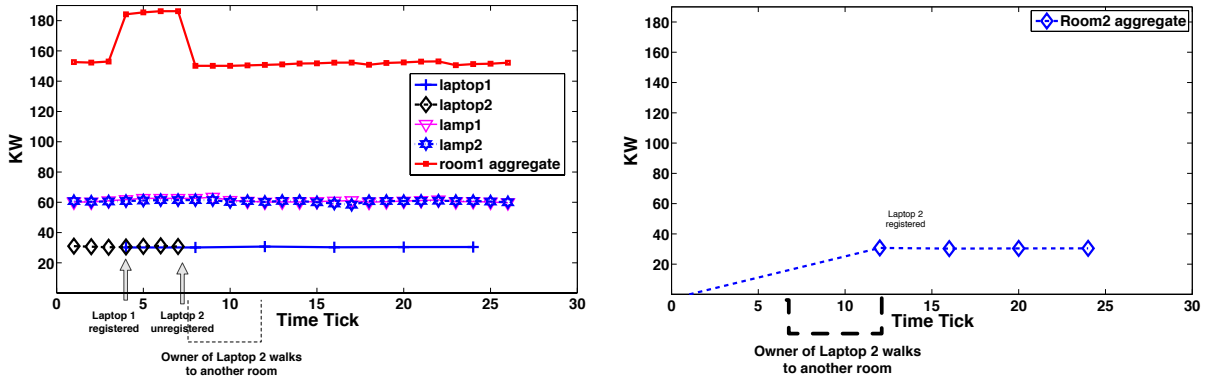
room to another. The evaluation sets up a realistics scenario and measures how the algorithm performs in comparison to the standard scheduling approach.

Dynamic Aggregation Scenario

We illustrate dynamic aggregation with a common usage scenario. It is typical to consider widespread deployment of wireless meters when performing an energy audit in a building. Because such meters are expensive, they are often re-used in order to cature a broad sample of energy-consumping items. We construct an energy auditing application that provides the user with a mobile mechanism for virtually "binding" meters with items, so that accounting of energy can be done correctly. Without record the explicit association between the meter and the item is is metering, the data stream collected from the meter is meaningless. Moreover, it is not just meaningless with respect to the item but meaningless in a broader context of aggregation (i.e. room or floor-level consumption statistics).

For a more specific scenario, imagine there are a number of people in a building, each owning a number of plug-load applicances and a laptop. When a person is in a room their laptop is plugged in and when they leave the room they unplug their laptop and take it with them. As people come and go they attached their laptop to a registered meter and the association is automatically recorded in StreamFS. The meter is constantly reporting readings to StreamFS as well.

We setup this experiment in a home office environment with two rooms and set up the room-level nodes as an *aggregation point* for all the meters in the room. The monitored 2 laptops and 2 lamps and we demonstate the aggregate power draw of both rooms as we move one of the laptops from room 1 at tick 7, walk to room 2, and register the laptop in room 2.



(a) Room 1 object and aggregate streams.

(b) Room 2 aggregate.

Figure 4.7: The power consumes by a laptop in *room 1* is shifted to *room 2* a time t=7. Notice the aggregagate drops in room 1 while it rises in room 2.

Figure 4.7 show the process of dynamic aggregation. Notice that around tick 5 there is a drop in the total power draw curve. As we walk to the new location, the energy consumption remains steady, but lower in room 1 while it remains steady. Then around tick 12, the energy consumption of the room 2 aggregate rises. Note, the level of rise and fall of both curves is the same 30 Watts.

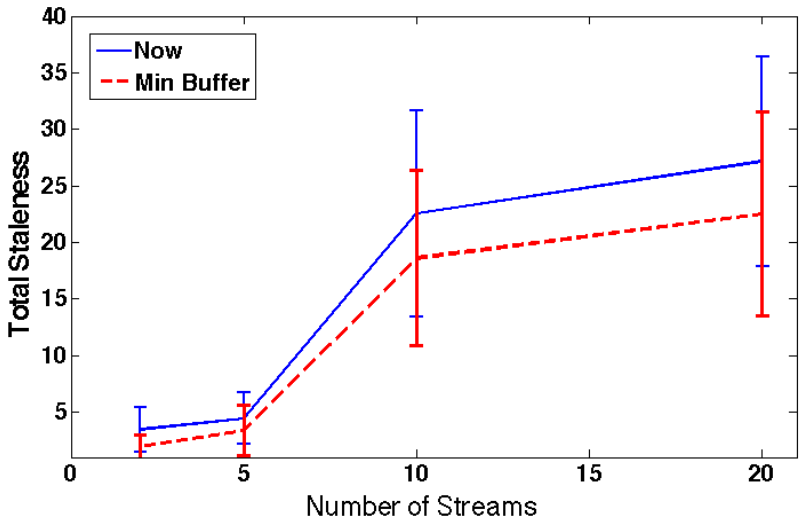


Figure 4.8: This figures shows the tradeoff between staleness and the number of streams being consumed by the job. Note that out algorithm reduces the staleness of the buffer.

Maximum Freshness Methodology and Results

We simulated the effects of the `max_buffer` algorithm shown in algorithm 1. We start up multiple streams and have them feed a subscription with the algorithm option enabled. A single run of the experiment consists of 10 runs with k active streams, where k is between 1 and 20. We let it run for some time and record the total statelness for each run, which is the sum for all data points, between when the data point arrived and the current time. We compare these against the default case where data is delivered as soon as at least one data point from every stream arrives. Figure 4.8 shows the returns of our experiment.

Note, the staleness increases for both cases. The variance of both is also similar, however, the average staleness is lower when the `min_buffer` algorithm is used. It's important to note that this is always true, since the decision is explicitly bounded by the the "now" case. The algorithm chooses between "now" and waiting later for a better staleness calculation.

We also examine the predictabilty in the report rate of the job that enables this feature. Most streams in the system report data periodically with a very small variance around the delivery mean. However, because the algorithm may decide that waiting is better, the delivery period variance is larger. Observe the effects of `min_buffer` versus the default case.

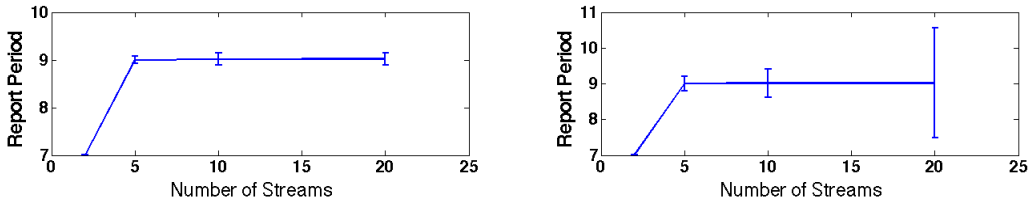


Figure 4.9: This figures shows that the `min_buffer` algorithm provides a similar average execution period but generally at the cost of higher variance in delivery times.

We see that the average are practically the same, however the error bar on the graph on the right are larger. This should be considered when the feature is enable. If there is a job that consumes feeds from the output of a process that has enabled this, it should tolerate delivery times with variable delivery rates.

4.7 Naming and The Filesystem Metaphore

From our experience with building applications, there are a clear set of requirements that are necessary. Most applications construct the notion of context using the naming convention ascribed to a sensor stream. The name conflates the notion of system, space, and type information. At the very least, these three should be supported, however, often other categorical needs must be met to perform various kinds of aggregate statistical, analytics, and control. In addition, we need to support the management of processing jobs that process stream data and provide integrated management facilities for them.

Building applications are essentially monitoring and control applications built on the streams generated by sensors embedded through the building or distillates of them. As the number of applications and streams increased, it becomes desireable to manage them in a centralized fashion. Moreover, the centralized apporach allows all applications to make use of a uniform naming convention and can allow applications to be interoperable. Systems that wish to support such applications require the following properties:

1. Logically accessible physical resources.
2. Representation of data producing and data consuming elements.
3. Representation of inter-relationships between elements.
4. Provide uniform naming and access.

This chapter describes the use of the filesystem abstraction for representing streams in space. The filesystem naming convention provide a a unified namespace to application, for accessing physical resources and streams. Moreoever, we support multi-naming through symbolic links – an important requirement for building applications.

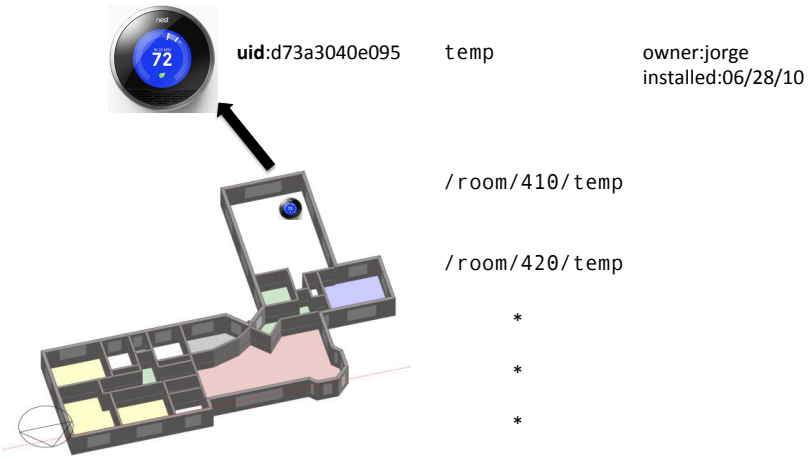


Figure 4.10: Everything is a file. Temperature sensor represented as a file in a folder that contains folders for each room. Note, the file that represents a temperature sensor producing a stream is given a unique identifer. The user may also decorate the file with extra metadata for searching purposes.

4.8 File Abstraction

Our naming scheme is hierarchically structured, like traditional filesystem naming, with support for symbolic links, allowing arbitrary links between sub-trees. We argue that this naming scheme is crucial, as it exposes the inter-relationships which inform aggregation semantics intended by the user. We introduce a naming scheme for physical objects and their inter-relationship.

Similar requirements to those aforementioned have been addressed in the design and implementation of filesystems. Filesystems provide logical access to physical resources through files, with different files and associated semantics, exposed to applications through a shell or programmtically. Filesystems representat collections of bits, encapsulated by a file, and grouped with folders. Symbolic links support the notion of multi-naming. A single file or folder could have multiple names that lead to the same underlying object. Filesystems even support the notion of streaming data through character and block device files. Moreover, pipe files allow programs to communite with each other through a piece of shared memory, where the source application writes to the pipe and the sink application consumes from the pipe.

We assert that these constructs should be directly adopted for supporting applications in the buildings. Our approach adopts the unix file philosophy where everthing is represented as a file. Each object created in StreamFS is assigned two names, by default, one which uniquely identifies the object and *not* human-readable and the second which is changeable and human-readable. Consider the example shown in Figure [**fig:everythingfile**].

In this example, the user is creating a temperature stream file in every room of the

building. The name of the file, given by the user, is *temp*. Upon creation, the file is uniquely identified by the system using a unique identifier, as shown. Like in a unix filesystem, the file is created within a folder. Ideally, the name of the folder would encode the placement of the sensor. In the figure, the user is create a temperature stream file in room 410 and room 420. Note the full filepath for the stream file is /room/410/temp. During creation, the user may also decorate the file with extra metadata, also shown in the figure. In this example, they have annotated the file with information about the owner and when the sensor was installed. This metadata is used for quickly locating the file or grouping files that contain similar tags, quickly.

File types and operations

As we map the filesystem abstraction into this problem space, we need to consider the various kinds of files our system will contain, their semantics, and how our system will expose and manage them. There are essentially 4 types of files and 6 sub-types. We summarize these in Table 4.1. There are also different kinds of operations that the each file type supports. Operational semantics are file dependent. For example, when you *read* a folder, you obtain the metadata associated with the folder and the name of its children. When you *read* a stream, you its metadata and the last timestamp-value it produced. *Writ*ing to a stream is a bit different. You can write to a stream to update its metadata tags and the stream source can write a value to it. The stream source is identified with a *publish identifier* (pubid). The stream source includes the pubid in the write operation for the specified stream file. Without the pubid, the source cannot write to the file. Any other writer should not be allowed to write to a stream file either.

type	**description**	**valid operations**
container	Container file. Used to group other kinds of files within it.	read, write, delete
stream	Represents a data stream.	read, write, delete, subscribe, query
controller	Represents a controller.	read, write, subscribe
special	There are several kinds of special files for management of jobs and pipes.	read, delete

Table 4.1: Summary of the 4 main file types and their valid operations in StreamFS.

Similar to a traditional filesystem, StreamFS includes *special files*. There are 6 special files and 5 of them are for management purposes. The only one that is not is the *symbolic link* file, which is essentially used to support multi-naming and inherents the operational semantics of the file it points to. The delete operation on a symlink, however, only deletes

the symlink. A description of these files and the operations they support is given in Table 4.3. A detailed description and examples with be presented in later sections.

operation	**file type**	**semantics**
read	folder, stream, ipd, ipi, epd, epi, sub	read the metadata and tags for the associated file.
write	stream	Write to stream file, only the appropiate stream source is permitted.
delete	folder, stream, ipd, ipi, epd, epi, sub	Folder must be empty. The others can be directly deleted.
query	stream, all	streams support time-range queries. All support metadata-tag queries.
subscribe	stream	Forwards data from a stream to the specified destination.

Table 4.2: File opertaitons, the file types that support them, and their general semantics.

Container, Stream, and Controller Files

A container or folder file serve primarily as a container for other kinds of files. It is used to group together different kinds of file and to represent a common attribute of the file within it. For example, a container file is usually used to construct the spatial hierarchy. Each file at the top level represents a floor, its children are also a set of container files, representing each of the rooms on that floor. A container file cannot be deleted unless it is empty (i.e. has no children).

File that represent streams are called stream files. They are tightly associated with the associated stream data in the timeseries data-store. A user created a stream file and that associated stream "writes" to it in order to have its data saved. StreamFS also forwards the data to the *subscription manager* in case any subscription sink has subscribed to this feed. When a stream file is create an id is returned to the user. This id must be used by the stream that wishes to push its data to StreamFS through this file. If this id is incorrect or not include, the write operation is rejected.

Because controllers accpet many kinds of input, we designed the the file that presents a controller similar to the external processing stub discussed in section 4.4. Writes to a controller file and forwarded directly to the control stub running at the controller itself or a proxy machine that communicates with the controller. Any reply is set as metadata in the controller file. Controller files also an associated output stream. If a controller process wishes to inform the process of internal state at the controller, it does so through the controller file output – a stream file itself. Table 4.1 lists the files in streamfs and the operations they support. The operational semantics are listed in Table 4.2.

Special Files

There are 6 types of special files. In chapter 4.1 we eluded to the various kinds of file that are created when a user creates an internal or external file. An internal process definition (ipd) file is created when the user submits a script to StreamFS. When a (set of) stream(s) is piped through the defintion file, an internal process instance (ipi) file is created that represents the output of the process. A subscription instance (sub) file is also created. The sub file contains information about which streams are feed the file, a reference to the ipi file, and statistics about the file. If either the sub file or the ipi file are deleted, the process ends. The ipi file is also a stream file. It can be used to pipe that output of the process to another processes or to an external `URL`.

type	**description**	**valid operations**
internal process defintion (ipd)	Javascript process definition.	read, write, delete
internal process instance (ipi)	Management file used for managing active processing of this script.	read, delete
external process definition (epd)	Gives information about where an external process lives.	read, write, delete
external process instance (epi)	An active processing stream to an external process.	read, write, delete
subscription instance (sub)	An instance of a subscription. Contains information about the subscription, such as source/sink and related statistics	read, delete
symbolic link (symlink)	Similar to a symbolic link in Unix.	

Table 4.3: Summary of the 6 special-file sub-types and their valid operations in StreamFS.

The same set of files are created when an external processes is defined and started. When the client stub is started on the client machines it creates an external procrocess definition file for each process that was listed in the configuration file for the stub on the client machine. Similarly, when streams are piped to the defintion files, the client starts the processes on the client and create their associated external process instance (epi) files. Those files are also a sub-type of the stream file and can be used to pipe the output to another process (internal or external) or an external `URL`. Any subscription or pipe that is instantiated creates an associated sub file in the `/sub` directory. These always contain information about the subscription and kill the fowarding process when deleted.

Finally, we used symbolic link (symlinks) the same way they are used in a traditional filesystem. They are also used to generalize the inter-relationship structure in the ERG. They are a crucial file in the support of multi-naming.

Interfaces

We built several interfaces to interact with a StreamFS deployment. Because StreamFS has its own file types and semantics we created a native shell application and a web-application console. The web application console is displayed in Figure 4.11. The console has various features, all mirrored in the shell application. It provides a graphical interface for viewing the files in the deployment, changed the attribute-value pairs associated with file

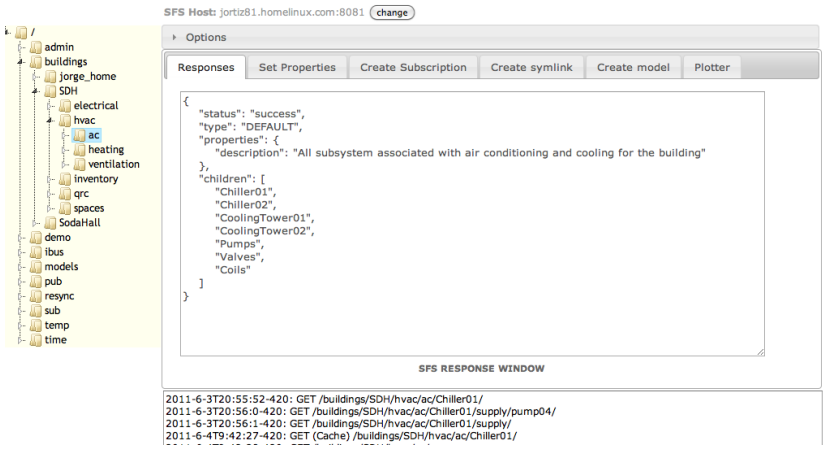


Figure 4.11: StreamFS console. The tool allows the user to view the namespace as a set of files, interact with the system, and view stream data.

Also available to the user is the ability to create and manage symbolic links, submit small processing jobs, pipe streams to external target or processing jobs, and plot. The plotter is a basic mechanism to check if streams are active and for exploration purposes.

4.9 Supporting Multiple Names

One of the goals of the naming scheme in StreamFS was the support multiple names for sensor and actuators. The inclusion of symlinks allows provides this ability. A file can be names and linked to from multiple hierarchies. This allows applications to refer to the same physical entity with a unique object id through multiple names. It is also used by our pub-sub system when the names are resolved and play a crucial role in *dynamic aggregation*.

For example, a temperature sensor may have at least two names that are expose to the end user. It may have a name that refers to it through the context of its spatial placement, such as `/soda/4F/410R/temp` and it may have a name that refers to it through the context of its association with a component in the HVAC system, such as `/soda/hvac/ahu1/vent1/temp`. Either of those are names that should access the same item and that item's associated data.

The underlying stream may actually write the data to a stream file named `/strms/temp` and the other two names are just symlinks to this one.

The pub-sub system uses these names when decided which data streams match a subscription topic. For example, in a typical pub-sub system, if a job wanted to subscribe to the temp stream, it would have to know the name *a priori*, or simply pick a single name. However, we support the notion of multi-topic stream tags. So, if a subscripter request all the streams with the topic `/soda/4F/410R/*` and a data point is written to `/strm/temp`, the subscription manager would list all the aliases for that name, which include `/soda/4F/410R/temp` and see that it matches the topic request for that subscription.

Notice how naming explicitly affects how we group sensors according to some hiearchical organization. Some of those groupings are physical associations with one another that are important to stay accurate. Let's re-examine the example presented in section 2.5. We present the figure from that section again in Figure 4.12.

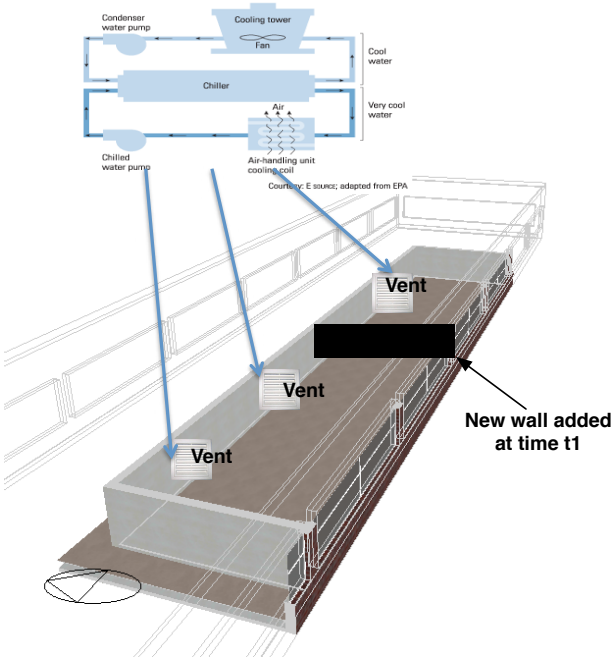


Figure 4.12: MPC example where metadata must be verified to maintain correct behavior.

Recall that the equation that drive the control process depending on knowing the mapping between vents and rooms. Therefore, if the room changes and a vent is addedd or removed, the control algorithm must be updated to account for the change. The naming structure would contain a reference to the vent in some group, linked to a sensor that belongs to a certain room, however, if a wall is put up, we should be able to automatically detect that this has occurred to update the control process. In summary, the naming scheme is used to express different kinds of organizational patterns for the sensors and their data. As the

building goes through some changes and evolves an automatic scheme or family of schemes are necessary to alert the user or process of the chance. In the next section we discuss the mathematical tools we used to verify that the groups specified by the naming convention are correct.

4.10 Related Work

FIAP [63] is a web-services architecture for exposes building-specific communication protocols. It essentially services to map BACnet and/or LonTalk to an HTTP resource architecture. It is not RESTful, however. You must download and use a communication stub in order to obtain resource information and related data. Essentially it tunnels building protocols over HTTP. Their approach to naming is ad-hoc and protocol specific, ours is also ad-hoc but we offer canonical constructions, allow the user to directly access the data through multiple interfaces – including a RESTful one.

sMAP [19] proposes a RESTful architecture for streaming sensor data. Their resource-oriented architecture defines a hiearchical resource structure for a sensor or actuator and provides facilities for reading and writing that data RESTfully. In contrast, they do not take a file system approach and only focus on sensor and actuators. They leave the processing jobs as a function outside of sMAP. Also, by adopting filesystem constructs, we support multi-naming through symlinks – a crucial feature that is missing from either of these.

4.11 Summary

In this chapter we discussed the file abstraction is more detail. We presented the 4 different types of files and how they are used in StreamFS. We summarized their opertations and semantics as well. We also discussed the challenges in dealing with maintaining an accurate view of the metadata over time and presented 4 types of verification. The last one is particularly important because we believe it is fundamental.

In the next chapter we will evaluate the architecture with respect to its system properties. We explore these properties by presenting applications that it support that could not be supported by legacy systems.

We described the details and motivation in the process management and related components. We introduced the notion of internal and external processing. The former is used for small, simple data-cleaning jobs while the latter is for integrating external processing jobs written in the client's native language. We also showed how we comine the entity-relationship graph to provide the infrastructure necessary to support OLAP-style queries. This is an important features, since many of the queries posed in the building domain have the following properties:

1. Temporally-driven, scan-heavy queries.

2. Hierarchical, unit-specific aggregates.

Dynamic aggregation is an efficient design for these kinds of queries. Unlike traditional OLAP, where the timestamps are uniform across other dimensions, we must interpolate the values to keep the "OLAP cube" populated with data at all intervals. It is also necessary to provide accurate aggregates in time.

Finally, we articulate our observation of the importance of scheduling with jobs that want a set of readings that are collectively the latest – the collective buffer freshness is maximized. We formalize the problem and present an algorithm solution and evaluation. In the next chapter we discuss the files and associated semantics in StreamFS. We show can they related to traditional filesystems and discuss the motivation for its design. We also present the mathematical tools for verifying the relationships between sensors that is constructed through the namespace.

Chapter 5

Architectural Evaluation

In this chapter we present an architectural evaluation through our deployment experience. We give an overview of our application programming interface (API) and discuss its use in the context on two applications. We describe how we are able to provide *extensibility, generalizability, scalability,* and *ease-of-management* through a description of the API and the applications that use it.

The first application is the *Mobile Energy Lens*. It is designed to collect building inventory and metering information and provide various live, aggregate view of the energy consumption of devices throughout the buidling. The application makes use of several StreamFS features and provides a working, real-world example one of our target emerging applications. The second application is an mounted Unix filesystem interface for legacy applications. We discuss how StreamFS can support legacy applciations with minimal change to application code. We discuss the mapping from StreamFS file semantics to Unix file semantics. The mounted FS demonstrates the *generalizability* of our architecture.

We also provide a detailed description of each application. We highlight components of the application architecture where StreamFS provides a critical service. For problems which StreamFS only provides a partial solution, we demonstrate how that partial solution is extended into a full solution design and we present an associated evaluation.

5.1 API Overview

In this section we give an overview of the application programming interface. We give a description of some of the main function calls and the corresponding RESTful interface version. We also provide a full tutorial of the HTTP/REST interface in Appendex B.

Table 5.1 gives and overview of API calls that are available through the standard library, available in several languages. It provides functions for creating and deleting files as well as decorating them with attribute-value pairs. The Energy Lens application makes extensive use of this library interface for dealing with consistency-management features and “atomic”

function	description
create	Create a file. Overlaps with several other calls.
delete	Delete a file.
hline register	Join as data source.
push	Write to associated stream file.
label	Add an attribute-value pair.
rlabel	Remove an attribute-value pair.

Table 5.1: Overview of StreamFS file-related API calls. Library written in Java, PHP, and C.

update operations. Most of the applications that runs on the phone, however, uses the HTTP/REST interface.

callback	description
search	General search of terms through file name and content. Similar to grep.
filter	Informs device of removal.
query	Timeseries query.

Table 5.2: Summary of control interface callbacks in StreamFS. Library written in Java, PHP, and C.

The search API is simple (summarized in Table 5.2. It includes three main calls, depending on what you are searching for. If you the application which to make a general search query based on the name of the file, the `search` call allows you to specify a set of terms to look for or to specify specific attribute-value pairs. It also provides a general `path` search option where you can specify that the path match a regular expression. The filter call can be combined with the search call to filter on metadata values returned from the `path` search. Finally, the `query` call runs a timeseries query on a specified path or all stream files that match a regular expression specified by through the `path` option.

Finally, the pub/sub library is used extensively throughout all our applications. It is specifically used in the Energy Lens for initiating the aggregation processing. Table 5.3 summarizes the two main API calls for dealing with subscriptions and pipes.

Internally the mechanism for dealing with either is similar, however, the semantics of the two calls implies that pipes direct streams to user-defined internal/external processing elements, while subscriptions are for external sinks running on the client. Subscriptions are managed through an `HTTP POST` operation. If the user is using a client stub, the `POST` is unmarshalled and a callback is triggered with the body of the `POST` submission.

callback	description
pipe	Pipes the list of specified streams to either an active process or a process definition file.
subscribe	Create an subscription from a set of streams to an external target through a callback function.

Table 5.3: Summary of control interface callbacks in StreamFS. Library written in Java, PHP, and C.

5.2 Energy Auditing With Mobile Phones

Mobile phone penetration continues to rise in the United States and abroad. As such, they serves as convenient interface between people and their environment. The combination of mobile network coverage and indoor wifi connectivity provides near-ubiquitous connectivity at all times. Coupled with the fact that mobile phones are personal devices that can serve as a proxy for the individuals, they provide a valuable point of interaction between the physical world, people, and computation. Through a combination of cloud technology, ubiquitious network access, and mobile phones, we can provide new ways to do in-situ diagnosis of opertional problems in the building, monitor personal energy consumption, and share related information with other occupants. The phone becomes a lens through which we can directly observe the dynamics of our surroundings.

There are several systems challenges that must be overcome in order to achieve this vision. Some can be solved easily through the primitives made available by StreamFS. We examine the challenges in capturing and maintaining a view of the physical world. We track people and things, deal with consistency management issues, and provide mechanisms for dealing with the occasional disconnection. We deploy a number of wireless power meters, use QR codes as a tagging mechanism, and build an android application that interacts directly with StreamFS and plug-load devices. We use mobile phones to construct the entity-relationship graph of the locations, meters, and plug-loads in the building and use the processing features in StreamFS to provide detailed energy-attribution statistics. The 'Mobile Energy Lens' uses QR-code sipe gesture to collect this information. For example, if a computer is inside room Y and connected to meter Z, the user can register the computer, the room, and their "is-in" and "attached-to" relationships explicitly through the a series of gestures that build the associated files and inter-relationships. We use these relationships to guide our analytics. For example, the load curve of room Y consists of the sum of all the power traces for loads inside room Y. Our work examines *three main challenges* in setting up and deploying a tag infrastructure through the building to support this application.

The first challenge is related to tracking. Mobile phones present classical, fundamental challenges related to mobility. Typically, mobility refers to the phone, as the person carrying

it moves from place to place. However, in the energy-attribution context, we refer to the movement of energy-consuming objects. Tracking their relationships to spaces and people is as important as tracking people. We describe how we deal with *both moving people and moving objects* and show that these historically difficult problems can be addressed relatively easily, if the proper infrastructure and services are in place. StreamFS plays an important role in offering the necessary services.

The second challenge is about capturing the inter-relationship semantics and having these inform our analytics. We adopt a general mechanism where physical tags identify objects in the world. Our system uses *QR codes* to tag things and locations in the building. Once tagged, there are three types of swipe interactions – registration, linking, and scanning – which establish important relationships. Registration is the act of creating a virtual object to represent a physical one. Linking captures the relationship between pairs of objects. Scanning is the act of performing an item lookup. Each of these interactions requires a set of swipe gestures. Linking requires two tag swipes while the other two actions require a single tag swipe. StreamFS conveniently already maintains an *entity-relationship graph (ERG)* and we use it extensively in the Energy Lens to record things, people, and locations through these gestures.

The third challenge appears in dealing with indoor network connectivity and access. In order to connect these components, we rely on 'ubiquitous' network connectivity. However, in practice, network *availability* is intermittent and our system must deal with the challenges of intermittency. We discuss how caching and logging are used to address these challenges. Moreover, when connectivity is re-established, we must deal with applying updates to StreamFS; updates recorded by the phone while disconnected. Conflicts may also occur during an update. For example, the two updates may conflict about which items are attached to which meters. We implement a very simple conflict resolution scheme, described in section 5.3. Finally, physical state-transitions are represented as a set of updates to StreamFS that must be applied atomically. However, StreamFS does not provide atomicity across multiple file operations. We implement transactions through log-replay and transaction manager. Our 'Energy Lens' system is deployed in Sutardja Dai Hall at UC Bekreley. We discuss its architecture and design.

5.3 Energy Lens Architecture and System Challenges

The Energy Lens application aims to approximate the vision described in section 2.3. We tag items with QR codes and allow users to 1) tag and register new items, 2) tell us which meters are attached to which items, and 3) scan QR codes to view their load curve over a 24-hour period. Both individual items and spaces are scannable – spaces present aggregate information of the energy-consuming items within them.

The architecture consists of three layers: the sensing and tag layer, the data management and processing layer, and the application layer. In this section we discuss how aspects of each layer address the aforementioned challenges. In deploying the application we run into

various issues that inform our design. For example, *QR code reading times vary substantially across phones and lighting conditions.* You must design for the least-common denominator in terms of camera quality and lighting. Another aspect to consider is network access. Within our building, although connectivity is mostly ubiquitous, network access can be intermittent. Network access may be unavailable for several reasons, including disassociation from an access point due to idleness, dead spots in the building where connectivity to both wifi and 3G/4G are unavailable, multipath-induced destructive interference, and various other reasons. Dealing with these throughout the data collection and update phase is especially troublesome. We discuss mechanisms and algorithms for dealing with disconnected operation.

StreamFS serves as the data management and processing layer. It must be *extensibile* in order to add and remove sensors and contextual information throughout the contextual capture process – the collection of device inventory, descriptive information, and associations between plug-loads and other items and locations in the deployment. It must provide *scalability* with the number of sensors and rooms in the building, since all the sensors produce periodic readings and live, aggregate statistics is the main goal of the application. Also, the application is interactive, so many mobile phone users are constantly performing lookups. StreamFS must handle these requirements.

Challenge 1: Tracking People and Things

The Energy Lens application consists of a web application that displays timeseries data and an Android-based smartphone app. The android app is relatively simple; consisting of a menu with only two options: Update deployment state, scan to view services. Swipe gestures manipualte a local portion of the entity-relationship graph – local with respect to a user's current location. Since each location (room, floor) has a QR code attached to it and items are associated with those locations, we can identify the location by name (`/buildings/SDH/spaces/4F/toaster`). Figure 5.1 goes through the various sets of swipes.

The first set is called a 'registration' swipe and we use it to register new items. The user scans a QR code and the item it is `attached` to. This creates an 'attached-to' link between them. Adding, removing, binding, and attaching items is done with a pair of swipes. A lookup is done with by swiping the QR code attached to an item.

We have designed a set of heuristics for setting the location during an update. It piggybacks on the swipe gesture. The following is a list of rules for automatically setting the location of people and things:

- When a user swipes at a location L, they are presumed to be at L for fixed period τ. An "association timer" is set to release this association after τ seconds.
- If the user swipes anything that is associated with a location l at time $t \leq \tau$, and $l(t) \neq$ L, then we set the new location of the *thing* they swiped to l(t) and reset the association timer.

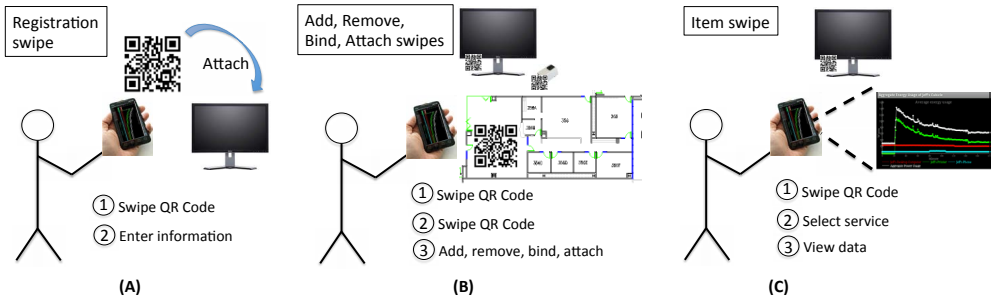


Figure 5.1: Swiping gestures in the mobile application. The registration swipe requires on a single swipe. The linking and registration gestures require two swipes, and the look-up requires a single swipe.

- If the user swipes anything at location l at time $t \geq \tau$, we set the location of the *person* to l(t). We reset the association timer to τ.
- If a user registers a new location, they are presumed to be at that location.

For each of these, we provide an interactive option to ask for location-change confirmation from the user. So if we think the user/item has moved but they have not, the preset action can be overridden. The guiding principal we follow in our design is to leverage the swipe gesture for as much contextual information as possible. Furthermore, we do not explicitly track users. Context is only set on the phone and used in operations sent to the server. We construct an entity relationship graph through naming in StreamFS. StreamFS uses filesystem constructs, such as hierarchical naming and symbolic links, which we use to express an acyclic relationship graph-structure. Because location swipes give us direct confirmation of a user's location, it can be coupled with wifi localization mechanisms and supervised learning algorithms to adjust the localization model as the user interacts with their environment.

Entity-Relationship Graph and Analytics

We construct the entity relationship graph through naming in StreamFS. StreamFS uses filesystem constructs, such as symbolic links and hierarchical naming which are useful for expressing an acyclic graph structure (StreamFS checks for cycles when symlinks are created). The following general path-naming text patterns are used to express different portions of the ERG. `/path/to/device_or_item`, `/path/to/qrc`, `/path/to/space`, `/path/to/taxonomy`, `/path/to/users`. Registered meters are placed in the device path, `/dev`. Items are stored in `/inventory`. QR codes are stored in `/qrc`. When an item is registered a symbolic link is created from the specific qr code directory to the item. `/spaces` contains a hierarchy of floors, rooms, and sub-spaces. `/users` contains the list of usernames. We also have a

/tax directory, where we construct an device hierarchy for access by plug-load category. Placement (location) is also captured with symbolic links.

Each hierarchy not only represents the physical relationships between items but also provides a structure for enabling hierarchical aggregation and querying. We define a load-curve generating process called loadcurve, displayed in Listing 5.1. The full process job can be found in the Appendix, Listing A.1.

Listing 5.1: Partial load curve code used to generate aggregate load curves in the Energy Lens application.

```
function(buffer, state){
    var outObj = new Object();
    var timestamps = new Object();
    outObj.msg = 'processed';
    if(typeof state.slope == 'undefined'){
        state.slope = function(p1, p2){
            if(typeof p1 != 'undefined' && typeof p2 != 'undefined' &&
                typeof p1.value != 'undefined' && typeof p1.ts != 'undefined' &&
                typeof p2.value != 'undefined' && typeof p2.ts != 'undefined'){
                if(p1.ts == p2.ts)
                    return 'inf';
                return (p2.value-p1.value)/(p2.ts-p1.ts);
            }
            return 'error:undefined data point parameter';
        };
        state.intercept = function(slope,p1){
            if(typeof p1 != 'undefined' &&
                typeof p1.value != 'undefined' && typeof p1.ts != 'undefined'){
                return p1.value - (slope*p1.ts);
            }
            return 'error:undefined data point parameter';
        };
    }
}
```

When a new meter is measuring a device in a room, the Energy Lens app actives the loadcurve process through a general subscription instance. For example sub.source(/sdh/4F/473R/*).filter(unit:KW).destination(/sdh/4F/473R/lc_324hb) creates a subscription for streams with the prefix specified in the source call with the filter on the metadata where the units produced on KW and pipes it into an instance of the loadcurve process. Note /sdh/4F/473R/lc_324hb is a symbolic link to /proc/loadcurve/324hb, the instance file the represent the aggregator for this room. The initial instance file can only be created if the source has a match. The app checks if there are any streams in the room every time a new meter is added. If it is the first stream in the room, the pipe and corresponding

symbolic link is created. If all the meters are removed from the room, the process is killed and the symlink becomes a dangling pointer that needs to be cleaned up. The cleanup is done by the Energy Lens application. StreamFS does not provide a clean-up mechanism.

Challenge 2: Consistency Management

We use an eventual-consistency model for maintaining the ERG over time. Naturally, the spatial inter-relationships change over time as items are moved and replaced. In order to deal with this we offer two options: 1) we periodically re-scan the items and their locations, essentially re-capturing the inventory collection portion of the audit or 2) we allow building occupants to participate as auditors, capturing their own personal items and shared items. This should provide at least as much value as a periodic energy audit and can be completed in a fraction of the time [**aceee˙mobileaudit**].

Placement (location) is captured with symbolic links as well. Node types inform the application of the nature of the relationship. We define five distinct types: item, meter, location, category, and tag. The following relationships are constructed with symlinks between different node types:

- **O**wned-by: When a meter/item/location is tagged as belonging to a user.
- **B**ound-to: When a meter is attached to an item and taking physical measurements associated with that device, we say that the meter is "bound-to" the device.
- **A**ttached-to: When a meter/qr code/item is attached to another meter/qr code/item but *not* taking any physical measurements for that item, we say that the meter/item is "attached-to" the other meter/qr code/item. QR codes cannot not be attached to each other.
- **I**s-in: When a meter/item is inside a location, we say that the meter/item "is-in" that location.
- **T**ype-of: When an item is labeled by as a known, specific, type, we say that the item is a "type-of" thing specified by the its label.

All symlinks are interpreted based on these rules of association. As items move, symlinks are removed and re-created in a different folder. We know your current location by following the path from the QR code directory, across a symlink, to a file in the space directory. Items associated with a space have a floor or room folder point to the item via a symlink. This is how we record the location of things throughout the building.

Challenge 3: Disconnected Operation

Although connectivity is ubiquitous, network access is not. This occurs due to dead zones, idle-disconnect and failed hand-off between access points. When encountered in practice,

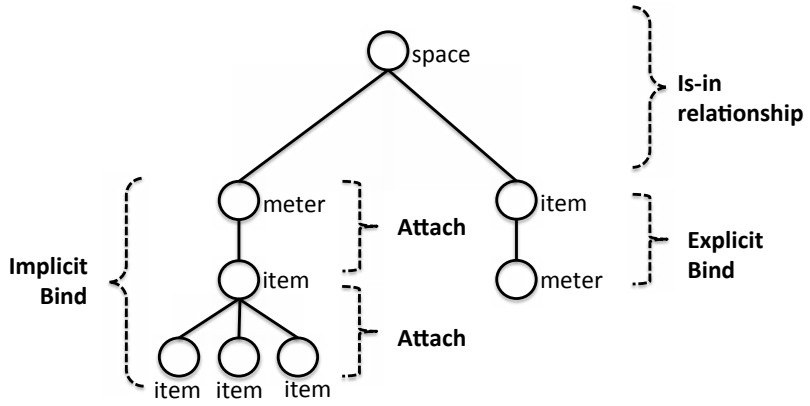


Figure 5.2: This diagram shows the relationship capture between the objects and locations in the building for the energy audit application. Children of a space node have an "is-in" relationship with the space. An item with another item as a child have a "is-attached" relationship and meters attached to items are bound to each other. Note, this is a *subset* of the relationship diagrams generated across our three applications.

especially while editing deployment state, it can be quite frustrating and discourage use of the application. We designed a mechanism that does smart caching, not only to improve performance, but also to allow for disconnected operation.

The Energy Lens application downloads a portion of the ERG when a user enters a new floor. The application fetches the portion of the graph rooted at the floor. Figure **??** shows just a small portion of that entity-relationship graph for the floor we are monitoring in our deployment. A prefetch populates the cache with all entity nodes on that floor, their associated metadata, and 30 minutes of data from the stream entities. The full fetch of the 4th floor data set includes 176 nodes and about 1 MB of meter data. In total, the app downloads 1.2 MB of data upon re-connection. Prefetching allows users to continue interacting with the application as if they were still connected (as long as they remain on the same floor). Without it, the application is not functional until a network connection is established.

The Energy Lens periodically syncs with StreamFS to obtain updates to the ERG and maintain a locally consistent view. Let OP_R be an operation performed on the node rooted at node R and t_i be the current time on the phone. After a set period, the phone sends the server its last performed operation and the time that operation was performed. The server responds with any operations that have take place since then. The client applies those operations internally to the cached version of the the ERG in order to maintain consistency. The "*active*" parameter-check, is for energy savings. If the phone application is active, the check occurs every 10 minutes, otherwise it occurs ever hour.

The process is demonstrated in Figure 5.3. The components shown are the *ERG cache*, the *operation log (OpLog)*, and the *prefetcher*. We separate the steps in the figure as a

```
while 1 do
    if connected and active then
        (1) req ← [t_{i−n}, OP_R];
        (2) resp ← send(req)=[t_{i−k}, OP_R], ..., [t_now, OP_R];
        (3) n >= k;
        for j = 1 → size(resp) do
            (1) op ← resp[j] = [t_{i−k+e}, OP_R];
            (2) apply(op);
        end
    end
    if active then
        (1) Sleep for 10 minutes;
        else
            (1) Sleep for 1 hour
        end
        ;
    end
end
```

Algorithm 2: Prefetch Loop.

READ sequence and a WRITE sequence. All reads go to the cache (steps 1 and 2 on the left hand side of the figure). Writes go through the OpLog (steps 1 - 5 on the right side of the figure). For writes, the application makes a write request (1) and it is forwarded to StreamFS (2). If StreamFS is reachable and the write is successful (3), the operation is applied to the ERG cache (4) and the response is sent the application (5). If the operation is not successfuly, step 4 is skipped. If StreamFS could not be reached, step 3 is skipped, and the operation is written to the OpLog. The OpLog is flushed to the server, by the prefetcher, upon re-connection.

The OpLog contains records of operations that are eventually applied to StreamFS. Some of those operations are actually groups of operations that need to be applied atomically. For example, when a bind or attach operation occurs, we append the timestamp to the item(s) that are being connected, as well as create a link between them in the graph. The application uses both the link and the added metadata to fetch the appropriate graph for display. These operations must be applied atomically or not be applied at all. When the log is dumped, the global transaction manager (GTXM) – the layer that handles log dumps and transaction processing – attempts to apply the log in timestamp order.

Log dumps and conflict resolution

When the Energy Lens application is started it contacts the server and attains the server's local clock time. It notes its local time as being equal to the timestamp of the server and

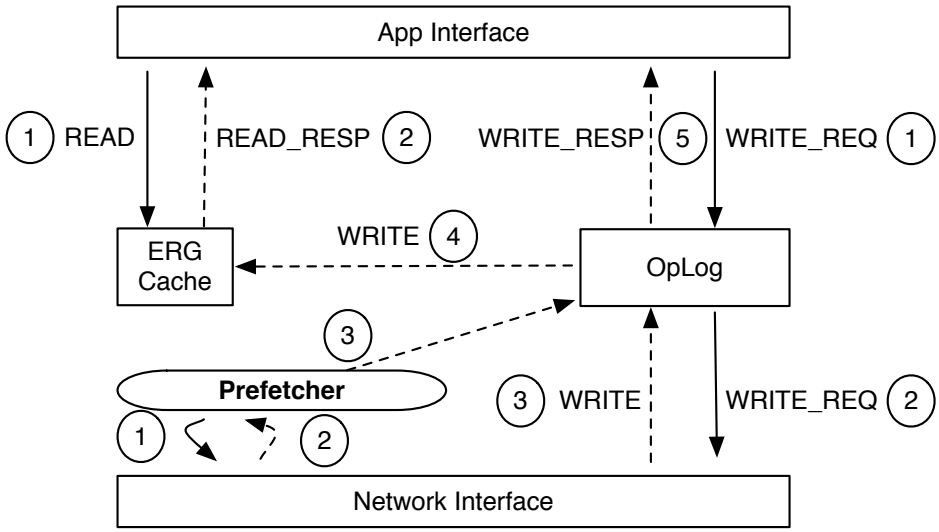


Figure 5.3: Standard mechanisms for consistency management on the phone. All READ request go to the local cached version of the ERG. All WRITES must go through the OpLog. They are eventually applied to the cache if successful and logged if the StreamFS is unreachable. These components are directly built into the Energy Lens application.

calculates all subsequent timestamps using its local clock. When an operation or transaction is added to the OpLog, a timestamp is appended. Let t_s be the timestamp on the server, t_l be the timestamp on the phone when t_s is recorded, and t_{now} be the current time on the phone. Each operation/transaction is timestamped with t_{approx} where:

$$t_{approx} = t_s + (t_{now} - t_l) \tag{5.1}$$

This timestamp is a general approximation of when the operation should be applied on the server. The server applies these is ascending timestamp order.

Generally, a conflict occurs if there is any operation or transaction that was applied after the timestamp of the current operation being processed. A typical transaction manager must rollback the state of the database, apply the operation, and replay the log. However, conflict resolution is much simpler in this context. *The latest operations reflect the state of the world because they are updates induced by direct interaction with the world at that point in time.* Therefore, if there is a conflict between a set of operations, the old ones can all be discarded.

Operations that are discarded are done so silently. We make failures silent for two reasons: 1) There is no way to contact the app when failure occurs. Mobile phones do not often have reversably reachable addresses. 2) Failure assumes the operation was based on a false assumption about the state of the world.

5.4 Energy Lens Experience and Results

Figure 5.4 shows three screen shots of power traces obtained from the ACme deployment, and displayed through the Energy Lens. Notice that there is some data missing in the graph. This occurs because of in the network transmission, reboots on the data management layer, or failed scripts that are automatically restarted. In all cases we get holes in the data. To compute aggregates, interpolation is necessary. StreamFS offers a real-time processing facility, whereby javascript operaters can be applied to streaming data. This allows us to clean it as it comes in and compute the aggregate traces. We are currently experimenting with various aggregation models to give occupants deeper insights.

Deployment Experience

We deployed 20 ACme power meters [**acme**] on a single floor of a building on campus. The data was made available through sMAP [19] and forwarded to our processing and data management layer, StreamFS [**streamfs**]. We distributed the ACmes throughout a single floor in our building and registered various plug loads as being measured by them. We also tagged hundreds of items and locations throughout the entire building. In total we tagged 20 meters, 20 metered items, 351 un-metered items, and 139 rooms over 7 floors. Figure 5.4 shows three screen shots of power traces obtained from the ACme deployment, and displayed through the Energy Lens.

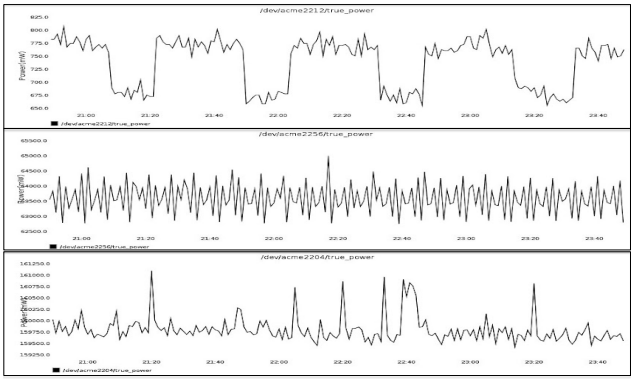


Figure 5.4: Power traces obtained from power meters attached to various plug load on one of the floors of a building on campus. These show screen shots of the Energy Lens timerseries data display.

In our initial deployment we found that the use of our tracking scheme to be effective, especially in conjunction with interactive confirmation. The ERG was effective at capturing deployment state, although highly mobile items, such as laptops, were particularly difficult to keep track of. Finally, our disconnected operation mechanism was effective at masking

intermittent connectivity. For future work we will baseline the floor's energy consumption before and after the deployment and measure if more visibility and analytics indeed motivates occupants to use less.

QR code design

Our choice to use QR codes is important, since the *only way to scale – in deployment size and management complexity – is to involve building occupants.* QR codes are cheap to produce. They can be printed and attached to items with tape or sticky paper. Figure 5.5b shows an example QR code used in our deployment. These are placed on physical objects and spaces throughout the building to link between the physical world and our virtual representation of it.

With the number of physical objects and places in a building, *we must rely on the occupants to scale our deployment and manage it.* Because QR codes are easy to produce, we provide occupants with a webpage that that produces them. They print them out, place them on items or places they want to interact with and use them to register items in StreamFS through the Energy Lens. Since occupants are our target users, we must make the application easy to use. For QR code scan-times, vary by lighting conditions, camera quality, and hand movements. In poor conditions scanning is cumbersome; ultimately demotivating continued use. QR codes must be designed to minimize scanning time. In our initial deployment and experiments we observe that complex QR code code not only have an longer average scanning time but also experience a larger variance in scanning time. The more complex the pixel design in the QR code, the harder it is for the camera to focus and capture it.

We scanned each QR code shown in Figure 5.5, under light and dark lighting conditions. Each experiment was run 10 times and Table 5.4 shows a statistical summary. Scanning the simple QR code under well-lit conditions performed the best. The complex QR code under the same condition takes about 28-36% longer to scan. Perhaps even more important is the variance. Notice that the variance with the simple QR code is smaller. QR code image complexity increases with the amount of information you encode on it. Therefore, it was important to decrease the amount of information we encoded, *placing the complexity in the lookup rather than the tag.*

Table 5.4 shows the results of some simple scanning experiment between the two tags shown above. We scanned each QR code under light and dark lighting conditions, off the screen of my laptop. Each experiment was run 10 times and the table shows the statistical overview of the results. Clearly, scanning the simple QR code under well-lit conditions performed the best. The complex QR code under the same condition takes about 28-36% longer to scan. On a generic QR code scanner, as used here, there is a portion of the scan time that is independent of the code complexity. As these are more heavily used, this is expected to be reduced substantially and the difference is acquisition complexity will be even more pronounced. Perhaps even more important is the variance. Notice that the variance with the simple QR code is much smaller and more stable under either condition. In our experience,

(a) Long QR Code (b) Short QR Code.

Figure 5.5: 5.5a resolves to the same `URL` as the 5.5b, after resolution and redirection is complete. The short label resolves to `http://tinyurl.com/6235eyw`. 5.5b encodes about half the characters as the 5.5a. We used tinyUrl to reduce the QR code image complexity and scan time.

	Average (sec)	**V**ariance (sec)
Short,light	1.66	0.33
Short, dark	2.08	0.35
Long, light	2.26	0.71
Long, dark	2.82	0.50

Table 5.4: Shows the time to scan a long QR code versus a short QR code in light and dark conditions (loosely defined). Notice that short QR codes scan faster and with less variance that long ones.

large variance in scan time is a major problem for complex QR codes. Thus we decided to re-design our codes and push more information in the lookup processes, as network access was more reliable than the focus of the camera on various mobile devices. Tags are placed on all types of devices in all kinds of locations with varying degrees of lighting. Simple QR codes are vital for widespread use.

The design choice forced us to examine others that were related. Not being able to encode much information on our QR codes means we are more reliant on the network to provide the bulk of the information, to be very reliable, and to be widespread enough that disconnection is not problematic. Moreover, there are a number of clients that can be used to access and display the information and the tag has to be meaningful for both. In order to meet these criteria we (1) shrunk `URL`'s using tinyURL [**tinyurl**] as a level of indirection and (2) designed two classes of applications: *shallow* applications, and *deep-inspection* applications. Shallow applications interact with the web-application directly while deep-inspection application use the `URL` of the web application to extract a unique identifier and provide deeper inspection and update capabilities of the entity-relationship graph.

This is an example `URL` we used in our deployment: `http://tinyurl.com/6235eyw`. When resolved, we get an empty response in the body, but we use the header to identify

the QR code identifier that we associate with the item. The response header is down in Figure 5.6.

```
HTTP/1.1 301 Moved Permanently
X-Powered-By: PHP/5.3.8
Set-Cookie: tinyUUID=ee81f56c2c15850975b7d175;
expires=Thu, 13-Dec-2012 04:00:18 GMT; path=/; domain=.tinyurl.com
Location: http://streamfs.cs.berkeley.edu/mobile/
mobile.php?qrc=4eb460a39fcd7
X-tiny: db 0.015100002288818
Content-type: text/html
Content-Length: 0
Connection: close
Date: Wed, 14 Dec 2011 04:00:18 GMT
Server: TinyURL/1.6
```

Figure 5.6: The header of the response from the `tinyUrl` when resolving a QR code. The 'Location' attribute is used to extract the unique identifier for the object this QR code tags. It is also used to re-direct users without the phone application to a meaningful web address for the object.

Notice the 'Location' attribute in the header. This is the location of the re-direct. This approach gives us flexibility in several ways:

1. It allows us to encode less information in the QR code, decreasing its visual complexity; making it more robust across phones with different camera quality, poor lighting conditions, and shaky hands.

2. It allows the added layer of indirection to serve two versions of the applications: the native application, where users can *deeply explore* and edit the entities and their relationships and the *shallow lookup*, which re-directs the mobile phone to a read-only view of the item that was scanned – such as a power trace or a description.

It provides a web address for users to re-direct to and find information and various read-only services for the object. However, because the `URL` also contains a unique identifier *qrc*, it can be used to provide for sophisticated services and capabilities. An example is the ability to change the virtual structure of inter-relationship between this object and other objects. This is demonstrated in our energy auditing application discussed in detail in section **??**. Once items are tagged, they can be added and removed by swiping the tag and pressing the button for what you want to do with the item. You also check into locations either explicit with a location-tag swipe or implicitly with an item swipe.

Shallow applications use the `URL` directly. The *qrc* `URL` is unqiue identifier for the item that this tag is attached to. A shallow application can obtain mostly read-only service through our web applications. For example, we'll see how to get either item-specific data or item-aggregated data with respect to the user making the request (i.e. the total energy consumed by *my* devices). *Deep-inspection* applications are native to the phone, so we can do much more with the tag. Our energy auditing application allows you to related the item to other items by maintaining state of swipe history. This is more difficult with the

web-applicaiton. We can also use the tag and item information to couple it with sensor information coming from sensors on the phone itself. For example, we could determine the direction an object is pointing by using the phone's directional sensor and negating their direction (i.e. phone is facing east, tag on item must be facing west).

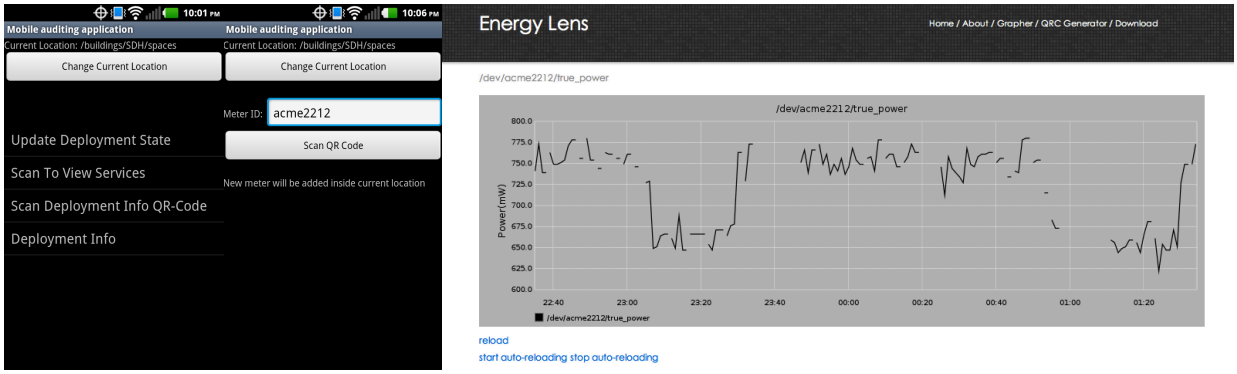


Figure 5.7: Screen shots of the mobile application. The screens on the left are for editing the state of the deployment. The graph on the right shows a live feed of a the sensor that's attached to the item that was scanned with the 'Scan To View Services' option in the mobile application. It can also be resolved by scanning the QR code and following the re-direct to the URL.

ACme meters are IP-enabled and forward their data through a router that runs sMAP. sMAP then forwards the incoming data to StreamFS, running in a machine in Amazon's EC2. StreamFS is a web service that organizes streaming data and metadata using a hierarchical naming convention. It also provide a pub/sub facility for streaming data. We construct a canonical naming convention within StreamFS to express the entity relationships between people, things, and meters. The pub/sub mechanism allows us to combine the ERG with streaming data, and feeds our timeseries data viewers.

EnergyLens Evaluation

In this section we measure prefetch download times and discuss strategies for providing scalability. We also look at the transaction manager and discuss conflict resolution.

Sensing and tag layer

We deployed 20 ACme power meters [44] on a single floor of a building on campus. The data was made available through sMAP [19] and forwarded to our processing and data management layer, StreamFS. We distributed the ACmes throughout a single floor in our building and registered various plug loads as being measured by them. We also tagged

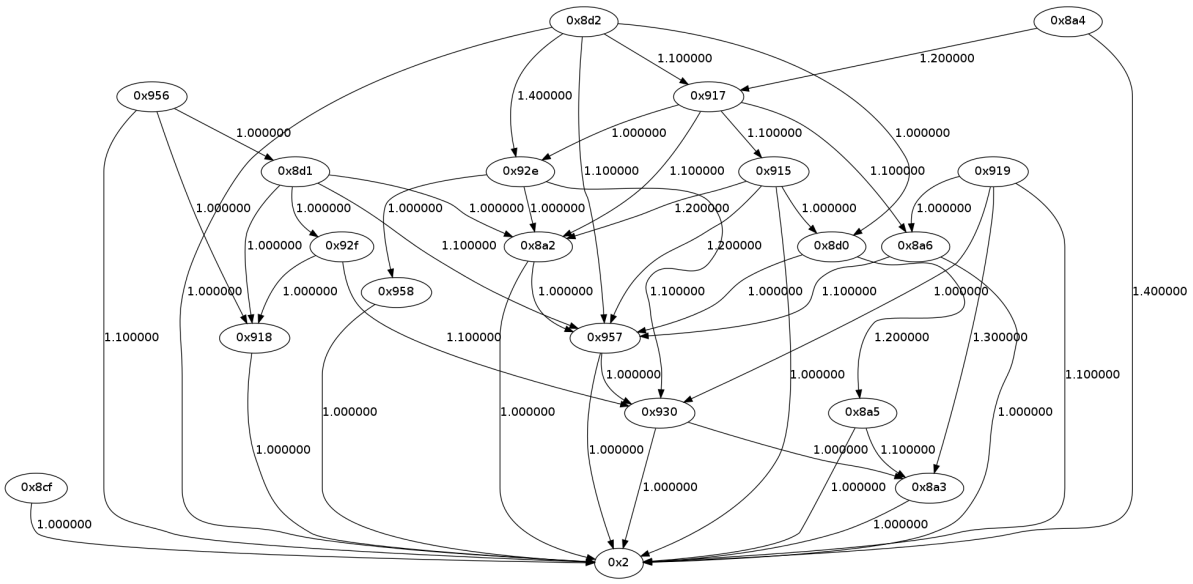


Figure 5.8:

hundreds of items and locations throughout the entire building. In total we tagged 20 meters, 20 metered items, 351 un-metered items, and 139 rooms over 7 floors.

StreamFS supports the addition and operations to record all the deployment information. It also supports active aggregates for 31 rooms on the 4th floor of the deployment and 1 for the entire building. The number of active users was no more than 10 at any time. Because these number are not that large they do not truly test the scalability of the system. However, they do demonstrate both *extensibility* and *generalizability*. Without the first property we could not have added the the sensors, related information, user context, spatial configuration information and updated it with ease. Without the second property we could not have supported both the collection of metadata and data about the deployment as well as *data services* that are absolutely crucial to the success of the Energy Lens application.

Also, we used a single function and instantiated 32 instances of it in the deployment. This demonstrate the *ease of management* property we set to provide through StreamFS. Because we follow the principle where "everything is a file" the entire deployment data, metadata, and analytics can be accessed and controlled from the same deployment.

Prefetching

Prefetching occurs when a user enters a new floor, as detected by a floor scan or an item scan. Table 5.5 shows that the prefetch times scale linearly with the number of items (and data) to prefetch. Each node holds approximate 100 bytes or information and for a 20-node deployment of power meters, producing 100 bytes of data per stream (three streams per

No. nodes	Fetch time (sec)	Std. Error (sec)
1	0.8902	0.0756
10	5.7342	1.7087
100	52.3145	14.1146

Table 5.5: Shows the time to fetch nodes based on the size of the fetch. The fetch time increased linearly with the number of nodes. Caching maintain fetch time near that of fetching a single node. A callback is used when cache is invalidated.

ACme) every 20 seconds, we fetch approximately 1 MB of data.

These prefetch times are non-trivial to deal with, especially since they cause the phone application to slow down until the data is received and loaded into the local cache. The overhead is dominated by the query in StreamFS that constructs the entire sub graph to send to the application. For future work, we are will implement a callback facility and pass the application a reference to it. The app can then periodically check back until the query completes and the data is ready to be downloaded. We can also include partial responses to the query in the prefetch-loop response. This also allows users to continue using the application without any frustrating waits.

Log dump measurements

Table 5.6 shows the operations that the transaction manager calls on the StreamFS server. Log replay and transaction processing is entirely dependent on the time to execute these operations on StreamFS. There are five types of transactions, a *move*, a *un/bind*, *un/attach*. A move is a combination of a 'delete' and a 'create link', a bind is a 'create link' and an 'update tags', an unbind and unattach is a 'delete' and 'update tag'. The transaction latency is the sum of these operations. By far the most expensive operation is a 'create node' operation. This occurs when a user adds a new item/space/person to the graph. The time to apply the operation also scales linearly with the size of the logs.

All logs dumps are processed sequentially. However, for future work we look to parallelize processing into parallel processes updating different portions of the graph. For example, log updates rooted at different floors could occur simultaneously.

The global transaction manager implements three main high-level operations – rollback, apply, and replay. Our current implementation is limited by the time is takes to perform an operation on StreamFS. Table 5.6 shows the three main operation that the transaction manager needs to do on the server.

Usually rollbacks consist of *delete* operation and applies consist of *creates*. In a worst-cases analysis of performance we expect the total conflict resolution time to be roughly bounded by $rollback\_time + apply\_time + replay\_time$, since $replay\_time = 3X rollback\_time$ and apply_time is negligible, the total time is approximately $4X rollback\_time$. Therefore the overhead is driven by how many new links were created that have to be deleted and then

Operation	**A**vg. exec. time (ms)
fetch	250
delete	326
update tags	267
create link	250
create node	1036

Table 5.6: Average operation execution time in StreamFS.

re-created. As an optimization, we limit the scope of a rollback. The naive approach is to blindly undo all operations up to a certain time. However, we can use the location of the node in the ERG to limit the conflict-scope to a sub graph, rather than the entire graph. The simplest approach is to check if the operation on a node either shares an immediate parent with the node that will have an operation undone on it or it the operation is on the same node. By limiting conflict-scope we minimize the number of operation that get execute and, hence, minimize the cost of resolution.

5.5 Mounted Filesystem and Matlab Integration

Although StreamFS files and semantics are not POSIX compliant, we wrote a FUSE [**fuse**] imeplementation that allows legacy applications to directly interact with StreamFS file and operate on them as if they were locally mounted.

```
system('echo "ts_timestamp=gt:now-10000" > /Users/jortiz81/StreamFS/streamfs-fuse/mac_osx/tmp/homes/jorge/acmes/2218/power');
system('cp /Users/jortiz81/StreamFS/streamfs-fuse/mac_osx/tmp/homes/jorge/acmes/2218/power /Users/jortiz81/StreamFS/streamfs-fuse/power_2218_10000s.dat');
power_2218 = importdata('/Users/jortiz81/StreamFS/streamfs-fuse/power_2218_10000s.dat');
```

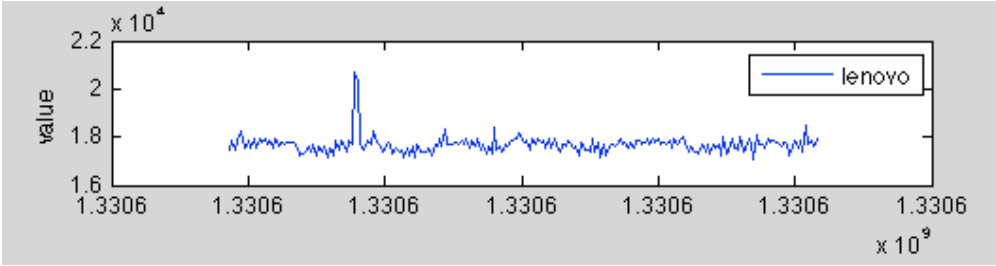


Figure 5.9: SFSFuse implementation. By mapping access and operational semantics to POSIX file operations we enable legacy, desktop application to interact with the deployment directly.

Figure 5.9 shows a screen shot of mlatba reading and writing to local files that communicate with the StreamFS server. This implementation represents an exercise and proof-point that the file abstract can serve as a conveninient interface representing building deployments. Although not an exact fit, the semantics of StreamFS can be translated to make them look like a regular unix-style file. Moreover, the security semantics are the same, since those are adopted directly.

operations	**description**
stream file read	Read the data from a timeseries query.
stream file write	Only accept query parameters.
hline container read	Folder read semantics.
container write	Folder write semantics.
tail -f	starts a stream subscription, writes to file
pipe	writes file contents to pipe

Table 5.7: Overview of StreamFS file-related API calls. Library written in Java, PHP, and C.

Table 5.7 gives an overview of the operations that we translate from StreamFS to a standard Unix file. Note that we did not translate all of the operations. These were just a convenient set of them for client-based applications that interact with StreamFS through a filesystem mount. Part of the verification work is implemented entirely through the StreamFS-FUSE implementation. The work on functional verification, discussed in Section 6.4 uses a mounted FS version of StreamFS to fetch chunks of the data for bulk, client-side analysis of the streams.

This interface made interaction with a large amount of data very simple. If the operation are mainly fetch operations, the implementation is simple enough that the semantics translate cleanly. Moreover, the suite of tools, like grep and pipe, translate directly. We can leverage the family of tools and security mechanisms. This also demonstrates the extensibility and generalizability of the system. This layer can offer a fraction of useful menchanisms that StreamFS makes available and can leverage a large number of legacy analytical applications.

5.6 Related Work

Our work touches on several areas from smart home research to logistics. In the building space, there has been some interest in building various kinds of energy-related visualization and control applications. This work focuses on the object definition, tracking, and management component of the architecture proposed by Hsu et al. [**hbci**]. Their work stratified the set of challenges that one could potentially face if the application were deployed at scale. Our work, in contrast, bases its design rationale on a *real deployment* that is taking place at scale in a building on our campus. We focus on solving fundamental systems challenges in

dealing with intermittent connectivity and conflict resolution in tracking people and things over time. We also focus on leveraging gestures to minimize the cost of interaction for users, while maximizing the information we can attain about the state of the world. smart home research [**cooltown**] to

Our work touches on several areas from logistics [**rfid˙gonz2006**] to context-aware mobile applications [**ACE**]. In the building space, there has been some interest in building various kinds of energy-related visualization and control applications. HBCI [**hbci**] proposes a high level archiecture that also relies on QR codes, mobile phones, and ubiquitous network access. HBCI introduces the notion of object capture through the mobile phone and individual services provides by the object, accessible via an object lookup. The proposed service model is *object-centric*, such as individual power traces or direct control accesss. Their "query" service is a tag lookup mechanism realized through QR code scanning of items. The 'Energy Lens' also embodies the "query" via a tag lookup, however we focus on context-related services rather than object-centric services. We build and maintain an entity-relationship graph (ERG) to capture the inter-relationships between items. The ERG informs our analytical processing. We use an eventual-consistency model to maintain the inter-relationship graph over time. HBCI does not address the challenges faced in realizing an indoor, interactive application that relies on ubiquitous network connectivity. Our architecture directly addresses this challenge, as we observe that indoor connectivity characteristics do not comply with the ubiquitous connectivity requirement for this class of application.

An important aspect of the Energy Lens is determining when people and things have moved. This requires some form of indoor localization. There's a large body of literature in the area of indoor localization with mobile phones ranging from using wifi [**radar**], to sonar [**cricket**], to ambient noise [**abs**], and a combination of sensors on the phone [**surroundsense**]. Dita [**dita**] uses acoustic localization of mobile phones and also uses the infrastructure to determine gestures in free-space that are classified into pre-defined control actions. Each of these require relatively complex software and/or infrastrure. We take a radically different, simple approach. We use cheap, easy to re/produce tags (QR codes), incrementally place them on things in the environment and use the natural *swiping gesture* that users make, when interacting with the Energy Lens application, to track when they have moved or when the objects around them have moved. The guiding principal is to attain as much information from a gesture as possible, to determine a move has occurred. We discuss our heuristics in section 5.3.

ACE [**ACE**] uses various sensors on the phone to infer a user's context. The context domain consists of a set of user activities and semantic locations. For example, ACE distinguishes between *Running, Driving, AtHome, or InOffice*. ACE can also infer one from the other. If a user is *AtHome* then they are not *InOffice*. Energy Lens uses inference to determine when a person or thing has moved. Certain swipe combinations give us information about whether they moved or whether an item moved. The main difference is that we only infer context when a user is actively swiping, rather than a continuous approach.

Prefetching is a fundamental technique used in many domains. The cost of a prefetch for mobile application out-weighs the benefits if the prefetched data is not useful. Informed mo-

bile prefetching [**imp˙mobisys2012**] uses cost-benefit analysis to determine when to prefetch content for the user. We prefetch data based on location swipes. Caching prefetched content not only improves performance and interactivity, but it is *necessary* to sustain normal operation during periods of disconnectedness.

Logistic systems focus on tracking objects as the move from distribution sites, to warehouses, and across purchase counters. Items are tracked through their unique bar codes, often embedded in an RFID. The workload in logistic systems is very structured and well defined. The authors of [**rfid˙gonz2006**] describe this structure and leverage it to minimize storage requirements and optimize query-processing performance. The Energy Lens uses QR codes as tags and the phone as an active reader. As objects move, users scan those items to their new location. However, objects may belong to one or many people, they can be metered by multiple meters a day, and their history in the system is on-going. In contrast, a typical logistics workload has a start (distribution site) and end point (leaving the store after a sale). Our workload is less well-defined. Furthermore, relationship semantics are important; For example, we interpret a bind relationship differently from an attach relationship. We discuss the difference later in the paper.

Furthermore, we take advantage of natural gestures the user makes with the phone while scanning QR codes to extract information about the current location of the user or things.

The key idea in the HP Cooltown [**bridgingphysical**, **cooltown**] work is to web-enable 'things' in the world, grouped-by 'place', and accessed by 'people' via a standardized acquisition protocol (HTTP) and format (HTML, XML). Cooltown creates a web presence for things in the world either directly (embedded web server) or indirectly (URL-lookup tag) as a web page page that display the services it provides. Many of the core concepts in Cooltown also show up in Energy Lens. The main overlap is the use of tags in the world that contain a reference to a virtual resource, accessible via HTTP through a network connection. Cooltown, however, explicitly chooses not maintain a centralized relationship graph, it leverages the decentralized, linking structure of the web to group associated web pages together. Furthermore, things are assumed to not move. People are the main mobile entities. The kind of applications we wish to support must track where things are and their specific interrelationships. We imposed a richer set of semantics on our, centrally maintained, relationship graph and use it to provide detailed energy information.

5.7 Summary

We demonstrate that StreamFS can serve a diverse set of applications from a single locations, provide *generalizability* and *ease of management* through the file system abstraction and data-processing "pipe" abstraction. In this Energy Lens application we examined two system challenges – mobility and consistency management – for enabling and energy analytics applications in buildings. StreamFS plays a crucial role in providing the kinds of services necessary for collecting and managing deployment data and metadata as well as providing live aggregate statistics on the deployment to the end-user. Furthermore, we demonstrated

No. deployments	7
Total files	> 10k
Locations	University of Tokyo, UCB Cory, SDH, Stanford Y2E2, Intel, Nokia, Samsung
Total data	1TB over 2 years

Table 5.8: This table summarizes the deployment statistics of for StreamFS over a two years.

the *extensibility* of the system, as all the StreamFS features were agile enough to deal with the evolving dynamics of the deployment and the services built to support fundamental challenges with consistency management and disconnected operation.

Altogether, StreamFS was deployed across 7 buildings in 2 countries, with very different climates, systems, and usage patterns. Table 5.8 summarizes these.

Our largest deployment had > 7k feeds simultaneously feeds a single deployment. Several hundred derivative streams were generated in these deployments as well. Several of these required a cluster of processing elements and datastore elements. Typically, no larger than 2 machines per component. This demonstrates the *scalability* of StreamFS in practice.

Chapter 6

Empirical Verification of System Functionality and Metadata

6.1 Verification through Sensor Data

Every system that manages data in the building specifies the building model manually at some point in the lifecycle of the metadata. Moreover, as the building changes and physical configuration of the building and deployment change, all changes in the virtual representation of the building are updated manually. As we move more and more towards software controlled spaces and system that rely on accurate state capture of the physical environment, it becomes of highest important to automate the process that verifies correct configuration specification.

In the architecture we propose we see it as a concurrent process that will be in parallel with all active deployments. The hierarchical naming scheme explicitly informs the verification process of "group-by" relationships between sensors. Such "group-by" relationships typically specify physical relationships, such as grouping sensors that are in the same room in the same group, or the same floor in the same group. Ideally, a verification process could perform several clustering algorithms, using the empirical data collected from the sensors, to infer such physical relationships. Most importantly, it can alert the user when previously statistically clustered sensors no longer show the same *inter-relationship*.

In this chapter, we discuss the analytical underpinning for a verification mechanism that could eventually become a component in the StreamFS – or any other – building software system. We introduce the various kinds of verification in depth, discuss the mathematical tools that help us both formulate and solve related verification problems, and present results for the empirical verification of various physical relationships between sensors. In the next section we discuss the various types of verification, then we describe the tools, methodologies, and results for solving each.

6.2 Types of Verification

There are 3 kinds of verification that we discuss in this section.

1. *Functional*: verifying whether the behavior of the components in changing.
2. *Spatial*: verifying the spatial relationship between components.
3. *Categorical*: verifying the unit of measure and phenomenon being observed.

Before we discuss the details and approach for each, let us first examine the mathematical tools that we use in our solutions. Then we discuss each type of verification in more detail. We give a detailed description of the methodology for each and present the results of our analysis.

Correlation

Throughout this work, we make extensive use of the correlation coefficient function defined as:

$$r(X,Y) = r_{X,Y} = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}}$$

where X, Y are separate sets of values, n is the total number of sample points in each set, and $\overline{X}$ is the mean value of X (same for $\overline{Y}$ and Y). For each pair of sensors, we compute the corrcoeff to ascertain the relationship between them.

Empirical Mode Decomposition

Another important tool that we used is empirical mode decomposition. We use it partition the empiriical data is its constituent sub-signals and examine what those sub-signals tell us about its behavior. We also use it to examine how signal behavve between one another at certain bands of importance that contain important physical information.

Empirical Mode Decomposition (EMD) [41] is a technique that decomposes a signal and reveals intrinsic patterns, trends, and noise. This technique has been widely applied to a variety of datasets, including climate variables [**lee:climateEMD2011**], medical data [**blanco:bioMed2008**], speech signals [**huang:signalProc2006**, **hasan:ieeeletter2009**], and image processing [**nunes:vision2005**]. EMD's effectiveness relies on its empirical, adaptive and intuitive approach. In fact, this technique is designed to efficiently decompose both non-stationary and non-linear signals without requiring any a priori basis functions or tuning.

EMD decomposes a signal into a set of oscillatory components called intrinsic mode functions (IMFs). An IMF satisfies two conditions: (1) it contains the same number of extrema and zero crossings (or differ at most by one); (2) the two IMF envelopes defined by

its local maxima and local minima are symmetric with respect to zero. Consequently, IMFs are functions that directly convey the amplitude and frequency modulations.

EMD is an iterative algorithm that extracts IMFs step by step by using the so-called sifting process [41]; each step seeks for the IMF with the highest frequency by sifting, then the computed IMF is removed from the data and the residual data are used as input for the next step. The process stops when the residual data becomes a monotonic function from which no more IMF can be extracted.

We formally describe the EMD algorithm as follows:

1. Sifting process: For a current signal $h_0 = X$, let m_0 be the mean of its upper and lower envelopes as determined from a cubic-spline interpolation of local maxima and minima.

2. The estimated local mean m_0 is removed from the signal, giving a first component: $h_1 = h_0 - m_0$

3. The sifting process is iterated, h_1 taking the place of h_0. Using its upper and lower envelopes, a new local mean m_1 is computed and $h_2 = h_1 - m_1$.

4. The procedure is repeated k times until $h_k = h_{k-1} - m_{k-1}$ is an IMF according to the two conditions above.

5. This first IMF is designated as $c_1 = h_k$, and contains the component with shortest periods. We extract it from the signal to produce a residual: $r_1 = X - c_1$. Steps 1 to 4 are repeated on the residual signal r_1, providing IMFs c_j and residuals $r_j = r_{j-1} - c_j$, for j from 1 to n.

6. The process stops when residual r_n contains no more than 3 extrema.

The result of EMD is a set of IMFs c_i and the final residue r_n, such as:

$$X = \sum_{i=1}^{n} c_i + r_n$$

where the size of the resulting set of IMFs (n) depends on the original signal X and r_n represents the trend of the data (see *IMFs* in Figure 6.5).

For this work we implemented a variant of EMD called Complete Ensemble EMD [**torres:icassp2012**]. This algorithm computes EMD several times with additional noise, it allows us to efficiently analyze signals that have flat sections (i.e. consuming no electricity in our case).

Reaggregation of Intrinsic Mode Functions

By applying EMD to energy consumption signals we obtain a set of IMFs that precisely describe the devices consumption patterns at different frequency bands. Therefore, we can

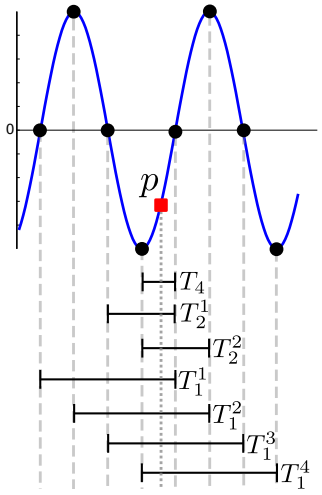


Figure 6.1: Generalized Zero Crossing: the local mean period at the point p is computed from one quarter period T_4, two half periods T_2^x and four full periods T_1^y (where $x = 1, 2$, and, $y = 1, 2, 3, 4$).

focus our analysis on the smaller time scales, ignoring the dominant patterns that prevent us from effectively analyzing raw signals.

However, comparing the IMFs obtained from different signals is also not trivial, because EMD is empirically uncovering IMFs from the data there is no guarantee that the two IMFs c_i^1 and c_i^2 obtained from two distinct signals S^1 and S^2 represent data at the same frequency domain. Directly comparing c_i^1 and c_i^2 is meaningless unless we confirm that they belong to the same frequency domain.

There are numerous techniques to retrieve IMF frequencies [**huang:aada2009**]. In this work we take advantage of the Generalized Zero Crossing (GZC) [**huang:patent2006**] because it is a simple and robust estimator of the instantaneous IMF frequency [**huang:aada2009**]. GZC is a direct estimation of IMF instantaneous frequency using critical points defined as the zero crossings and local extrema (round dots in Figure 6.1). Formally, given a data point p, GZC measures the quarter (T_4), the two halves (T_2^x), and the four full periods (T_1^y), p belong to (see Figure 6.1) and the instantaneous period is computed as:

$$T = \frac{1}{7}\{4T_4 + (2T_2^1 + 2T_2^2) + (T_1^1 + T_1^2 + T_1^3 + T_1^4)\}$$

Since all points p between two critical points have the same instantaneous period GZC is local down to a quarter period. Hereafter, we refer to the time scale of an IMF as the average of the instantaneous periods along the whole IMF. Because the time scale of each IMF depends on the original signal, we propose the following to efficiently compare IMFs from different signals. We cluster IMFs with respect to their time scales and partially reconstruct

each signal by aggregating its IMFs from the same cluster. Then, we directly compare the partial signals of different devices.

EMD yields distinct components in different time scales and we compute the instantaneous frequencies [40] of IMFs using Generalized Zero-Crossing [39]. We break the time scales into four frequency bands:

- High Frequency: a time scale smaller than 30 minutes, mainly reflecting the operation characteristics of devices and noise in system.
- Medium Frequency: a time scale between 30 minutes and 6 hours, which is within the time span of daily activities inside a building.
- Low Frequency: a time scale between 6 hours and 7 days.
- Residue: everything has a time scale longer than 7 days and shows long-term patterns, such as seasonal changes.

Later in this thesis we will explain how we use the medium-frequency band for both spatial verification and functional verification. We discuss the 4 types of verification in the next section.

Functional Verification

With an increased push for operational efficiency, operators are relying more on historical data processing to uncover opportunities for energy-savings. However, they are overwhelmed with the deluge of data and seek more efficient ways to identify potential problems. In this thesis, we present an approach called the Strip, Bind and Search (SBS); a method for uncovering abnormal equipment behavior and in-concert usage patterns. SBS uncovers relationships between devices and constructs a model for their usage pattern relative to other devices. It then flags deviations from the model.

The intuition behind the proposed approach is that each service provided by the building requires a minimum subset of devices. The devices within a subset are used at the same time when the corresponding service is needed and a savings opportunity is characterized by the partial activation of the devices. For example, office comfort is attained through sufficient lighting, ventilation, and air conditioning. These are controlled by the lighting and HVAC (Heating, Ventilation, and Air Conditioning) system. Thus, when the room is occupied both the air conditioner (heater on a cold day) and lights are used together and should be turned off when the room is empty. In principal, if a person leaves the room and turns off *only* the lights then the air conditioner (or heater) is a source of electricity waste.

Following this basic idea we propose *Strip, Bind and Search* (SBS), an unsupervised methodology that systematically detects electricity waste. Our proposal consists of two key components:

Strip and Bind The first part of the proposed method mines the raw sensor data, identifying inter-device usage patterns. We first *strip* the underlying traces of occupancy-induced trends. Then we *bind* devices whose underlying behavior is highly correlated. This allows us to differentiate between devices that are used together (high correlation), used independently (no correlation), and used mutually exclusively (negative correlation).

Search The second part of the method monitors devices relationships over time and reports deviations from the norm. It learns the normal inter-device usage using a robust, longitudinal analysis of the building data and detect anomalous usages. Such abnormalities usually present an opportunity to reduce electricity waste or events that deserve careful attention (e.g. faulty device).

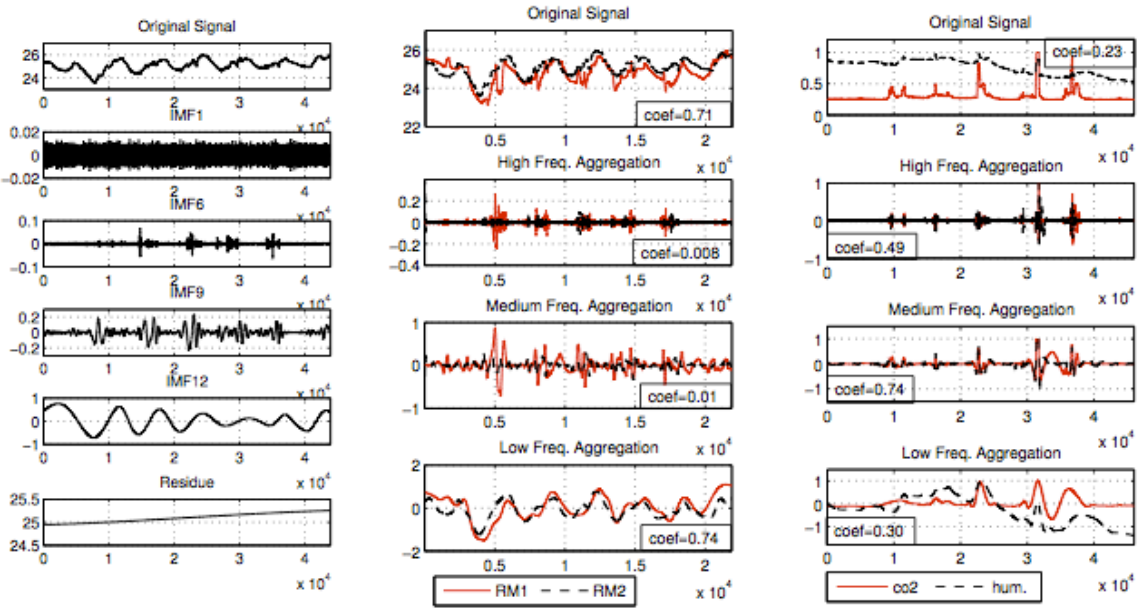
SBS overcomes several challenges. First, noisy sensor traces that all share a similar trend, making direct correlation analysis non-trivial. Device energy consumption is mainly driven by occupancy and weather, all the devices display a similar daily pattern, in roughly overlapping time intervals and phases. Therefore, one of the main contributions of this work is uncovering the intrinsic device relationships by filtering out the dominant trend. For this task we use Empirical Mode Decomposition [41], a known method for de-trending time-varying signals.

Another key contribution of this work is in using SBS to practically monitor building energy consumption. Moreover, the proposed method is easy to use and functions in any building, as it does not require prior knowledge of the building nor extra sensors. It is also tuned through a single intuitive parameter.

Spatial Verification

Typically, placement information is embedded in the name or associated metadata for each sensor in the building. These are used to group sensors by location. For example, in our building data, all sensors that contain the string '410' in their name are in room 410. Processes typically group streams in this fashion: using regular-expression matching or field-matching queries on the characters in the sensor name or metadata. If these are not updated to reflect changes then such group-by query results will not accurately represent true spatial relationships. We observe that spatial associations can be derived empirically. We start with this approach in our work and explore, more deeply, the extent to which it can be used as a verification tool for corroborating the groups constructed from character-matching queries. We refer to this process as *spatial verification*.

We start our analysis by extending the methodology used for functional verification, based on empirical mode decomposition (EMD). In our analysis, we collect traces from several sensors and run EMD on them. This produces a set of "intrinsic mode functions" (IMF), which we separate by frequency range and re-aggregate them into distinct bands. Then, we inspect the relationship between the sensors by computing the corrcoeff within



(a) An example of EMD

(b) IMFs aggregation

(c) IMFs aggregation

Figure 6.2: (a) EMD decomposes a signal and exposes intrinsic oscillatory components; (b) Aggregation of IMFs within a pre-defined frequency range makes seemingly similar signals from different locations more distinguishable; (c) IMF aggregation makes seemingly distinct signals of different sensors in the same room show high correlation.

a particular band, which gives us the spatial information we are interested in. Finally, we separate the result set into sub-sets, and closely examine their statistical characteristics.

Categorical Verification

Categorical verification is the ability to cluster sensor traces by the type of sensor that it (i.e. its unit of measure) and the thing it is measuring. Ideally we should be able to separate feeds that are measuring different physical attributes and/or are placed in different parts of the building. For example, if there are sensors measuring temperature in a pipe and temperature in a room we should be able to separate them from one another as well as separate them from sensor pressuring pressure in a valve. We will show that this can be by examining their distribution and for small data sets with different sensor our approach works quite well, however we present some challenges in a large, more realistic deployment data set and we discuss why it is so challenging to deal with.

6.3 Experimental Building Deployments and Datasets

In this thesis, we run all of our experiments and analysis on data from the four buildings. They serve as the main sites for experiemental data collection and/or sites from which we obtained data dumps from their building management system.

Cory Hall - UC Berkeley

Cory Hall, at UC Berkeley, is a 5-story building hosting mainly classrooms, meeting rooms, laboratories and a datacenter. The HVAC system in the building is centralized and serves several floors per unit. There is a separate unit for an internal fabricated laboratory, inside the building.

Sutardja Dai Hall - UC Berkeley

Sutardja Dai Hall is a large building at Berkeley. It is a 7-story, 141000 square-foot building which contains classrooms, meeting rooms, laboratories, a nano-fabrication laborty, and a cafe. The HVAC system in the building is centralized and serves several floors per unit. There is also a separate unit for an internal fabricated laboratory, inside the building. It was built in 2009.

Soda Hall - UC Berkeley

Soda Hall is another building at Berkeley. It is the main building for the computer science department. It was built in 1994 and also has 7 floors and a centralized HVAC system. It contains classrooms, offices, server rooms, and open office spaces for several labs.

Engineering Building 2 - Todai

Engineering building 2, at the University of Tokyo (Todai), is a 12-story building completed in 2005. It contains classrooms, laboratories, offices and server rooms. The electricity consumption of the lighting and HVAC systems of 231 rooms is monitored by 135 sensors. Rather than a centralized HVAC system, small, local HVAC systems are set up throughout the buidling. The HVAC systems are classified into two categories, EHP (Electrical Heat Pump) and GHP (Gas Heat Pump). The GHPs are the only devices that serve numerous rooms and multiple floors. The 5 GHPs in the dataset serve 154 rooms. The EHP and lighting systems serve only pairs of rooms and which are directly controlled by the occupants. In addition, the sensor metadata provides device-type and location information (room number), therefore, the electricity consumption of each pair of rooms is separately monitored.

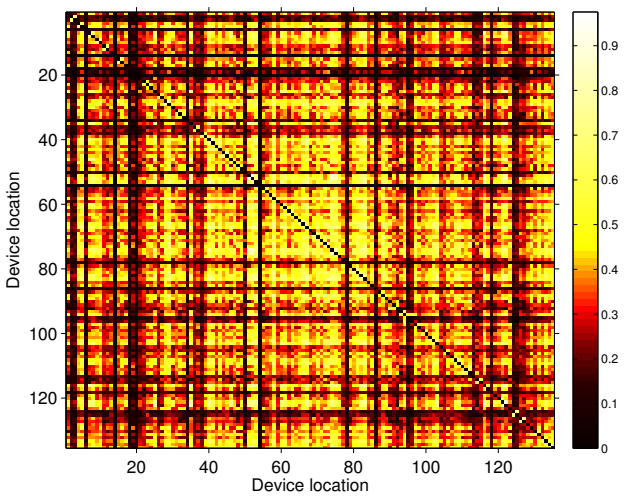


Figure 6.3: Correlation coefficients of the raw traces from the Building 1 dataset (Section 6.3). The matrix is ordered such as the devices serving same/adjacent rooms are nearby in the matrix.

6.4 Functional Verification Methodology

We run SBS on a set of building sensor traces; each containing hundred sensors reporting data flows over 18 weeks from two separate buildings with fundamentally different infrastructures. We demonstrate that, in many cases, SBS uncovers misbehavior corresponding to inefficient device usage that leads to energy waste. The average waste uncovered is as high as 2500 kWh per device.

We validate the effectiveness of our approach using 10 weeks of data from a modern Japanese building containing 135 sensors and 8 weeks of data from an older American building containing 70 sensors. These experiments highlight the effectiveness of SBS to uncover device relationships in a large deployment of 135 sensors. Furthermore, we inspect the SBS results and show that the reported alarms correspond to significant opportunities to save energy. The major anomaly reported in the American building lasts 18 days and accounts for a waste of 2500 kWh. SBS also reports numerous small anomalies, hidden deep within the building's overall consumption data. Such errors are very difficult to find without SBS.

The primary objective of SBS is to determine *how* device usage patterns are correlated across all pairs of sensors and discover when these relationships change. The naive approach is to run correlation analysis on pairs of sensor traces, recording their correlation coefficients over time and examining when there is a statistically-significant deviation from the norm. However, this approach does not yield any useful information when applied to *raw data traces*. For example, the two raw signals shown in Figure 6.5 are from two independent HVAC systems, serving different rooms on different floors. Since each space is independently

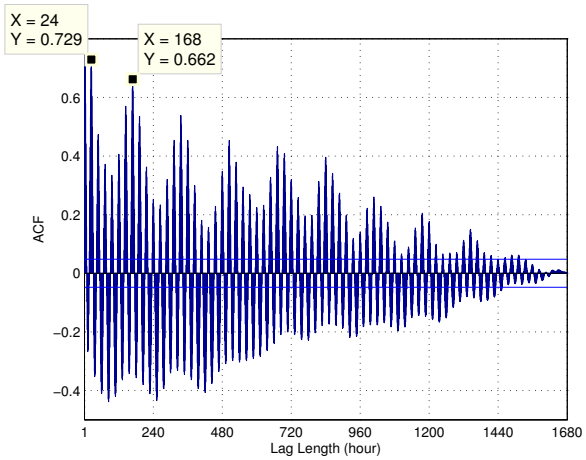


Figure 6.4: Auto-correlation of a usual signal from the Building 1 dataset. The signal features daily and weekly patterns (resp. $x = 24$ and $x = 168$).

controlled, we expect their power-draw signals to be uncorrelated (or at least distinguishable from other signal pairs). However, their correlation coefficient (0.57), is not particularly informative – it is statistically similar to the correlation between itself and other signals in the trace.

Using a larger set of devices, Figure 6.3 shows a correlation matrix with 135 distinct lighting and HVAC systems serving numerous rooms in a building (described later on in Section 6.3). The indices are selected such that their index-difference is indicative of their relative spatial proximity. For example, a device in location 1 is closer in the building to a device in location 2 than it is to a device in location 135. The color of the cell is the average pairwise correlation coefficient for devices in the row-column index. The higher the value, the lighter the color. Devices serving the same room are along the diagonal. Because these devices are used simultaneously, we expect high average correlation scores, lighter shades, along the diagonal figure. However, we observe no such pattern. Most of the signals are correlated with all the others and we see no discernible structure.

An explanation for this is that the daily occupant usage patterns drive these results. Figure 6.5 demonstrates this more clearly. It shows two 1-week raw signals traces which feature the same diurnal pattern. This trend is present in almost every sensor trace, and, it hides the smaller fluctuations providing more specific patterns driven by local occupant activity. Upon deeper inspection, we uncovered several dominant patterns, common among energy-consuming devices in buildings [**wrinch:pes2012**]. Figure 6.4 depicts the auto-correlation of a usual electric power signal for a device. The two highest values in the figure correspond to a lag of 24 hours and 168 hours (one week). Therefore, the signal has some periodicity and similar (though not equal) values are seen at daily and weekly time scales. The daily pattern is due to daily office hours and the weekly pattern corresponds to weekdays and weekends.

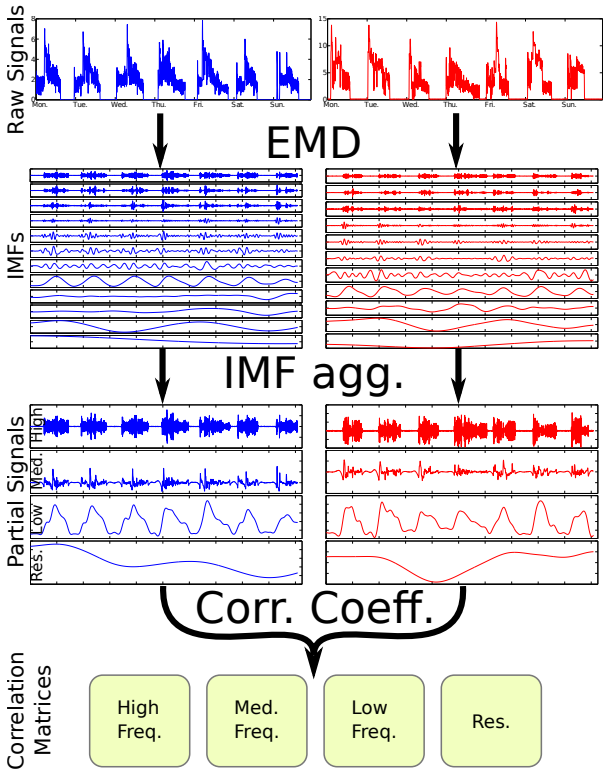


Figure 6.5: *Strip and Bind* using two raw signals standing for one week of data from two different HVACs. (1) Decomposition of the signals in IMFs using EMD (top to bottom: c_1 to c_n); (2) aggregation of the IMFs based on their time scale; (3) comparison of the partial signals (aggregated IMFs) using correlation coefficient.

Correlation analysis on *raw* signals cannot be used to determine meaningful inter-device relationships because periodic components act as non-stationary trends for high-frequency phenomenon, making the correlation function irrelevant. Such trends must be removed in order to make meaningful progress towards our aforementioned goals.

In the next section we describe SBS. We discuss *strip and bind* in section 6.4, which addresses de-trending and relationship-discovery. Then, we describe how we *search* for changes in usage patterns, in section 6.4, to identify potential savings opportunities.

Strip and Bind

Discovering devices that are used in concert is non-trivial. SBS decomposes each signal into an additive set of components, called Intrinsic Mode Functions (IMF), that reveals the signal patterns at different frequency bands. IMFs are obtained using Empirical Mode Decomposition (see Figure 6.5 and Section 6.2). We only consider IMFs with time scales shorter than a day, since we are interested in capturing short-scale usage patterns. Consequently,

SBS aggregates the IMFs that fall into this specific time scale (see *IMF agg.* in Figure 6.5). The resulting partial signals of different device power traces are compared, pairwise, to identify the devices that show un/correlated usage patterns (see *Corr. Coeff.* in Figure 6.5).

The IMFs are clustered using four time scale ranges:

- The *high frequencies* are all the IMFs with a time scale lower than 20 minutes. These IMFs capture the noise.
- The *medium frequencies* are all the IMFs with a time scale between 20 minutes and 6 hours. These IMFs convey the detailed devices usage.
- The *low frequencies* are all the IMFs with a time scale between 6 hours and 6 days. These IMFs represent daily device patterns.
- The *residual data* is all data with a time scale higher than 6 days. This is mainly residual data obtained after applying EMD. Also, it highlights the main device trend.

These time scale ranges are chosen based on our experiments and goal. The 20-minute boundary relies on the sampling period of our dataset (5 minutes) and permits us to capture IMFs with really short periods. The 6-hour boundary allows us to analyze all patterns that have a period shorter than the usual office hours. The 6-day boundary allows us to capture daily patterns and weekday patterns.

Aggregating IMFs, within each time scale range, results in 4 partial signals representing different characteristics of the device's energy consumption (see *Partial Signals* in Figure 6.5). We do a pairwise device trace comparison, calculating the correlation coefficient of their partial signals. In the example shown in Figure 6.5, the correlation coefficient of the raw signals suggests that they are highly correlated (0.57). However, the comparison of the corresponding *partial signals* provides new insights; the two devices are poorly correlated at high and medium frequencies (respectively -0.01 and -0.04) but highly correlated at low frequencies (0.79) meaning that these devices are not "intrinsically" correlated. They only share a similar daily pattern.

All the devices are compared pairwise at the four different time scale ranges. Consequently, we obtain four correlation matrices that convey device similarities at different time scales. Each line of these matrices (or column, since the matrices are symmetric) reveals the behavior of a device – its relationships with the other devices at a particular time scale. The matrices form the basis for tracking the behavior of devices and to search for misbehavior.

Search

Search aims at identifying misbehaving devices in an unsupervised manner. Device behavior is monitored via the correlation matrices presented in the previous section. Using numerous observations SBS computes a specific reference that exhibits the normal inter-device usage pattern. Then, SBS compares the computed reference with the current data and reports devices that deviate from their usual behavior.

Reference Matrix

We define four reference matrices, which capture normal device behavior at the four time scale ranges defined in Section 6.2. The references are computed as follows: (1) we retrieve the correlation matrices for n consecutive time bins. (2) For each pair of devices we compute the median correlation over the n time bins and obtain a matrix of the median device correlations.

Formally, for each time scale range the computed reference matrix for d devices and n time bins is:

$$R_{i,j} = \text{median}(C^1_{i,j}, ..., C^n_{i,j})$$

where i and j ranges in $[1, d]$.

Because anomalies are rare by definition, we assume the data used to construct the reference matrix is an accurate sample of the population; it is unbiased and accurately captures the range of normal behavior. Abnormal correlation values, that could appear during model construction, are ignored by the median operator thanks to its robustness to outlier (50% breakdown point). However, if that assumption does not hold (more than 50% of the data is anomalous), our model will flag the opposite – labeling abnormal as normal and vice-versa. From close inspection of our data, we believe our primary assumption is sound.

Behavior change

We compare each device behavior, for all time bins, to the one provided by the reference matrix. Consider the correlation matrix C^t obtained from the data for time bin t ($1 \leq t \leq n$). Vector $C^t_{i,*}$ is the behavior of the i^{th} device for this time bin. Its normal behavior is given by the corresponding vector in the reference matrix $R_{i,*}$. We measure the device behavior change at the time bin t with the following Minkowski weighted distance:

$$l^t_i = \left(\sum_{j=1}^{d} w_{ij} \left(C^t_{i,j} - R_{i,j} \right)^p \right)^{1/p}$$

where d is the number of devices and w_{ij} is:

$$w_{ij} = \frac{R_{i,j}}{\sum_{k=1}^{d} R_{i,k}}.$$

The weight w enables us to highlight the relationship changes between the device i and those highly correlated to it in the reference matrix. In other words, our definition of behavior change is mainly driven by the relationship among devices that are usually used in concert. We also set $p = 4$ in order to inhibit small differences between $C^t_{i,j}$ and $R_{i,j}$ but emphasize the important ones.

By monitoring this quantity over several time bins the abnormal device behaviors are easily identified as the outlier values. In order to identify these outlier values we implement a

robust detector based on median absolute deviation (MAD), a dispersion measure commonly used in anomaly detection [**huber:wiley2009**, **chan:springer2005**]. It is a measure that robustly estimates the variability of the data by computing the median of the absolute deviations from the median of the data. Let $l_i = [l_i^1, ..., l_i^n]$ be a vector representing the behavior changes of device i over n time bins, then its MAD value is defined as:

$$\text{MAD}_i = b\,\text{median}(|l_i - \text{median}(l_i)|)$$

where the constant b is usually set to 1.4826 for consistency with the usual parameter σ for Gaussian distributions. Consequently, we define anomalous behavior, for device i at time t, such that the following equation is satisfied:

$$l_i^t > \text{median}(l_i) + \tau\,\text{MAD}_i$$

Note, τ is a parameter that permits to make SBS more or less sensitive.

The final output of SBS is a list of alarms in the form (t, i) meaning that the device i has abnormal behavior at the time bin t. The priority of the alarms in this list is selected by the building administrator by tuning the parameter τ.

We evaluate SBS using data collected from buildings in two different geographic locations. One is a new building on main campus of the University of Tokyo and the other is an older building at the University of California, Berkeley.

Data pre-processing is not generally required for the proposed approach. Nevertheless, we observe in a few exceptional cases that sensors reporting excessively high values (i.e. values higher than the device actual capacity) that greatly alter the performance of SBS by inducing a large bias in the computation of the correlation coefficient. Therefore, we remove values that are higher than the maximum capacity of the devices, from the raw data.

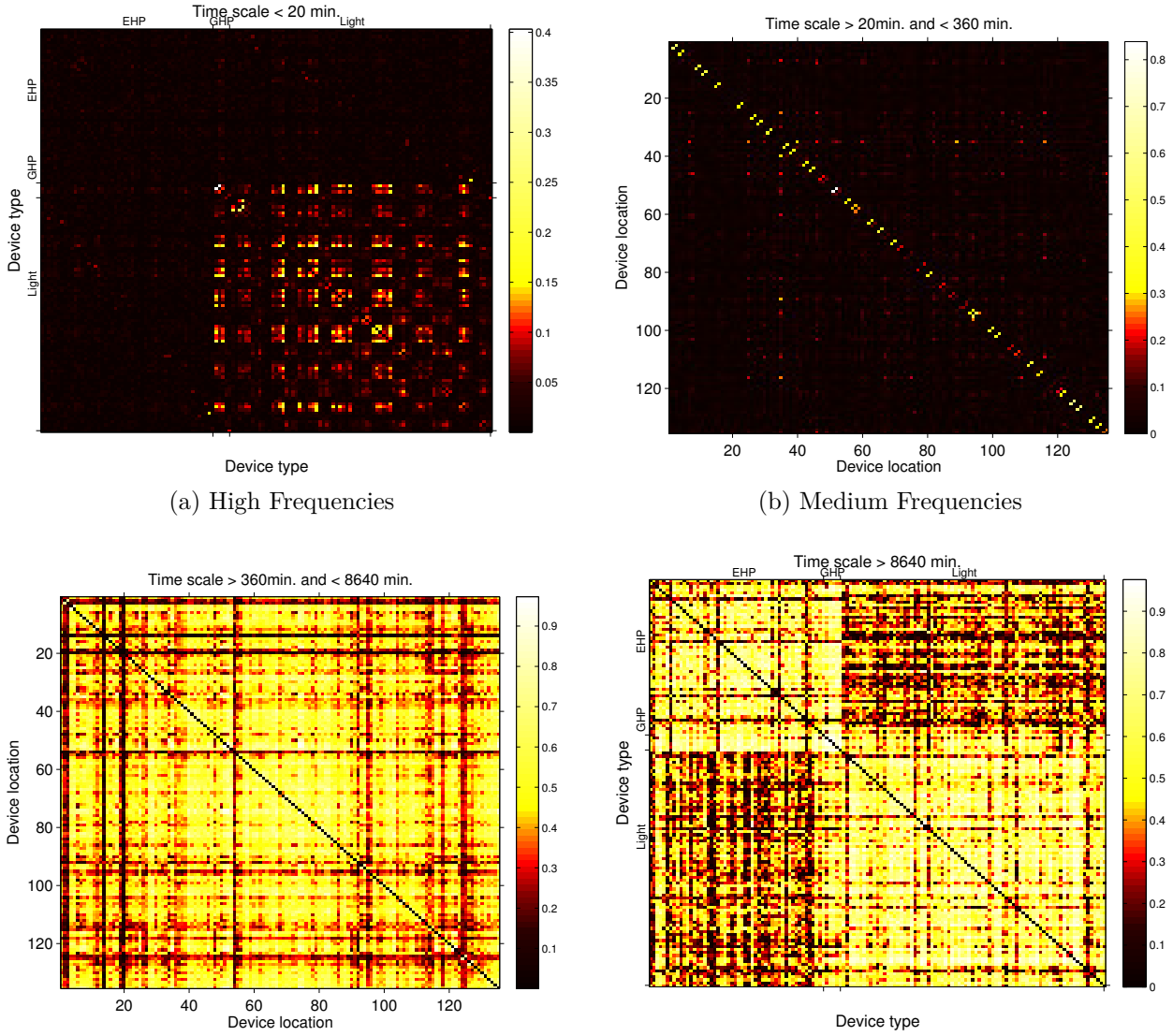
The Todai dataset we use contains 10 weeks of data starting from June 27, 2011 and ending on September 5, 2011. This period of time is particularly interesting for two reasons: 1) in this region, the summer is the most energy-demanding season and 2) the building manager actively works to curtail energy usage as much as possible due to the Tohoku earthquake and Fukushima nuclear accident.

Furthermore, this dataset is a valuable ground truth to evaluate the Strip and Bind portions of SBS. Since the light and HVAC of the rooms are directly controlled by the room's occupants, we expect SBS to uncover verifiable devices relationships.

The Cory Hall dataset we use consists of 8 weeks of energy consumption traces measured by 70 sensors starting on April 5^{th}, 2011. In contrast to the other dataset, a variety of devices are monitored, including, electric receptacles on certain floors, most of the HVAC components, power panels and whole-building consumption.

These two building infrastructures are fundamentally different. This enables us to evaluate the practical efficacy of the proposed, unsupervised method in two very different environments.

In this section we evaluate SBS on our building traces. We demonstrate the benefits of striping the data by monitoring patterns captured at different time scales. Then, we thoroughly investigate the alarms reported by SBS.



(a) High Frequencies

(b) Medium Frequencies

(c) Low Frequencies

(d) Residual data

Figure 6.6: Reference matrices for the four time scale ranges (the diagonal $x = y$ is colored in black for better reading). The medium frequencies highlight devices that are located next to each other thus intrinsically related. The low frequencies contains the common daily pattern of the data. The residual data permits to visually identify devices of the similar type.

Device behavior at different time scales

The Strip and Bind part of SBS is evaluated using the data from Eng. Bldg 2. This dataset is appropriate to measure SBS's performance, since lighting and HVAC systems serving the same room are usually used simultaneously. Consequently, we analyze this data using SBS and verify that the higher correlations at medium frequencies correspond to devices

located in the same room.

The dataset is split into 10, one-week bins and each bin is processed by SBS. Using the 10 correlation matrices at each time scale range, SBS uncovers the four reference matrices depicted in Figure 6.6.

High frequencies In this work the high frequencies correspond to the signals *noise*, therefore, we do not expect any useful information from the corresponding matrix (Figure 6.6a). Indeed, the corresponding reference matrix does not provide any help to determine a device's relative location. Thus, we emphasize that high frequency data should be ignored for uncovering device relationships (in contrast to [**romain:iotapp12**]). Interestingly, we find that the sensors monitoring the lights generate consistent noise.

Medium frequencies Our main focus is on the medium frequencies as it is designed to capture the intrinsic device relationships. Figure 6.6b shows the correlation matrix at medium frequencies. It is significantly different from the one obtained with the raw signals (Figure 6.3): high correlation coefficients are concentrated along the matrix diagonal. Since devices serving the same or adjacent rooms are placed nearby in the matrix it validates our hypothesis: *high correlation scores within the medium frequency band shows strong inter-device relationships.*

Considering this reference matrix as an adjacency matrix of a graph, in which the nodes are the devices, we identify the clusters of correlated devices using a community mining algorithm [9]. As expected we obtain mainly clusters of only two devices (light and HVAC serving the same room), but we also find clusters that are composed of more devices. For example a cluster contains 3 HVAC systems serving the three server rooms. Although these server rooms are located on different floors, SBS shows a strong correlation between these devices. Coincidentally, they are managed similarly. Interestingly, we also observe a couple of clusters that consist of independent devices serving adjacent rooms belonging to the same lab. The bigger cluster contains 33 devices that are 2 GHP devices and the corresponding lights. This correlation matrix and the corresponding clusters highlight the ability for SBS to identify such hidden inter-device usage relationships.

Low frequencies Low frequencies capture daily patterns, embedded in all the device traces. Figure 6.6c depicts the corresponding reference matrix which is similar to the one of raw signal traces (Figure 6.3) and it shows no particular structure.These partial signals are discarded as they do not help us in identifying inter-device usage patterns.

Residual data The residual data shows the weekly trend, which gives us no information about device relationships. But, surprisingly, by reordering the correlation matrix based on the type of the devices (Figure 6.6d) we can visually identify two major clusters. The first cluster consists of HVAC devices (see EHP and GHP in Figure 6.6d) and the second one contains only lights. An in-depth examination of the data reveals that long-term trends are

inherent to the device types. For example, as the consumption of both the EHP and GHP devices is driven by the building occupancy and the outside temperature, these two types of devices follow the same trend. However, the use of light is independent from the outside temperature thus the lighting systems follow a common trend different from the EHP and GHP one.

We conduct the same experiments by splitting the dataset in 70 bins of 1 day long and observe analogous results at high and medium frequencies but not at lower frequencies. This is because the bins are too short to exhibit daily oscillations and the residual data captures only the daily trend.

Methodological Shortcomings

Because our analysis is done on historical data, some of the faults found by SBS could not be fully corroborated. In order to fully examine the effectiveness of our approach, we must run it in real time and physically check that the problem is actually occurring. When a problem is detected in the historical trace, months after it has occurred, the current state of the building may no longer reflect what is in the traces. Some of the anomalies discussed in this section uncover interpretable patterns that are difficult to find in practice. For example, simultaneous heating and cooling is a known, recurring problem in buildings, but it is very hard to identify when it is occurring. Some of the anomalies we could not interpret might be interpretable by a building manager, however, we did not consult either building manager for this study. Therefore, the results of this study do not examine the true/false positive rate exhaustively.

The true/false negative rate is impractical to assess. It may be examined through synthetic stimulation of the building via the control system. However, getting cooperation from a building manager to hand over control of the building for experimentation in non-trivial. Therefore, we forgo a full true/false negative analysis in our evaluation.

Because of these challenges, the evaluation of SBS focuses on comparing the output with known fault signatures. We examine anomalies, in either building, where the anomaly is easily interpretable but difficult to find by the building manager. We forego a comparison of SBS with competing algorithms because related algorithms require detailed knowledge of the building, *a priori*. The advantage of SBS is that it requires no such information to provide immediate value.

6.5 Functional Verification Experimental Results

We evaluate the *search* performance of SBS using the traces from the Eng. Bldg 2 and Cory Hall. Due to the lack of historical data, such as room schedule or reports of energy waste, the evaluation is non-trivial. Furthermore, getting ground truth data from a manual inspection of the hundreds traces of our data sets is impractical. The lack of ground truth data prevents us from producing a systematic analysis of the anomalies missed by SBS (i.e.

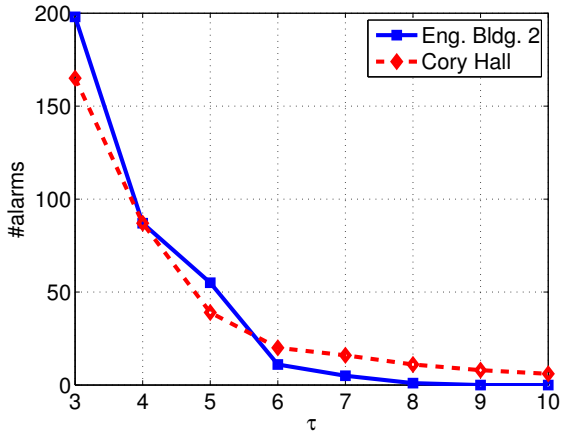


Figure 6.7: Number of reported alarms for various threshold value ($\tau = [3, 10]$).

false negatives rate). Nevertheless, we exhibit the relevance of the anomalies uncovered by SBS (i.e. high true positive rate and low false positive rate) by manually checking the output of SBS.

Anomaly classification To validate SBS results we manually inspect the anomalies detected by the algorithm. For each reported alarm (t, i) we investigate the device trace i and the devices correlated to it to determine the reason for the alarm. Specifically, we retrieve the major relationship change that causes the alarm (i.e. $\max(|w_j(C^t_{i,j} - R_{i,j})|)$, see Section 6.4) and examine the metadata associated to the corresponding device. This investigation allows us to classify the alarms into five groups:

- *High power usage*: alarms corresponding to electricity waste.
- *Low power usage*: alarms representing the abnormally low electricity consumption of a device.
- *Punctual abnormal usage*: alarms standing for short term (less than 2.5 hours) raise or drop of the electricity consumption.
- *Missing data*: alarms raised due to a sensor failure.
- *Other*: alarms whose root cause is unclear.

Experimental setup For each experiment, the data is split in time bins of one day, starting from 09:00 a.m. – which is approximately the office's opening time. We avoid having bins start at midnight since numerous anomalies appear at night and they are better

	High	Low	Punc.	Missing	Other
Eng. Bldg 2	9 (5)	6 (5)	1 (1)	36 (1)	3 (3)
Cory Hall	25 (7)	7 (3)	4 (4)	0 (0)	3 (3)

Table 6.1: Classification of the alarms reported by SBS for both dataset (and the number of corresponding anomalies).

highlighted if they are not spanning two time bins. Only the data at medium frequencies are analyzed, the other frequency bands are ignored, and the reference matrix is computed from all time bins.

The threshold τ tunes the sensitivity of SBS, hence, the number of reported alarms. Furthermore, by plotting the number of alarms against the value of τ for both datasets (Figure 6.7) we observe an elbow in the graph around $\tau = 5$. With thresholds lower than this pivot value ($\tau < 5$), the number of alarms significantly increases, causing less important anomalies to be reported. For higher values ($\tau > 5$), the number of alarms is slowly decreasing, providing more conservative results that consist of the most important anomalies. This pivot value provides a good trade off for either data set.

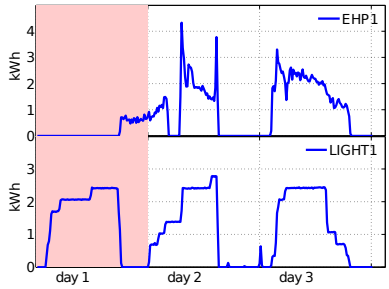
Table 6.1 classifies the alarms reported by SBS on both datasets. Anomalies spanning several time bins (or involving several devices) may raise several alarms. We display these in Table 6.1 as numbers in brackets – the number of anomalies corresponding to the reported alarms.

Alarms in Todai

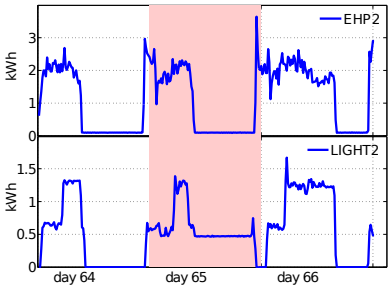
SBS reported 55 alarms over the 10 weeks of the Eng. Bldg 2 dataset. However, 36 alarms are set aside because of sensor errors; one GHP has missing data for the first 18 days. Since this device is highly correlated to the GHP in the reference matrix, their relationship is broken for the 18 first bins and for each bin one alarm per device is raised.

In spite of the post-Fukushima measures to reduce Eng. Bldg 2's energy consumption, SBS reported 9 alarms corresponding to high power usage (Table 6.1). Figure 6.8a depicts the electricity consumption of the light and EHP in the same room where two alarms are raised. Because the EHP was not used during daytime (but is turned on at night, when the light is turned off) the relationship between the two devices is "broken" and an alarm is raised for each device. Figure 6.8b shows another example of energy waste. The light is on at night and the EHP is off. The top-priority anomaly reported by SBS is caused by the 10 days long constant use of an EHP (Figure 6.8d) and this waste of electricity accounts for 165 kWh. SBS partially reports this anomaly but lower values of τ permits us to identify most of it.

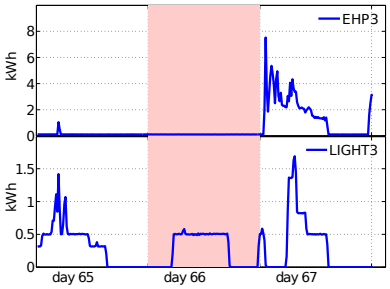
We observed 6 alarms corresponding to abnormally low power use. Upon further inspection we notice that it corresponds to energy saving initiatives from the occupants – likely



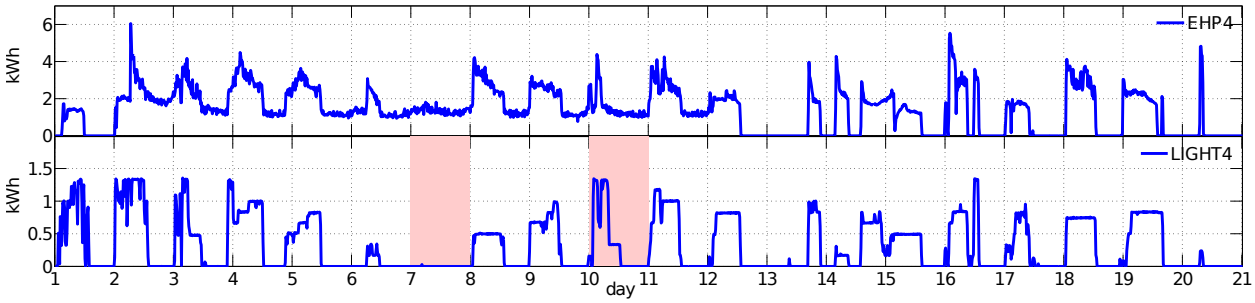
(a) High power usage where the HVAC (EHP) is turned on at night



(b) High power usage where the light is left on at night



(c) Low power usage where the HVAC (EHP) is not used during office hours



(d) Long term high power usage partially detected

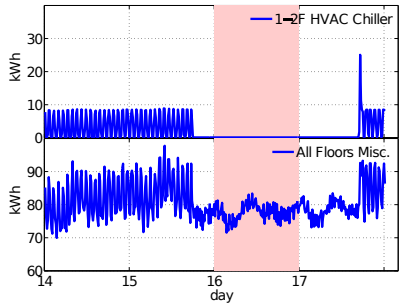
Figure 6.8: Example of alarms (red rectangles) reported by SBS on the Eng. Bldg 2 dataset

due to electricity concerns in Japan. This behavior is displayed in Figure 6.8c. The room is occupied at the usual office hours (indicated by light usage) but the EHP is not on in order to save electricity.

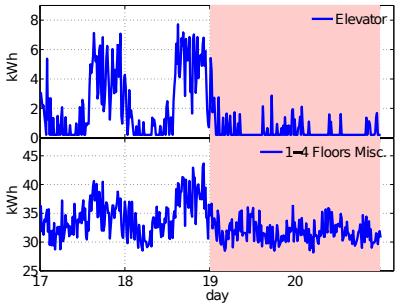
Alarms in Cory Hall

SBS reported 39 alarms for the Cory Hall dataset (Table 6.1). 7 are classified as low power usage, however, our inspection revealed that the root causes are different than for the Eng. Bldg 2 dataset. We observe that the low power usage usually corresponds to device failures or misconfiguration. For example, Figure 6.9a depicts the electricity consumption of the 2^{nd} floor chiller and a power riser that comprises the consumption of multiple systems, including the chiller. As the chiller suddenly stops working, the correlation between both measurements is significantly altered and an alarm for each device is raised.

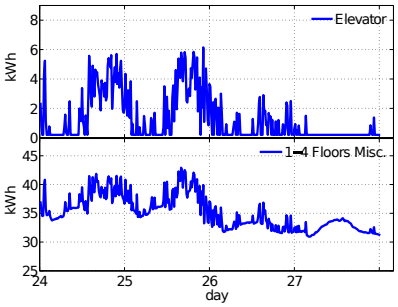
SBS also reports 25 alarms corresponding to high power usage. One of the identified anomalies is particularly interesting. We indirectly observe abnormal usage of a device from the power consumption of the elevator and a power panel for equipment from the 1^{st} to the 4^{th} floor. Figure 6.9b and 6.9c show the electricity consumption for both devices. SBS uncovers the correlation between the these two signals, as the amount of electricity



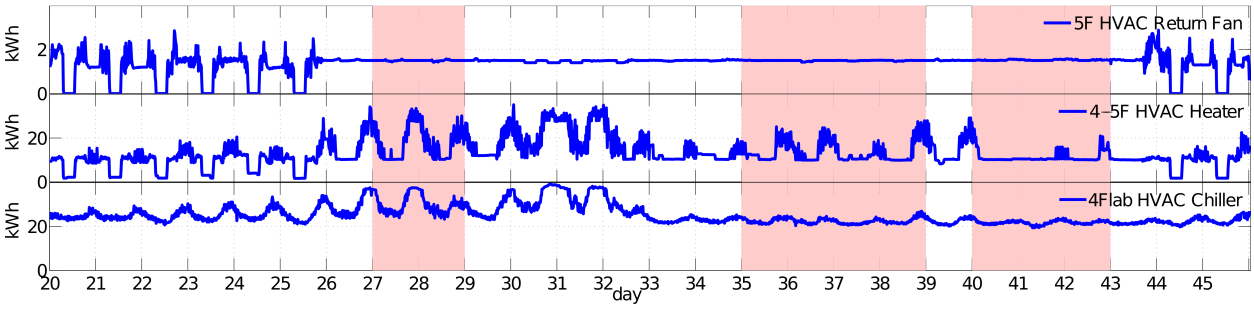
(a) Low power usage due to a chiller failure



(b) High power usage highlighted by the elevator usage



(c) Normal power and elevator usage



(d) Long term high power usage due to competing heating and cooling

Figure 6.9: Example of alarms (red rectangles) reported by SBS on the Cory Hall dataset

going through the panel fluctuates along with the elevator power consumption (Figure 6.9c). In fact, the elevator is a good indicator of the building's occupancy. Anomalous energy-consumption is identified during a weekend as the consumption measured at the panel is independently fluctuating from the elevator usage. These fluctuations are caused by a device that is not directly monitored. Therefore, we could not identify the root cause more precisely. Nevertheless, the alarm is worthwhile for building operators to start investigating.

The most important anomaly identified in Cory Hall is shown in Figure 6.9d. This anomaly corresponds to the malfunctioning of the HVAC heater serving the 4^{th} and 5^{th} floors. The heater is constantly working for 18 consecutive days, regardless of the underlying occupant activity. Moreover, in order to maintain appropriate temperature this also results in an increase of the 4^{th} floor HVAC chiller power consumption and several fans, such as the one depicted in Figure 6.9d. This situation is indicative of simultaneous heating and cooling – whereby heating and cooling systems are competing – and it is a well-know problem in building management that leads to significant energy waste. For this example, the electricity waste is estimated around 2500 kWh for the heater. Nevertheless, as the anomaly spans over 18 days, it is hidden in the building's overall consumption, thus, it is difficult to detect by building administrators without SBS.

6.6 Spatial Verification Methodology

We examine the use of EMD for verifying spatial relationships. We run our separate analyses on two separate data sets. The first is a data set at Todai and the other is a data set in Sutardja Dai Hall at UC Berkeley. For the Todai data set, we use a simple methodology whereby we run a pairwise correlation analysis on the IMFs for traces in that building. We create clusters of traces that share a high correlation value and examine the spatial characteristics of the clusters as we sweep through the acceptance threshold on the correlation values.

The second analysis is on a separate data set in Sutardja Dai Hall. There, we expand the methodology from the Todai dataset and apply machine learning techniques to systematize the clustering processes. We also examine the effectiveness of our clustering algorithm as we sweep through a series of threshold values. We present both methodologies and results in this section.

Todai Data Set Analysis Methodology

For the first investigation, we focus on a three-week span in the summer of 2011 (from July 4th to July 24th). The dataset captures regular work days, weekends, and one holiday (July 18th). This timeframe captures the typical usage of the equipment, triggered by occupant activity. For the initial analysis, we focus on three sensors; two pumps – eletric heat pump (EHP) and a gas heat pump (GHP) and a light feed, that measure the light level is lumens. The room lighting system serves the same room as the EHP. The GHP serves a different room on the same floor. The expanded portion of this analysis pivots on the EHP and does a pairwise comparison between it and all other sensors in the building. We use EMD to detrend each of the traces and pay particularly close attention to the high-frequency IMFs. Our hypothesis is that correlating at the higher frequencies will yield more meaningful comparisons. In buildings, metadata is poorly and unsystematically managed within a single system domain. Moreover, with the ever growing number of additional sub-meters, it is important to quickly integrate sensor data from multiple systems to understand the full state of the building. It is also important to understand how sensors are used in concert. Anomalies in usage may indicate underlying problems with the equipment or inefficient/incorrect usage.

Sutardja Dai Hall Analysis Methodology

The methodology used in the SDH is more extensive and consists of several steps. Each is discussed in great details in this section.

Distribution Analysis

For the second part of our evaluation, we perform an empirical study on sensor data collected from 15 sensors across 5 rooms. Each room has three sensors: a temperature sensor, a CO_2 sensor, and a humidity sensor. The data from these is reported to an sMAP [19] archiver. The data set used comes from two separate deployments: one is a deployment [72] lasting over 6 months on several floors in Sutardja Dai Hall (SDH) at UC Berkeley, where one sensor box – which contains a thermometer, a humidity sensor and a CO_2 sensor – is placed in each room. The box reports data over 6LowPAN [42] to a sMAP archiver every 15 seconds. The other is a long-term deployment comprised of thousands of sensors in dozens of buildings on campus. We choose the portion of the SDH data set where the sensor devices, accessible via BACnet [**BACnet**], report data to the archiver every few minutes. Due to intermittent data loss, we pick a time span without interruption, starting in January until mid-February, 2013, for evaluation.

Let $ts^i_{j,t}$ be a time-series for sensor j in room i observed over some time interval t. For simplicity, we ignore t in defining subsequent functions and re-introduce it where necessary. For each trace we run EMD and obtain a set of n IMFs, denoted as follows:

$$\Phi^i_j = EMD(ts^i_j) = \{IMF_{1\sim n}\}$$

IMFs are traces themselves, so we divide and re-aggregate them into the four bands, B, further described in Section **??**.

$$B = \{H(igh), M(edium), L(ow), R(esidue)\}$$

Let the re-aggregation of the bands be denoted as:

$$Aggr(\Phi^i_j) = \left\{IMF^i_{f,j}\right\}$$

where $f \in B$. We pick the *medium* frequency band (M) to compute the pairwise corrcoeff of the sensor traces. In order to understand and characterize the boundary between sensors we consider two sets of corrcoeffs for each room; the "intra"-room set and "inter"-room set, as defined:

$$R^i_{intra,t} = \left\{r(IMF^i_{M,j,t}, IMF^i_{M,k,t})\right\},\ s.t. \forall j,k \in S_i$$

The intra set only contains pairs of sensors in the same room, so both $ts^i_{j,t}$ and $ts^i_{k,t}$ are traces from sensors in room i.

$$R^i_{inter,t} = \left\{r(IMF^i_{M,j,t}, IMF^{i'}_{M,k,t})\right\},$$

$$s.t. \forall j \in S_i,\ \forall k \in S'_i,\ i \neq i'$$

By contrast, the *inter* set contains pairs across rooms, meaning $ts_{j,t}$ is a trace from a sensor in room i and $ts_{k,t}$ is a sensor trace from some other room i'. Note the use of t in the

definitions. We re-introduce t here to denote that the construction of each set is performed with respect to a specific time interval.

Finally, we examine populations, R^i_{intra} and R^i_{inter}, across multiple time intervals (in days):

$$R^i_{intra} = \bigcup_{\forall t} R^i_{intra,t},\ s.t.\ t \in \{1, 3, 5, 7, 14, 21, 28\}$$

$$R^i_{inter} = \bigcup_{\forall t} R^i_{inter,t},\ s.t.\ t \in \{1, 3, 5, 7, 14, 21, 28\}$$

We generate a CDF for each of the two populations with respect to each room. This allows us to closely examine the statistical characteristics of the relationship between sensors in the same space and those in different spaces. Each room offers a potentially different perspective on this relationship.

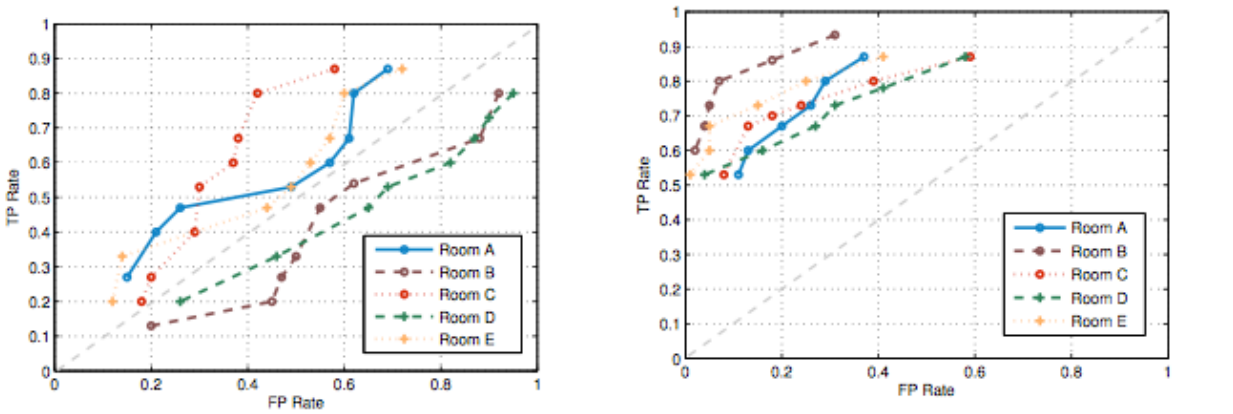
Threshold Analysis

In order to understand the statistical properties, we generate two corrcoeff distributions by computing the corrcoeff between pairs of traces within and across each room, as detailed in the previous section. Figure 6.15 shows how we divide the corrcoeff values into two sets. The figure shows two intra and two inter sets. Specifically, we examine how a choice in cut-off threshold affects the ability to separate the sets, when their separation is not known a priori, relative to each room. Our hypothesis is that there exists a computable, statistical boundary between sensors in different rooms.

To test our hypothesis, we choose a threshold value relative to the distribution of corrcoeffs. All pairs with a corrcoeff larger than the threshold will be classified as being in the same room. To closely analyze the threshold parameter, we generate a receiver operating characteristic (ROC) curve by varying the threshold value. Then, we look for a good trade-off point between the true-positive and false-positive rate; one that maximizes the difference between TPR and FPR. We compare the ROCs generated for our "medium" frequency band IMFs against raw-signal, cross-correlation values, in order to ascertain the extent to which the SBS [28] methodology is advantageous for discovering a statistical separation, analogous to a physical one. We also examine whether there is a uniform boundary between clusters across all the rooms.

Experimental Setup

We perform an empirical study on sensor data collected from 15 sensors across 5 rooms on 4 different floors of a Sutardja Dai Hall, as detailed in Table 6.2. Each room has three sensors: a temperature sensor, a CO_2 sensor, and a humidity sensor. The data from these is reported to an sMAP [19] archiver. The data set used comes from a deployment [72] lasting over 6 months on several floors in Sutardja Dai Hall (SDH) at UC Berkeley, where one sensor box – which contains a thermometer, a humidity sensor and a CO_2 sensor – is placed in each



(a) Correlating the raw signals.

(b) Correlating the re-aggregated IMFs in the "medium" frequency band.

Figure 6.10: The ROC curves depict the sensitivity of the raw signal and mid-frequency IMFs to the threshold value. We choose the 0.2 FPR point as the boundary threshold for each room.

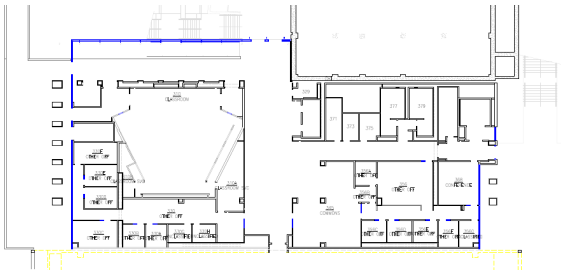


Figure 6.11: We collect data from 15 sensors in 5 rooms sitting on 4 different floors. This is a map of a section of the 3rd floor in Sutardja Dai Hall.

room. The box reports data over 6LowPAN [42] to a sMAP archiver every 15 seconds. Due to intermittent data loss, we pick a time span without interruption, starting in January until mid-Feburary, 2013, for evaluation.

Table 6.2: Room Specs

Room#	Orientation	Floor	Type
A	West	2	Computer Lab
B	South	4	Conference Room
C	No Window	2	Classroom
D	North	7	Conference Room
E	South	5	Conference Room

	Raw trace	1st IMF	2nd IMF	3rd IMF	Residual
EHP, Light	0.7715	0.43909	0.49344	0.63469	0.82132
EHP, GHP	0.6370	0.0060274	0.063546	0.16764	0.79378

Table 6.3: Correlation coefficients of the analyzed trace and their IMFs uncovered by EMD

6.7 Spatial Verification Results

Our initial results on the Todai dataset were not surprising. The diurnal pattern dominates the comparison between the sensors. Weather is the main driver for this behavior and it affects the readings in almost all of the sensors in our dataset. Cross-correlation on raw sensor data is insufficient for filtering intrinsically related behavior. Upon closer examination of the data we assess the following:

- The main underlying diurnal trend occurs in almost all the traces.
- Occupancy and room activities occur at random times during the day and change at a higher frequency than weather patterns.
- Sensors that serve the same location observe the same activities. Therefore, their underlying measurements should be correlated.

In order to uncover these relationships we must remove low-frequency trends in the traces and compare the readings at high frequencies.

We test our hypothesis in this section by using EMD to remove low-frequency trends in the data and run correlation calculation at overlapping IMF timescales. We discover that EMD allows us to find and compare high-frequency instrinsic behavior that is spatially correlated across sensors. We begin with a small set of three sensors (EHP, GHP, light) and expand our scope to include all the sensors in the dataset.

Initial Todai Analysis Results

Figure 6.12 shows the raw traces for the three devices discussed in the previous section (EHP, GHP, light). All three exhibit a diurnal usage pattern. On weekends, each draw less power. For our initial analysis, we calculated the pairwise correlation coefficient for all sensors in the set. The correlation coefficient for the EHP and light is 0.7715 and the correlation coefficient for the EHP and GHP is 0.6370. Running correlation across them yields high correlation coefficients, mostly due to their underlying daily usage pattern.

We would like to know if the EHP trace is correlated with the two other traces. Recall that the correlation coefficients of the raw feeds was 0.7715 and 0.6370, corresponding to the light and GHP, respectively. As stated in previous section this result is correct but not so meaningful, since most of the traces display the same diurnal pattern. Figure 6.12

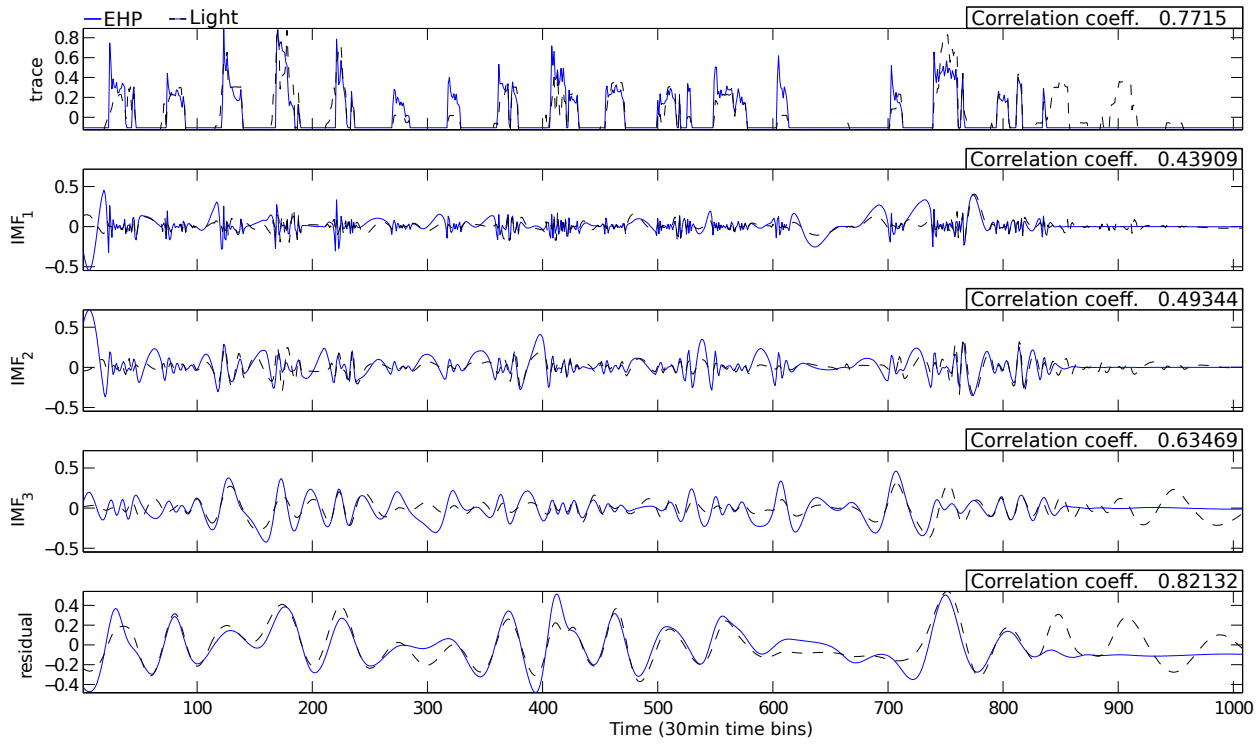


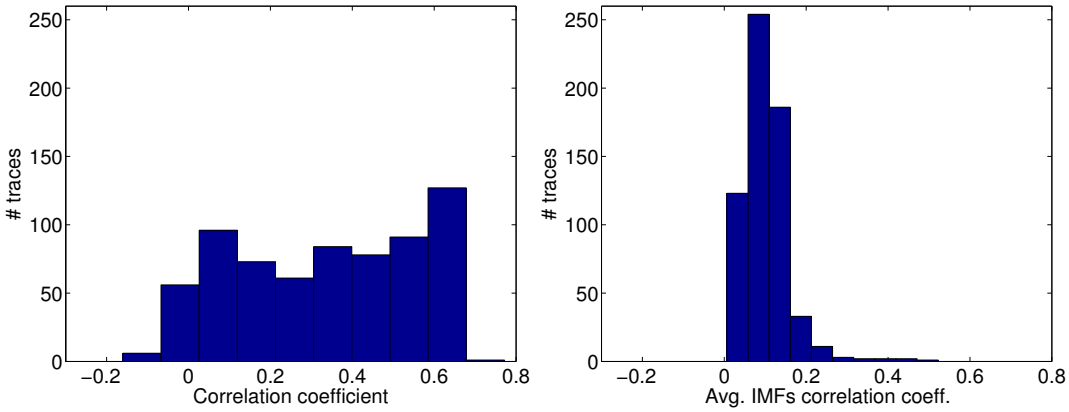
Figure 6.12: Decomposition of the EHP and light trace using bivariate EMD. IMFs correlation coefficients highlight the intrinsic relationship of the two traces.

and Figure **??** show the EMD decomposition of the three traces. For each trace, EMD has retrieved three IMFs that highlight the higher frequencies of the traces.

Figure 6.12 shows the normalized raw trace (top) and EMD output IMFs and residual as well as the correlation coefficients calculated on the IMFs for the EHP and light traces. The correlation coefficients are 0.43909, 0.49344 and 0.63469 corresponding to the IMF1, IMF2, and IMF3, respectively. Notice the high correlation between the high-frequency IMFs. We know that the light and EHP serve the same room, and their high-frequency IMF correlation corroborates our prior knowledge. Figure **??** shows a complementary result, for the EHP and GHP comparison. The correlation coefficients for the EHP and GHP IMFs suggest that the two may be independent. In fact, they *are* indepdent; they serve completely different rooms in the building. EMD allows us to remove low-frequency trends that add noise to the original analysis. By comparing IMFs, we see both intrisically correlated and *uncorrelated* behavior.

Initial Observations

We analyze the same three-week time span for *all* 674 sensors deployed at Todai. For each trace S we compute two scores: (1) the correlation coefficient between S and the EHP trace and (2) the average value of the IMF correlation coefficients.



(a) Raw traces correlation coefficients (b) Average IMFs correlation coefficients

Figure 6.13: Distribution of the correlation coefficients of the raw traces and correlation coefficients average of the corresponding IMFs using 3 weeks of data from 674 sensors.

Figure 6.13a shows the distribution correlation coefficients. Notice that a large fraction of the dataset is correlated with the EHP trace. *Half* the traces have a correlation coefficient higher than 0.36. As expected, the underlying trend is shared by a large number of device. Although the highest score (i.e. 0.7715) corresponds to the light in the same room that the EHP serves, there are 118 pumps, serving all areas of the building, with a correlation higher than 0.6. Using only these results, it is not clear where the threshold should be set. The distribution is close to uniform, making it difficult to know of how well your threshold discriminates against unrelated traces.

Figure 6.13b shows the distribution of the average correlation value for the IMFs of each trace and the EHP. The number of traces correlated in the high frequency IMFs is significantly smaller than the previous results. It's clear from the distribution that only a small set of devices are *intrinsically correlated* with the EHP. In fact, *only 10 traces out of 674* yielded a score higher than 0.25. This allows us to easily rank traces by correlation.

Upon closer inspection of the 10 most correlated IMF traces, we find that there is a spatial relationship between the EHP and the ten devices. In fact, there is a direct relationship between score and distance from the areas served by the EHP. Figure 6.14 shows a map of the floor that contains the rooms served by this EHP. The EHP directly serves room $C2$. We introduce a correlation threshold to cluster correlated traces by score. We highlight rooms by the threshold setting on the IMF correlation score. When we set the threshold at 0.5 we see that the sensors that have a correlation higher fall within room $C2$ – the room served

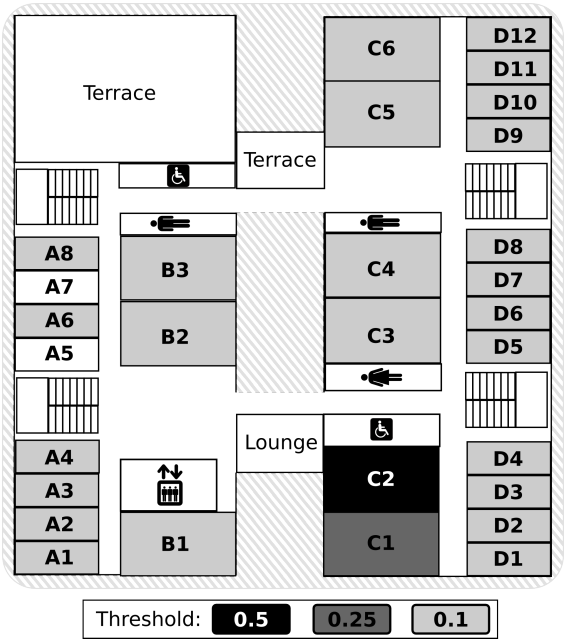


Figure 6.14: Map of the floor where the analyzed EHP serves (room $C2$). The location of the sensors identified as related by the proposed approach are highlighted, showing a direct relationship between IMF correlation and spatial proximity.

directly by the EHP. As we relax the threshold, lowering it to 0.25 and 0.1 we see radial expansion from $C2$. The trace with the highest score, 0.522, is the trace corresponding to the lighting system *in the same room*. The two highest scores for this floor (i.e. 0.316 and 0.279) are the light and EHP traces from next door, room $C1$. Lower values correspond to sensors measuring activities in other rooms that have no specific relationship to the analyzed trace. The results show a direct relationship between IMF correlation and spatial proximity and *supports our initial hypothesis*.

EMD is useful for finding underlying behavioral relationships between traces of sensor data. However, when we set the timescales smaller than a day, the results were not as strong. The trace has to be long enough to capture the trend. For this data set, the underlying trend is daily, therefore it requires there to be a significant number of samples over many days. Although this was a limitation for this dataset, it really depends on the underlying phenomenon that the sensors are measuring. Its underlying trend is ultimately what EMD will be able to separate from the intrinsic modes of the signal.

Figure **??** shows a comparison of two temperature sensor feeds from different rooms and their respective decomposition. Despite strong correlation in the raw time series, the medium frequency IMF shows little correlation. Only the low frequency diurnal pattern is correlated. Alternatively, Figure **??** shows a CO_2 trace and a humidity trace.

While the raw signals appear to be very different, and indeed have modest correlation,

the medium frequency components are strongly correlated. We conjecture that the medium frequency band "records" local activity. Occupants and movement in the space affect the levels of various physical phenomenon, namely temperature, humidity, CO_2 levels, etc. Over shorter time spans, noise in the system hides the effects of local activity. Longer time-spans capture long-term trends related to weather or building operation schedules. The medium frequency band captures activities such as meetings and office occupation times. These examples illustrate the basis for an automated process. By isolating a particular component of the signal we seek to strip away common diurnal factors and also eliminate differences in the response of various sensors to environmental factors. We combine this observation with a simple classifier to derive colocation.

SDH Spatial Clustering Results

We conduct two sets of experiments. First, we quantify the sensitivity of our method for different threshold values and examine the effect of different time spans on the threshold. We then cluster the traces based on our threshold analysis and compare it with a baseline approach using multidimensional scaling and k-means.

Baseline and Metrics

As a baseline, after we generate the two distributions described previously, we apply multidimensional scaling (MDS) to the corrcoeff matrix, in order to transform the original high-dimensional relative space to a 3-D space with an absolute origin, and run the k-means clustering algorithm. We choose the true-positive rate (TPR, also known as recall rate) and false-positive rate (FPR) as metrics to evaluate the performance of our method versus the naive approach, which correlates the raw traces. A true-positive (TP) is when a sensor pair in a room is classified as being co-located while a false-positive (FP) is when a sensor that is not in room is classified as being so.

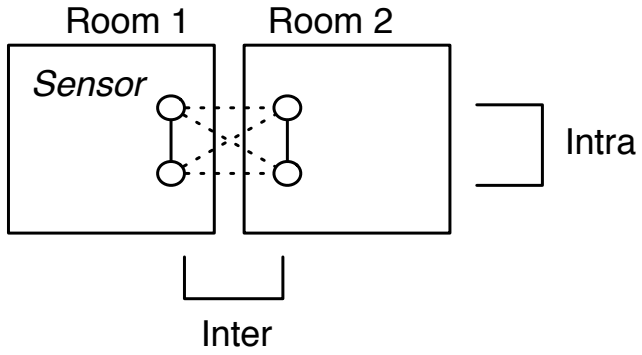


Figure 6.15: Two populations are examined for our threshold analysis. A solid line connects sensors in the same room while a dotted line connects to a pairs in different rooms.

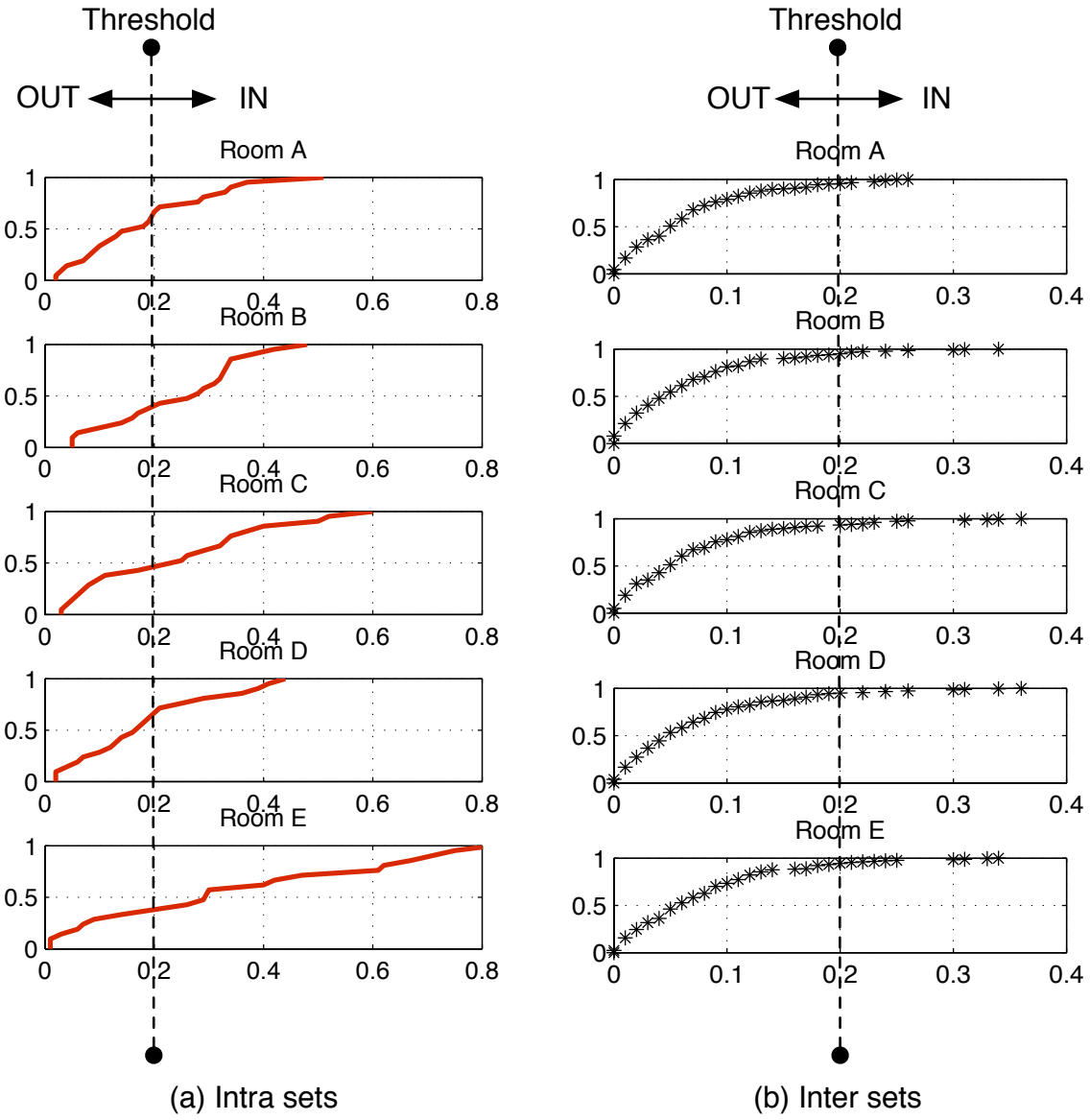


Figure 6.16: CDF of correlation coefficients between IMFs of sensor feeds: the dotted lines point to some threshold which divides the distribution and produces a TPR and FPR.

Characterizing the Boundary

To corroborate our boundary-existence hypothesis, we first need to characterize the boundary between sensors in different rooms. We compute the pairwise correlation coefficients (corrcoeffs) between sensor traces in both of populations depicted in Figure 6.15, over different time spans – ranging from one day to one month. After generating points over

different time spans for each room, we accumulate the corrcoeffs to obtain distributions as shown in Figure 6.16, for each of the five rooms.

The dashed vertical lines in Figure 6.16 represent an arbitrary threshold that partitions the distribution into two sets. Pairs of sensors to the right of the line are classified as being in the same room. Pairs of sensors to the left are classified as being in different rooms. The CDFs on the left column show the distribution of corrcoeffs for pairs known to be in the same room and the CDFs on the right show the distribution of corrcoeffs in different rooms. Note in the figure, we set the threshold to the same value to both the left and right side, in order to observe the effect of the true/false positive rates. By adjusting the threshold, we get different TPRs/FPRs parameterized by the threshold. Figure 6.10 captures the range tradeoff in a corresponding ROC curve.

Figure 6.10 illustrates the TPR/FPR sensitivity to different threshold values for our method and the naive approach. A good cluster achieves a high TPR and a low FPR. As we vary the threshold, we see that our approach achieves a TPR between 52%–93% and a FPR between 5%–59%. We can see that the average TPR for the ROC graph on the right is higher than the ROC graph on the left. Moreover, the corresponding average FPR is lower on the right than on the left. In general, as the TPR rises, the FPR also goes up – *a tradeoff exists between maximizing TPR and maintaining a lower FPR.*

The "boundary" is represented as the corrcoeff that produces a "good" TPR with an "acceptable" FPR. In Figure **??**, we choose 0.2 FPR as the boundary threshold. This point represents the largest difference between TPR and FPR – an acceptable tradeoff point. Looking at Figure 6.16, the 0.2 FPR corresponds roughly to the 80th-percentile correlation coefficient, on the "inter" set (the set of CDFs on the right). The recall rate for each room – using a 80th-percentile corrcoeff threshold value – ranges between 62%-86% and the threshold value falls into a narrow interval between 0.1 to 0.12. This shows that *we are able to choose a uniform value for all the rooms regardless of the sensor type.*

Convergence over Time

Using the threshold the roughly 80th-percentile corrcoeff corresponds to in the distribution, we examine how it affects the classification rate across traces that span different lengths of time. Convergence and consistency across different time spans is critical to automate the parameter selection process. Observe how the threshold values differ quite significantly in Figure 6.17. However, the threshold values gradually converge, as the length of training data increases from one day to one month. The values derived after 14 days of data are approximately the same as the final convergence value (around 0.07). In other words, we can determine a threshold from two weeks of data.

Clustering Results

We cluster the sensor traces over the entire one-month period, and use the roughly 80th percentile corrceff (0.07) as the boundary threshold. A sensor is classified into the cluster

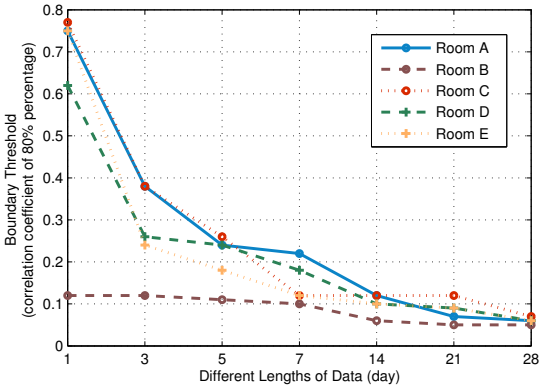


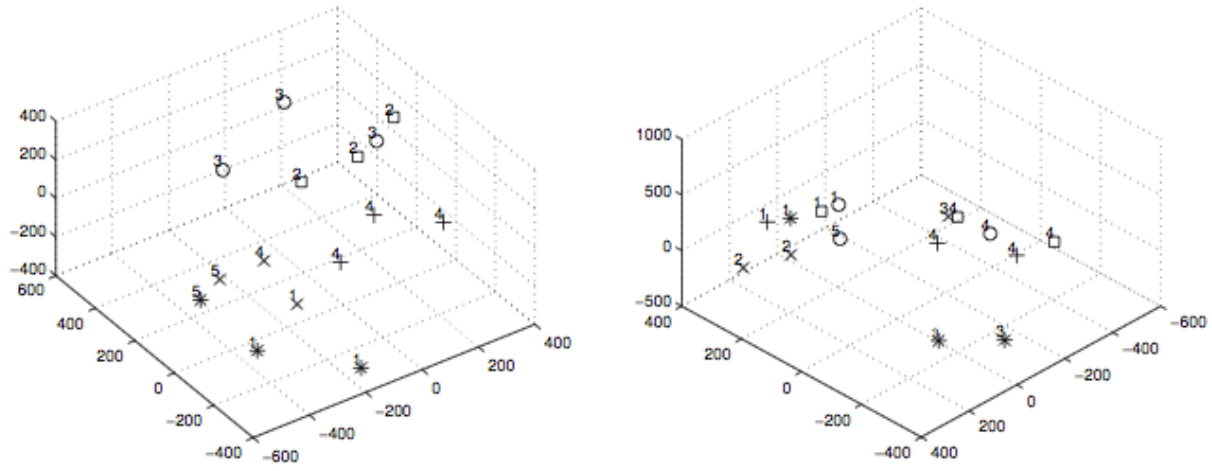
Figure 6.17: The threshold values all converge to a similar value and we can derive the optimal value with as minimal as 14 days data.

with the largest corrcoeff. The clustering result is shown in Table 6.4. A "1" means the sensor is classified as inside the corresponding room. In general, after obtaining the sensor clusters, we don't know which room each cluster corresponds to without further information such as the metadata of sensors. The labels "A-E" in Table 6.4 are used to indicate the ground truth of where each sensor is physically placed since we have such information. Overall, the classification accuracy is 93.3%. We do not cluster on the corrcoeffs obtained among raw signals because the 80%-percentile corrcoeff values do not converge across rooms. The reason that we are able to get such a high accuracy, which is seemingly different from the statistics in Figure 6.16 and Figure 6.10, is because the statistics in the two figures are generated out of the corrcoeffs accumulated over different time spans (the same intervals in Figure 6.17) while the clustering here is performed on the corrcoeffs from the entire one-month period.

To compare with our threshold-based method, we also cluster using a baseline approach. The pairwise corrcoeff for sensors in different rooms can be interpreted as a "distance" between them. A larger coefficient indicates a closer "distance", and vice versa. However, since the distances between pairs is relative, we use multidimensional scaling [23] to find a common basis in three dimensions, re-map the relative distance metric (feature vector) into this three-dimensional grid and use k-means to classify the traces. We set k to equal the number of rooms, since the goal of the approach is to verify spatial placement at room-level granularity. Generally, we believe that k should equal the number of rooms you wish to classify the sensors into. The clustering results are shown in Figure 6.18. Ground truth is shown through different markers (x, o, +, star, box). Each marker stands for one room. The cluster each sensor assigned to is denoted with a number. The classification accuracy of the baseline approach on corrcoeffs matrix of re-aggregated IMFs is 80%. For raw traces, the baseline approach achieves an accuracy of only 53.3%.

	A	B	C	D	E	
$Sensor A_1$	1	0	0	0	0	✓
A_2	1	0	0	0	0	✓
A_3	1	0	0	0	0	✓
B_1	0	1	0	0	0	✓
B_2	0	1	0	0	0	✓
B_3	0	1	0	0	0	✓
C_1	0	0	1	0	0	✓
C_2	0	0	1	0	0	✓
C_3	0	0	1	0	0	✓
D_1	0	0	0	1	0	✓
D_2	0	0	0	1	0	✓
D_3	0	0	1	0	0	×
E_1	0	0	0	0	1	✓
E_2	0	0	0	0	1	✓
E_3	0	0	0	0	1	✓

Table 6.4: Clustering result using the thresholding method: a "1" means the sensor is classified as inside the room. We get the "✓" and "×" by comparing the clustering results with ground truth.



(a) Clustering on corrcoeffs from our method.

(b) Clustering on corrcoeffs from the naive approach.

Figure 6.18: Clustering with k-means on the corrcoeff matrix after applying multidimensional scaling (MDS): The EMD-based set achieves an accuracy of 80% while the results with raw-trace is only 53.3% classification accuracy.

6.8 Categorical Verification Methodology

For categorical classification we took a use a very simple approach. For every trace, we partition the range into 10 bins and take the average. We sort the bins and take the top 2 and the combine it with the average. This combination forms our feature vector with three components.

We run our type analysis on 3 data sets from separate buildings. The first is a from the Univeristy of Tokyo. It contain 6 types of sensors measuring power, pressure, temperature, CO2, light, and occupancy. The nother is a deployment in Sutardja Dai Hall at UC Berkeley which measures 4 different type that include lumens, CO2, temperature, and humidity. Finally, we used a data set from Soda hall at UC Berkeley which contains 23 different types.

6.9 Categorical Verification Results

Categorical verification works quite well using the simple method describe in the the methdology. Table 6.5 gives a summary of a pair of large traces with few categories.

Building	**No. Sensors**	**No. Types**	**Accuracy**
Todai	400	3	89%
KETI	400	4	97%

Table 6.5: Categorical classification results for two data traces.

The simple methodology is able to separate them quite easily. It is clear that the mean and standard deviation provides enough inforomation for the classifier to differentiate between the different cateogories of traces.

The soda hall data traces were much more challenging to deal with. Soda hall was a much larger trace with 63 different categorical tags on the traces. Figure 6.19 shows the AGN vs AGO cateogories. Because the metadata for these is not available, we do not have any semantic information about the meaning of the tags. Note, there is a boundary between mean values 3 and 4.

Our classifier was able to correctly classify the types with relatively few samples, giving us over 99% classification accuracy in the range presented in the the graph. Similar results were obtained for separating the values presented in Figure 6.20. There is a clean boundary that is observable at around a mean value of 100. Every trace with a mean greater than 100 was labeled as a VR trace. The accuracy is 100%.

However, note the temperature traces between in Figure 6.21. The categorical clusters overlap signficantly. We examined this distribution more closely by construct a Gaussia mixture model and plotting the cluster centers. The results are presented in Figure 6.22.

Note, the cluster centers are not easily distinguishable. The classifier varied in accuracy, leading us to believe that these sets of traces were mostly indistinguishable. Moroever,

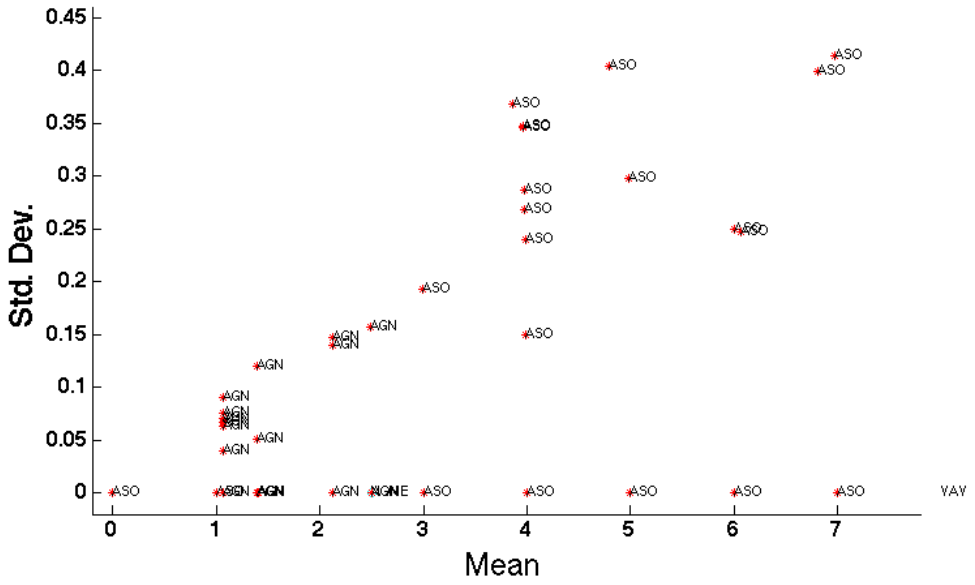


Figure 6.19: ASO versus AGN. There is a clear value-based boundary between the two sets of traces at around 3 in the mean.

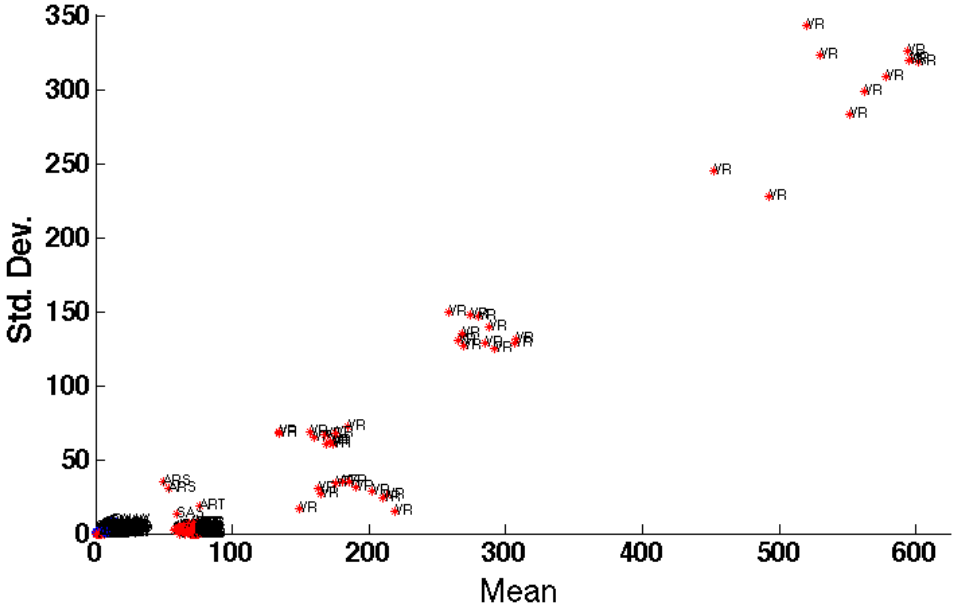


Figure 6.20: The VR traces span a wide range, however, any mean above 100 is a VR trace.

our approach for this trace is very tightly tailored to the particulars of this trace. More exploration is necessary.

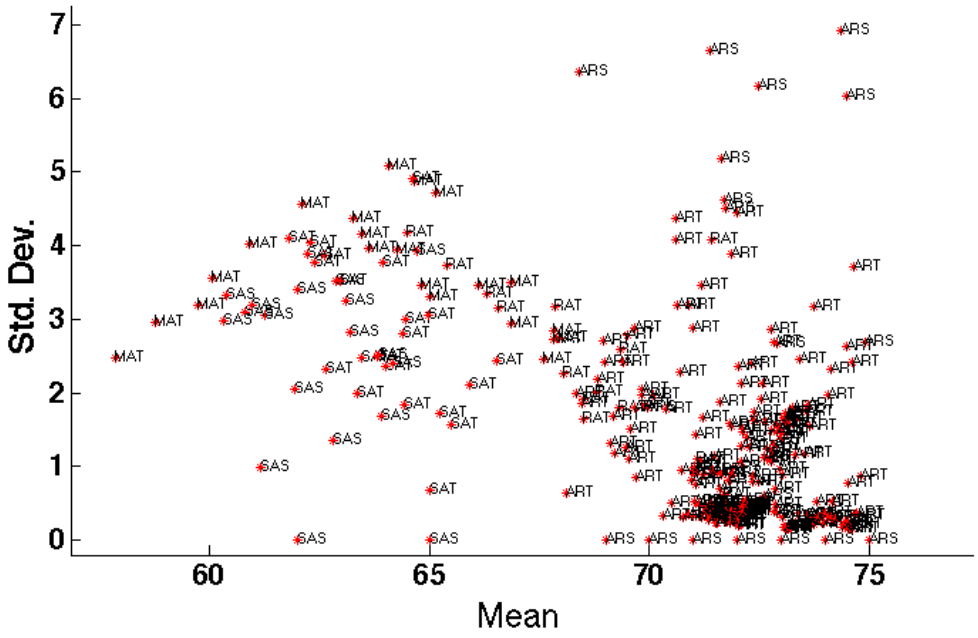


Figure 6.21: All temperature streams. Note, these are much more difficult to tease part. A Gaussian mixture model can separate them with approximately 77% accuracy, but it may not generalize.

6.10 Related Work

The research community has addressed the detection of abnormal energy-consumption in buildings in numerous ways [**katipamula:1review2005**, **katipamula:2review2005**].

The rule-based techniques rely on a priori knowledge, they assert the sustainability of a system by identifying a set of undesired behaviors. Using a hierarchical set of rules, Schein et al. propose a method to diagnose HVAC systems [**schein:hvacr2006**]. In comparison, state machine models take advantage of historical training data and domain knowledge to learn the states and transitions of a system. The transitions are based on measured stimuli identified through a domain expertise. State machines can model the operation of HVAC systems [**patnaik:toist2011**] and permit to predict or detect the abnormal behavior of HVAC's components [**bellala:buildsys2012**]. However, the deployment of these methods require expert knowledge and are mostly applied to HVAC systems.

In [**seem:energybldg2007**], the authors propose a simple unsupervised approach to monitor the average and peak daily consumption of a building and uncover outlier, nevertheless, the misbehaving devices are left unidentified.

Using regression analysis and weather variables the devices energy-consumption is predicted and abnormal usage is highlighted. The authors of [**brown:buildperf2012**] use ker-

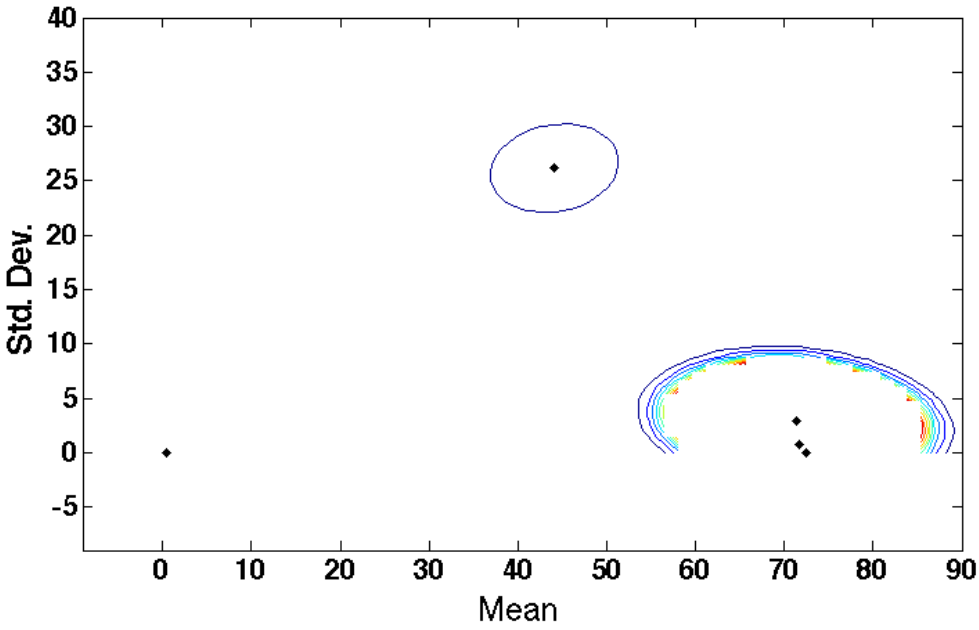


Figure 6.22: Centers for our Gaussian mixture model. Note, 3 of the 5 centers are very close to each other. This makes these traces very difficult tease apart and accurately classify.

nel regression to forecast device consumption and devices that behave differently from the predictions are reported as anomalous. Regression models are also used with performances indices to monitor the HVAC's components and identify inefficiencies [**zhou:wiley2009**]. The implementation of these approaches in real situations is difficult, since it requires a training dataset and non-trivial parameter tuning.

Similar to our approach, previous studies identify abnormal energy-consumption using frequency analysis and unsupervised anomaly detection methods. The device's consumption is decomposed using Fourier transform and outlier values are detected using clustering techniques [**wrinch:pes2012**, **chen:aaaiw2011**, 8]. However, these methods assume a constant periodicity in the data and this causes many false positives in alarm reporting. We do not make any assumption about the device usage schedule. We only observe and model device relationships.

Reducing a building's energy consumption has also received a lot of attention from the research community. The most promising techniques are based on occupancy model predictions as they ensure that empty rooms are not over conditioned needlessly. Room occupancy is usually monitored through sensor networks [**agarwal:ipsn2011**, **erickson:ipsn2011**] or the computer network traffic [47]. These approaches are highly effective for buildings that have rarely-occupied rooms (e.g. conference room) and studies show that such approaches can achieve up to 42% annual energy saving. SBS is fundamentally different from these approaches. SBS identifies the abnormal usage of any devices rather than optimizing the

normal usage of specific devices. Nevertheless, the two approaches are complementary and energy-efficient buildings should take advantage of the synergy between them.

Recently, there has been increased interest in minimizing building energy consumption. Our approach differs quite substantially from related work. Agarwal et al. [2] present a parameter-fitting approach for a Gaussian model to fit the parameters of an occupancy model to match the occupancy data with a small data set. The model is then used to drive HVAC settings to reduce energy consumption. We ignore occupancy entirely in our approach. It appears as a hidden factor in the correlation patterns we observe.

Kim et al. [47] use branch-level energy monitoring and IP traffic from user's PCs to determine the causal relationships between occupancy and energy use. Their approach is most similar to ours. Understanding how IP traffic, as a proxy for occupancy, correlates with energy use can help determine where inefficiencies may lie.

In each of these studies and others [29, 59], occupancy is used as a trigger that drives efficient resource-usage policies. Efficiency when unoccupied means shutting everything off and efficiency when a space is occupied means anything can be turned on. There is no question that this is an excellent way to identify savings opportunities, however, we take a fundamentally different approach. We are agnostic to the underlying cause or driver for efficient policies to be implemented. More generally, we look to understand *how the equipment is used in concert*. This may help uncover unexpected underlying relationships and can be used in an anomaly detection application to establish "(in)efficient", "(ab)normal" usage patterns. The latter should identify savings opportunities in cases where the space is unoccupied as well as occupied, because it has to do with the underlying behavior of the machines and how they generally work together. Our approach could help achieve both generality and scale for such an application. This article focuses on the first step of this application, the identification of correlated devices.

SBS is a practical method for mining device traces, uncovering hidden relationships and abnormal behavior. In this paper, we validate the efficacy of SBS using the sensor metadata (i.e. device types and location), however, these tags are not needed by SBS to uncover devices relationships. Furthermore, SBS requires no prior knowledge about the building and deploying our tool to other buildings requires no human intervention – neither extra sensors nor a training dataset is needed.

SBS is a best effort approach that takes advantage of all the existing building sensors. For example, our experiments revealed that SBS indirectly uncovers building occupancy through device use (e.g. the elevator in the Building 2). The proposed method would benefit from existing sensors that monitor room occupancy as well (e.g. those deployed in [**agarwal:ipsn2011**, **erickson:ipsn2011**]). Savings opportunities are also observable with a minimum of 2 monitored devices and building energy consumption can be better understood after using SBS.

SBS constructs a model for normal inter-device behavior by looking at the usage patterns over time, thus, we run the risk that a device that constantly misbehaves is labeled as normal. Nevertheless, building operators are able to quickly identify such perpetual anomalies by validating the clusters of correlated devices uncovered by SBS. The inspection of these

clusters is effortless compare to the investigation of the numerous raw traces. Although this kind of scenario is possible it was not observed in our experiments.

In this paper, we analyze only the data at medium frequencies, however, we observe that data at the high frequencies and residual data (Figure 6.6) also permits us to determine the device type. This information is valuable to automatically retrieve and validate device labels – a major challenge in building metadata management.

There has been much research work on sensor stream clustering and trace analysis. Chen and Tu [13] investigate how to cluster data streams in real-time using a density-based approach with a two-tiered framework. The first tier captures the dynamics of a data stream with a density decaying technique and then maps it to a grid. The second tier computes a grid density based on how it clusters the grid. Their approach differs from ours in that they focus on decreasing algorithm complexity for real-time sensor stream clustering. We run our analysis on historical traces and use correlation analysis in our clustering algorithm.

Kapitanova et al. [46] describe a technique to monitor sensor operations in the home and identify sensor failures. The classifier is trained on historical sensor data to obtain the relationship between sensors, assuming the number and location of sensors is known. When a failure or removal of a sensor occurs, the classifier's behavior deviates and the event is captured. Our method does not require any prior knowledge and instead tries to cluster feeds to discover their relative placement.

Lu and Whitehouse [52] formulate a new algorithm, particularly leveraging the semantic constraints interpreted from sensor data to determine sensor locations. The algorithm identifies how many rooms are present using motion sensors and determines room position based on physical constraints. Finally, it maps each sensor into the associated room. Our efforts focus on using intrinsic patterns typically pre-existing in building system sensor feeds to uncover physical relationships.

Fontugne et al. [27] propose a new method to decompose sensor signals with EMD. They extract the intrinsic usage pattern from the raw traces and show that sensors close to each other have higher intrinsic correlation. However, they do not explore the observation more deeply by answering whether there is a statistically discoverable boundary between sensor clusters in different rooms, or if there is a uniform threshold in the correlation coefficients able to be generalized to different rooms.

Fontugne et al. [28] carry on the work and propose an unsupervised method to monitor sensor behavior in buildings. They constructed a reference model out of the underlying pattens, obtained with EMD, and use it to compare future activity against it. They report an anomaly whenever a device deviates from the reference. This work exploits EMD as a method to detrend the signals and capture the inter-device relationships.

Much work utilizes EMD on medical data [5], speech analysis [38], image processing [61] and climate analysis [51]. Our method adopts EMD to determine whether a discoverable statistical boundary exists in sensors traces from sensors in different rooms and whether such a boundary can be generalized across rooms with various kinds of sensors.

6.11 Summary

This chapter aims to establish a set of methodology for classifying traces and verifying that relationships specified by users are accurate and continue to stay accurate over time. We present empirical techniques to identify abnormalities in device power traces and inter-device usage patterns.

We proposed an unsupervised method to systematically detect abnormal energy consumption in buildings: the Strip, Bind, and Search (SBS) method. SBS uncovers inter-device usage patterns by striping dominant trends off the devices energy-consumption trace. Then, it monitors device usage and reports devices that deviate from the norm. Our main contribution is to develop an unsupervised technique to uncover the true inter-device relationships that are hidden by noise and dominant trends inherent to the sensor data. SBS is used on two sets of traces captured from two buildings with fundamentally different infrastructures. The abnormal consumption identified in these two buildings are mainly energy waste. The most important one is an instance of a competing heater and cooler that caused the heater to waste around 2500 kWh.

EMD allows us to effectively identify fundamental relationships between sensor traces. We believe that identifying meaningful usage-correlation patterns can help reduce oversights by the occupants and faults that lead to energy waste. A direct application of this is the identification of simultaneous heating and cooling [60]. Simultaneous heating and cooling is when the heating and cooling system either compete with one another or compete with the incoming air from outside. If their combined usage is incorrect, there is major energy waste. This problem is notoriously difficult to identify, since the occupants do not notice changes in temperature and building management systems do not perform cross-signal comparisons. For future work, we intend to run our analysis on the set of sensors that will allow us to identify this problem: the outside temperature sensors, the cooling coil temperature, and the air vent position sensor. If their behavior is not correlated as expected, an alarm will be raised.

We can also apply it to other usage scenarios. In our traces, we found an instance where the pump was on but the lights were off; where, typically, they are active simulatenously. The air conditioning was pumping cool air into a room without occupants. With our approach this could have been identified and corrected. In future work, we intend to package our solution to serve these kinds of applications.

This chapter we also set out to examine the underlying relationship between sensor traces to find interesting correlations in use. We used data from a large deployment of sensors in a building and found that direct correlation analysis on the raw traces was not discriminatory enough to find interesting relationships. Upon closer inspection, we noticed that the underlying trend was dominating the correlation calculation. In order to extract meaningful behavior this trend has to be removed. We show that empirical mode decomposition is a helpful analytical tool for detrending non-linear, non-stationary data; inherent attributes contained in our traces.

We ran our correlation analysis across IMFs, extracted from each trace by the EMD

process, and found that the pump and light that serve the same room were highly correlated, while the the other pump was not correlated to either. In order to corroborate the applicability of our approach, we compared the pump trace with *all* 674 sensor traces and found a strong correlation between the relative spatial position of the sensors and their IMF correlations. The most highly-correlated IMFs were serving the same area in the building. As we relax the admittance criteria we find that the spatial correlation expands radially from the main location served by the reference trace.

We plan to examine the use of this method in applications that help discover changes in underlying relationships over time in order to identify opportunities for energy savings in buildings. We will use it to build inter-device correlation models and use these models to establish "(ab)normal" usage patterns. We hope to take it a step further and include a supervised learning approach to distinguish between "(in)efficient" usage patterns as well.

From the results illustrated in Figure 6.16, we observe a bi-modality in the corrcoeff distribution for the two population sets. Sensors in the same room correlate to each other more (typically a corcoeff of 0.4 or higher) than sensors in different rooms. This bi-modal distribution may provide insight for us to understand the boundary and search for an effective discriminator more broadly.

To further validate the effectiveness of the proposed method, we should consider using data from different sources. For example, in room B in Sutardja Dai Hall, there are two different sets of temperature sensors reporting data at different rates and granularities. We demonstrate our ability to classify sensor streams on the same platform (recall the sensor box we used to collect data). It would be more convincing to verify the effectiveness of our method with sensor streams generated from devices on different systems – since separate systems are independent. For instance, we can use temperature data from the second deployment and use the CO_2 and humidity sensor data from the first deployment and compare the results to what we have gathered.

In our results, the boundary threshold parameter converges to a narrow interval, as the data set expands over a longer time range. This may suggest that our method generalizes across rooms in a building, although further validation in a larger, more representative data set is necessary. This study looked at 5 different rooms with a large physical separation from one another. A more representative data set would consider all the rooms and pay special attention to rooms that share a common orientation and are separated by a single wall or floor slab.

We conjecture that local activity modulates various types of physical signals – captured by the various kinds of physical sensors embedded throughout the building – and that those signals are attenuated over distance and physical boundaries (such as walls). We believe that this is what drives our observations. If the conjecture is true, the effects will be less pronounced in larger rooms, such as an auditorium or a large laboratory space.

As our approach performs slightly better than traditional learning techniques, we must further evaluate its robustness versus the baseline method; across the entire building and across multiple buildings. In future work, we will examine the two approaches across larger intra-building data sets and compare results across multiple buildings. A key factor is the

variance of classification accuracy – smaller variance demonstrates robustness.

We present a new method for spatial placement clustering. We first characterize the corrcoef distribution of medium frequencies IMFs between sensors in the same/different room(s), and then we learn the tradeoff between achieving a higher TPR and maintaining a lower FPR by manipulating a discriminator parameter within these two distributions. For a preliminary sample of relatively well separated rooms, we find that there is a clear boundary between sensor clusters in terms of their spatial placement and the boundary can be probed statistically. We also find a uniform discriminator can be learned and generalized across these rooms. For this initial study, our method is able to classify the sensors of 93.3% accuracy, which is 13% higher than a tradition k-means approach, with a TPR between 62%-86% and a FPR less than 20%.

These results are very encouraging. However, we recognize that they are far from definitive. While the rooms in the study were picked arbitrarily, they are neither comprehensive nor a systematic sampling. While they are clearly separated by our approach, and not by analyses of the raw time series, they do differ substantially in placement and usage. A key question going forward is, "how well will highly similar rooms be separated?" - say, adjacent rooms facing the same side of the building and with similar occupancy. Will these techniques hold, more powerful techniques be required, or is further discrimination intractable? In future work, we will examine how far this method takes us and explore how it may be used in combination with other techniques to improve the results more generally. Automated metadata verification is important to include in the lifecycle of building data management.

We also attempt to address the categorical classification problem. With fairly simple approaches we can use the mean and standard deviation of the trace to classify the category of the trace, as labled by the user. However, for large traces with many overlapping categories we observed that the traces are very similar and cannot be distinguised. In order to uncover we may need out-of-band information. Statistically they are indistinguishable with the techniques we present.

Chapter 7

Lessons Learned and Future Work

Our experience with the design, imeplementation, and wide-spread deployment of StreamFS teaches us several lessons about the mechanisms necessary to "app-ify" the building and the value of opening up the building as a platform for application development in the interest of 1) reducing energy consumption and increased operation efficiency, 2) providing deeper insights about the operational dynamics of the building, 3) enabling the building to participate in a broader software ecosystem in the interest of more intelligent use of resources.

7.1 Lesson Learned

In this section we discuss the lesson learned from deploying our system. We present 5 statements that we experienced and find true moving forward, all based on our deployment experience and through our many interactions with building systems, building managers, and the broader building science software ecosystem.

Data Services Fundamental for Building Applications

Fundamentally application in the building must deal with data coming from a distributed, diverse deployment of sensors taking physical measurements. The vast majority of building applications are analytical in nature and as we start coupling streaming sensor data with actual building model and control, more data services and jobs will be used across several applications simultaneously.

We implemented and deployed 7 applications on top of StreamFS. Some applications were for instance management of the various components in StreamFS, however we built a viewer console application, which initiated time-series queries and displayed them to the user, simple control applications that triggered a cascading re-activation sequence of home appliances when total energy consumption was above a user-defined threshold, and the mobile Energy Lens application provides the user with aggregate statistics based on the spatial configuration of the deployment.

All these applications shared a small set of processing elements. One of the elements is for removing statistical outliers, another for interpolating values, one for computing the aggregate load curve, and another to compute a moving average of consumption. Because StreamFS provides the ability to define the function once, it could be used across applications. The hierarchical model prediction application presented in Chapter 2 would also require similar *clean* and *aggregate* capabiltiies before the data is fed into the model. For jobs that require any prediction or machine learning, the same assertion holds.

Analytics is Dependent on Inter-relationships

In StreamFS and the associated applications we demonstrate the value of coupling the inter-relationships explcitly with our aggregation jobs. Many jobs require the streams be fetched in the context of the metadata of the deployment and what the metadata captures about the placement or category of the stream. We find that it is convenient to consider coupling these associations explicitly, since many applications require it. We found the use of *aggregation points* to be a convenient tools for analysis, both from a historical perspective and from a live feed perspective.

Moreover, queries for determining which sensors are where and how many sensors there are in a particular space are crucial for achieving generality across buildings – a property that is rare to find in *any* aspect of building design and operation. Providing the ability to traverse the relationship structure and discover, through the structure, the relationship between the sensors allows applications from one building to potentially be dropped into a new building without much learning. With this kind of capabilty, we could write applications for one building and port them across the building stock; allowing a solution to have true impact on the entire building stock.

Centralized Management is better than Distributed Management

StreamFS adopts the Unix philosophy where everything is represented as a file. This makes it simple to manage the raw and application-specific derivative data. It also simplifies the sharing of processing code across applications. In addition, access control can be provided from a centralized location. Application writers can devide which streams and processes they want to share from the individual resource level to groups of resources organized as a collection within a container file. Generally, when there are many disparate devices that make up a system, there is a tension between extensibility and ease of management. Large deployment typically enable protocols for joining the network, but not ones that are easy to manage. The ones that are easy to manage, typically do not handle other kinds of sensors or devices easily joining the deployment.

When the deployment is presented as a distributed system to the end user, there are usually complex mechanisms in place for doing discovery. Disovery over a centralized management system is much easier since it is clear where to target the search. Typically the argument against centralized management is a single point of failure, however the web-services

model deals with centralized services rather well. The systems is presented in a unified fashion to applications, while internally it is actually distributed; shared for load management and replicated for failure tolerance. StreamFS follows these principles throughout the design of each of its components.

Similar to the proposal by Dixon et al. [22], we believe that the abstraction should be raised from the network to the operating system level. The Unix philosphy allows us to present unified namespace and access to the building deployment information and various levels granularity. This allowed us to manage many applications simulataneously, control information sharing between them, and provide a unified view to the application writer; making it clear what resources are available to her. We believe these centralized view is the right one. Although StreamFS is a distributed system in the cloud, it present a unified layer for access and control which makes application development and management simpler.

Accurate Capture of Physical Configuration is Crucial

As we move towards software-defined building infrastructure and more analytical and control applications rely on building state to make detailed control decision, it becomes very important to accurately capture the physical configuration of the building and provide mechanisms for discovering inconsistencies between the virtual representation and the corrispinding physical state. There is a lot of work in the vision community for constructing a virtual representation of the physical world, accurately. In our work we show ways the the data *already being collected from the building sensor deployments* can be used to ascertain physical relationships between sensors. We introduced three kinds of verification – functional, spatial, and cateogrical – however there are other ways we could explore these physical relationships. For example, we can combine these technique with vision-related work which uses cameras to determine the physical layout of a space and combine it to use physical models of heat transfer to determine if the readings we are collecting are accurate.

Ultimately these should inform the layer presented to the application about the accuracy of the deployment information. We explained how inaccuracies leads to errors in aggregation and control. For software interfaces for the built environment to become more widespread, we must solve these issues related to verification.

OS Abstractions for Managing Truly Physical Resources

Mapping the jobs of organizing and controlling access to physical resources, the use of operating system abstractions allows us to reason about how to componentize a management architecture and lets us frame how solutions can be constructed as applications that make use of the primitives provided by the operating system components. In our work we presented a filesystem abstraction which adopted several data and management services for organizing the information in the building. Related work in the home [22] and in buildings [18] take a similar approach.

Questions remain about how building models fit into a operating system services architecture. Perhaps they can be included with the verification services. Ideally, we can start moving towards automated plug-and-play building applications with guarantees for service quality and efficient management of physical resources. We believe that the best way to make this a reality it to view the building as a hardware platform and to move towards a truly distributed operating system for managing the systems, devices, and applications that run on the building platform.

7.2 Future Work

We explored many aspect of a design for building information information systems. However, there are still many open questions have have not been tackled in this work. We discuss three main future or on-going project topics in this section.

Explore More Diverse Verification Methods

There's a lot of work in capturing the physical state of the environment and building a virtual representation of it. Althought SBS shows a lot of promise, in terms of its effectiveness and generality, spatial verification shows poor generality under certain conditions and categorical classification seems to only work with small data sets. We look to expand our exploration in the two pieces of work that showed fractional success. We must characterize the conditions for which these verification approaches work well and formulate algorithms to detect whether those conditions hold in the data prior to initiating those verification processes. We can partition the characterization to discover the statistical or semantic properties or pieces of information that must be known about the deployment beforehand and tackle those problems only.

For the others, we must explore different techniques that generalize better. Although it is valuable to solve the problem for a small set of homogenous buildings, the real value comes through generalization, since in order to have widespread impact solutions must be brought to a large fraction of the building stock, quickly.

Deeper Exploration of Control Applications

We did not get a chance to support the kind of control application proposed in the beginning on the dissertation. It did not allow us to experiment with vairous kinds of actuator interfaces for integrating them more generally into the architecture. We intend to expand our control application work by implementing model-predictive control processes on building deployments that use StreamFS. We also need to closely examine a diverse set of control interfaces and APIs in order to generalize the control interface exposed through StreamFS.

We only explored the class of controllers that consume binary signals. Specifically, we intergrated with the ACme [**acme**] wireless power meter to turn devices on and off, remotely. There are other kinds of controllers, that do not accept binary input. They input variable controllers, set-point driven controllers, parameterized controllers, time-based controllers, etc. For example, controllers based on physical models may used physical configuration-based parameters to determine how to drive the load for the system controlling the space. These must be considered and applications should be explored in this context before a determination can be made.

Version Control For Buildings

Provenance checking is important in many systems. The building has many actors interacting with it and distributed changes cause the once efficient configurations to slowly deviate back to an inefficient state. Version control would allow us to track changes in the deployment and associated configuration decisions. It also introduces the notion of state rollback, whereby we roll the associated state information back to a previous, safe, efficient state. We would like to explore how rollbacks manifest themselves in the physical environment, since it involves not just rolling back the settings, register values, processes, etc. but the instant rollback operation affects how the physical resources in the environment are activated. Also, certain conditions *cannot* be rolled back, such as the weather conditions, so a rollback operation needs to check if the proper rollback conditions are in place before the rollback is committed and executed.

In addition, how do we determine commit conflicts between two committers. In a traditional version control system it is based on the actual data being written to the file. In the building context, conflict must be determined by models. Model based on first-principles of the underlying physics or statistical models that used several empirically-derived parameters to project the state of the system at some point in time. For example, if two commits are made to change the setpoint of a thermometer, how to we determine that the setpoints will conflict and how to we resolve them. Both commits are writing to the same devices, so that cannot be the sole criteria. Any approach must consider how the set point affects the state of the room temepature that is controlled by that setting or the behavior of the air-handling unit when commits are happening quickly. Either involves the use of a model to determine what the correct behavior is and project whether the new set point will lead to the right behavior. We look to explore these and other related concept further and hope that it leads to smart, software-defined buildings.

Chapter 8

Conclusion

This thesis examines the state of the art of building information systems and evaluates their architecture in the context of emerging technologies and applications for deep analysis of the built environment. With the increasing interest in attaining a deeper understanding of the operational dynamics of buildings and an increased interest in energy efficiency, we assert that the only way to enable the wide spread of solutions across the entire stock of buildings is to build a platform that "app-ifies" the built environment. We observe that modern building information systems hinder wide spread development of building applications. They are difficult to extend, do not provide general services for application development, do not scale, and are difficult to set up and manage.

We propose a new architecture that embodies these system principles through a filesystem abstraction and data services. We decompose all deployment data and metadata into three types of data – structural inter-relationships and naming, attribute-value descriptive pairs, and timeseries data – and expose them through a filesystem abstraction. We adopt the Unix philosophy that "everything is a file" in a system called StreamFS. We introduce 4 types of files – containers, streams, controllers, and special – that can be aribtrarily composed to reflect information about the physical configuration of the sensors and actuators relative to the systems and spaces in the building. The filesystem makes these physical resources directly accessible, in a centralized and controlled fashion. We also adopt filesystem security that mediates access between the client-side application and the sensors in the deployment. Because containers can be used to group files together, the security layer can also explicitly mediate access to *collections* of files that represent physical resources.

StreamFS also provides a number of data services, made available to applications through the pipe abstraction. Applications can designate a set streams to flow through user-defined *process elements*. The output of process elements are made available through the filesystem as stream file themselves, allowing applications to construct complex processing pipe-lines. We also leverage the relationship structure to inform common processing pipelines to support *OLAP-style* aggregation queries. We describe how the underyling *entity-relationship graph (ERG)* is used to support the notion of *aggregation points*. We show this graphical structure and associated mechanisms map to a traditional OLAP cube. Portions of the hierarchy that

have enabled aggregation points can then support *drill-up/drill-down*, *pivot*, and *slice and dice* queries.

We deploy StreamFS in 7 different buildings with very different setting and wrote several applications for it as well. One of the driving application is the Mobile Energy Lens. The Energy Lens app provides occupants with mechanisms for collecting building information in a unified platforms and provides a way to view aggregate energy consumption associated with the spatial deployment of plug-load devices. We present a 3-layer application architecture, where one of the main layers is implemented entirely through StreamFS data management and data processing services. We also discuss the challenges that have to be overcome in order to provide a better user experience and discuss how each of mechanisms used to solve those problems were also implemented on top of StreamFS.

Finally, we introduce the notion of verification of physical relationship through empiricial data. We characterize and motivate the problem in buildings as being a fundamental problem that needs to be solved as we move toward software-based control of the built environment. We partition the verification problem into three sub problems: 1) functional verification, 2) spatial verification, and 3) categorical verification. We should how empirical mode decomposition, correlation, and simple machine learning techniques can give us information about how the sensors are related to each other, physically. We propose the use of this information to verfiying the manually specified physical configuration setting.

Through our deployments we demonstrate the importance of specific features and overall, we demonstate an *extensible, generalizable, scalable, and easy-to-manage* system for supporting the "appification" of the built environment. There are still many questions to explore more deeply, but we believe that StreamFS and verification provide a solid foundation on which to continue our exploration.

Appendix A

StreamFS Process Code

Listing A.1: Load curve code used to generate aggregate load curves in the Energy Lens application.

```
function(buffer, state){
    var outObj = new Object();
    var timestamps = new Object();
    outObj.msg = 'processed';
    if(typeof state.slope == 'undefined'){
        state.slope = function(p1, p2){
            if(typeof p1 != 'undefined' && typeof p2 != 'undefined' &&
                typeof p1.value != 'undefined' && typeof p1.ts != 'undefined' &&
                typeof p2.value != 'undefined' && typeof p2.ts != 'undefined'){
                if(p1.ts == p2.ts)
                    return 'inf';
                return (p2.value-p1.value)/(p2.ts-p1.ts);
            }
            return 'error:undefined data point parameter';
        };
        state.intercept = function(slope,p1){
            if(typeof p1 != 'undefined' &&
                typeof p1.value != 'undefined' && typeof p1.ts != 'undefined'){
                return p1.value - (slope*p1.ts);
            }
            return 'error:undefined data point parameter';
        };
    }
    if(typeof state.multibuf == 'undefined'){
        state.multibuf = new Object();
    }
    outObj.inputs = new Array();
    var noted = new Object();
```

```
for(i=0; i<buffer.length; i++){
    var streamid = buffer[i].pubid;
    var ts = buffer[i].ts;
    if(typeof state.multibuf[streamid] == 'undefined'){
        state.multibuf[streamid] = new Array();
    }
    state.multibuf[streamid].push({'ts':buffer[i].ts,
        'value':buffer[i].value,'path':buffer[i].is4_uri});
    if(typeof noted[buffer[i].is4_uri] == 'undefined'){
        noted[buffer[i].is4_uri]=true;
        outObj.inputs.push(buffer[i].is4_uri);
    }
    timestamps[ts] = true;
}
var streamids = Object.keys(state.multibuf);
var tss = Object.keys(timestamps);
tss = tss.sort();

var ts_per_stream = new Object();
if(streamids.length>=2){
    for(j=0; j<streamids.length; j++){
        var this_streamid = streamids[j];
        var dpts = state.multibuf[this_streamid];
        if(dpts.length<2){
            outObj.stat = 'pending';
            return outObj;
        } else {
            for(dpidx = 0; dpidx<dpts.length; dpidx ++){
                if(typeof ts_per_stream[this_streamid] == 'undefined'){
                    ts_per_stream[this_streamid] = new Object();
                }
                var thists = dpts[dpidx].ts;
                ts_per_stream[this_streamid][thists]=true;
            }
        }
    }

    var cleaned = new Object();
    for(j=0; j<streamids.length; j++){
        var this_streamid = streamids[j];
        var dpts = state.multibuf[this_streamid];
        cleaned[this_streamid]=new Array();
        for(tss_idx = 0; tss_idx < tss.length; tss_idx++){
            var timestamp = tss[tss_idx];
```

```
                if(typeof ts_per_stream[this_streamid][timestamp] == 'undefined'){
                    var p1 = dpts[0];
                    var p2 = dpts[dpts.length-1];
                    var slope = state.slope(p1,p2);
                    if(slope != 'inf' || slope.indexOf('error:')<0){
                        var intercept = state.intercept(slope,p1);
                        var newdpt = new Object();
                        newdpt.ts = timestamp;
                        newdpt.value = (slope*timestamp)+intercept;
                        cleaned[this_streamid].push(newdpt);
                    } else {
                        outObj.slope=slope;
                    }
                } else {
                    for(idx = 0; idx<dpts.length; idx++){
                        if(dpts[idx].ts==timestamp){
                            cleaned[this_streamid].push(dpts[idx]);
                            break;
                        }
                    }
                }
            }
        }

        var loadcurve = new Array();
        var cleaned_keys = Object.keys(cleaned);
        var pts_per_key = cleaned[cleaned_keys[0]].length;
        for(ts_idx=0; ts_idx<tss.length; ts_idx++){
            var sum = 0;
            for(idx=0; idx<cleaned_keys.length; idx++){
                var thissubid = cleaned_keys[idx];
                sum += cleaned[thissubid][ts_idx].value;
            }
            loadcurve.push({'ts':cleaned[thissubid][ts_idx].ts,
                            'value':sum});
        }
        outObj.data = loadcurve;
    } else {
        outObj.stat = 'pending';
    }
    buffer = new Array();
    return outObj;
}
```

Appendix B

StreamFS HTTP/REST Tutorial

This tutorial is meant to help you get started quickly with StreamFS. StreamFS integrates all kinds of sensor data and organizes it for easy access, processing, and integration with external applications. In this tutorial we will go through creating and deleting files in StreamFS, as well as accessing stream data from incoming data streams.

- Creating a resource
- Creating a stream file
  - Streaming data through stream file
  - Bulk data insertion
- Bulk file creation
- Queries
- Subscriptions
- Symlinks
- Move
- Stream Processing
  - Create process file
  - Starting the process
  - View the output
  - Stopping the process

Term	Description	Examples
container/default file	An logical object that represents a physical or logical entity.	`/hvac/heater` refers to a heater in the hvac system
stream file	A file that represents a data stream. Data is pushed into the stream files and queried through the same file path..	`/room1/temperature/` is the name for a temperature stream coming from room1.
control file	A file that represents a control channel for an associated actuator.	`/hvac/heater/switch` refers to the switch for the heater. Writing a 1 to that file send an 'ON' signal to the heater, 0 is 'OFF'.
subscriber	An external target URL to which data is forwarded as it comes into StreamFS. Subscription are used to process incoming data in real time in external applications.	`/subs/550e8400` represents a subscription file created after a subscription request satisfied. The Id is a unique id for the subscriber. It can be used to manage the subscription – to see which streams are pushing data to which URLs and its deletion removes the subscription.

Table B.1: Terminology

B.1 Terminology

B.2 Creating a resource

A clean installation of StreamFS, it comes with a set of core resources described in StreamFS documentation. The resource to start with is /. To make sure that StreamFS is up and running do a GET on that resource. You should receive a reply with some information about the instance as well as child resource for that resource. Observe the example below:

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Thu, 19 Jan 2012 12:44:54 GMT

{
    "status": "success",
    "children": [
```

```
        "ibus",
        "sub",
        "resync",
        "pub",
        "time",
        "models",
        "admin",
    ],
    "uptime": 890,
    "uptime_units": "seconds",
    "activeResources": 37
}
```

Now lets create a simple file that we'll use as our working directory for the tutorial. Create a temporary directory where you want to save the file you'll create for this tutorial. Open a text file and copy-paste this json below (not the PUT line) into the file. Then use curl (on linux systems) to POST the document to the StreamFS server.

```
echo "{\"operation\":\"create_resource\", \
\"resourceName\":\"temp\",\"resourceType\":\"default\"}" \
 > create_def.json
```

This creates a text file with json in it that specifies the type of resource file you want to create and what its name is.

```
curl -i -X PUT "http://localhost:8080/" -d@create_def.json
```

The -d parameter in curl specifies the data portion of the PUT request. In this case we're forwarding the data to the root path. Once created, we issue an HTTP PUT request to send the request to the root directory of the StreamFS instance that is running. If created successfully you should get a reply that looks like the following:

```
HTTP/1.1 201 Created
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Thu, 19 Jan 2012 12:35:25 GMT
Notice that it's a '201 Created' HTTP status. That means the StreamFS was able
    to create a default resource for you under /. Now that's check to make sure
    it's there.

curl -i "http://localhost:8080/"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
```

```
Connection: close
Date: Thu, 19 Jan 2012 12:44:54 GMT

{
    "status": "success",
    "children": [
        "ibus",
        "sub",
        "resync",
        "pub",
        "time",
        "models",
        "admin",
        "temp",
    ],
    "uptime": 990,
    "uptime_units": "seconds",
    "activeResources":
}
```

Notice that when we issue the same GET request to the root directory in StreamFS we see "temp" in the children array associated with /. Now we can treat the "temp" resource as a directory and work there (i.e. all requests for resource creation, deletion, etc, will be forwarded to that file).

```
curl -i "http://localhost:8080/temp"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Thu, 19 Jan 2012 12:50:12 GMT


{
    "status": "success",
    "type": "DEFAULT",
    "properties": {},
    "children": []
}
```

As we populate this directory, the name of the newly created files will show up the "children" array. Now, lets create a stream file that represents a real-time data stream. We'll create it as a child of the "temp" folder.

B.3 Creating a stream file

This processes is very similar to create a regular file. Lets create another json file with the command to create a stream file.

```
echo "{\"operation\":\"create_generic_publisher\", \
\"resourceName\":\"stream1\"}" > create_stream.json
```

The operation that we're running this time is to creata a generic publisher. In StreamFS, that's understood to mean a stream file. Once created, lets post it to StreamFS, but this time lets post it to the directory we created in the previous section:

```
curl -i -X PUT "http://localhost:8080/temp" -d@create_stream.json
HTTP/1.1 201 Created
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Thu, 19 Jan 2012 12:58:31 GMT

{
    "status": "success",
    "is4_uri": "/temp/stream1",
    "PubId": "789cf943-bbc8-428e-97ce-03e7cfe5fc12"
}
```

In reply above is shown. You should receive explicit confirmation from StreamFS that the stream file was created and should note the associated publisher ID. The publisher identifier is used when data is pushed to this resource. The underlying stream uses it when it POSTs data to this new file.

B.4 Pushing data to a stream file

In order to push data to the stream file, we have to use the pubid associated with the stream. We can either copy-paste it was from the response we received when we created the file, or we can simply call a GET on the file in StreamFS to obtain it:

```
curl -i "http://localhost:8080/temp/stream1"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Thu, 19 Jan 2012 13:31:55 GMT

{
```

```
    "status": "success",
    "pubid": "789cf943-bbc8-428e-97ce-03e7cfe5fc12",
    "head": {},
    "properties": {}
}
```

Now that we have it noted, lets create a fake data object. For simplicity, we leave out any complicated information other than the value that we wish to save. The units are omitted for now and i'll explain why at the end.

```
echo "{\"value\":123}" > datapt.json
```

Finally, we post the newly create file to the stream file as follows. To inform StreamFS to save the file correctly, we need to include the "type" and "pubid" as URL parameters. The type is always equal to "generic" and the pubid is set to the pubid we noted earlier. If either is missing or the pubid does not match the pubid associated with this stream, the POST will fail.

```
curl -i -X POST \
"http://localhost:8080/temp/stream1?type=generic&pubid=\
789cf943-bbc8-428e-97ce-03e7cfe5fc12" -d@datapt.json

HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Thu, 19 Jan 2012 13:35:25 GMT

{"status":"success"}
```

If successfully you should get the response above. This response can be used by the stream an an acknowledgement that the data has been succesfully saved. We can also check that by calling GET on the stream file again and seeing the "head" attribute. The "head" attribute is the last received data object saved and the timestamp associated with it.

```
curl -i "http://localhost:8080/temp/stream1"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Thu, 19 Jan 2012 13:36:23 GMT

{
    "status": "success",
    "pubid": "789cf943-bbc8-428e-97ce-03e7cfe5fc12",
```

```
    "head": {
        "value": 123,
        "ts": 1326980179
    },
    "properties": {}
}
```

Notice the "properties" attribute in the GET response to all resources. This is where the user can place arbitrary information about the object this file represents. For streams, however, the "units" attribute in the properties object is of particular importance, and we'll discuss it's importance later. For now, I'll show you how to update the properties. Recall from the creation of a fake data point that we did not specify the type of data (i.e. the units of measurement).

Lets create a simple json document with the "properties" attribute defined as a json object with the "units" attribute. Below, I create a properties object with "psi" (pressure) units.

```
echo "{\"operation\":\"overwrite_properties\",\
 \"properties\":{\"units\":\"psi\"}}" > overwrite_props.json
```

Then we POST it to the stream file.

```
curl -i -X POST \
"http://localhost:8080/temp/stream1" -d@overwrite_props.json
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Thu, 19 Jan 2012 22:52:57 GMT

{"status":"success"}
```

If successful, we should get the preceding reply and we can check to make sure that everything is set up ok.

```
curl -i "http://localhost:8080/temp/stream1"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Thu, 19 Jan 2012 22:53:07 GMT

{
    "status": "success",
    "pubid": "789cf943-bbc8-428e-97ce-03e7cfe5fc12",
```

```
    "head": {
        "value": 123,
        "ts": 1326980315
    },
    "properties": {
        "units": "psi"
    }
}
```

We can set any properties on the object, as long as it's a valid json object. The only attribute that's currently reserved on the properties object is "units". The fields added here are used for properties-related queries – they effectively serve as arbitrary tags on the files, so that we can find the specific ones later without traversing the entire tree.

B.5 Bulk data insertion

For efficiency, we can also push multiple values in a single request as shown below:

```
{
    "path":"/temp/stream1",
    "pubid":"789cf943-bbc8-428e-97ce-03e7cfe5fc12",
    "data":[{"value":0},{"value":1},{"value":2, "ts":1347307033198},{"value":3,
        "ts":1347307033199}]
}
```

Note, each element in the data array includes at least the 'value' field. The 'ts' field is optional. If not included, StreamFS will add it during processing.

B.6 Queries

The next section will show you how to run queries on the data and queries on the properties that are set on the files.

Timeseries

Finally, lets do a simple queries, starting with those that are timeseries in nature. We'll run a query that return any data point that was saved by this stream file in the last 10000 seconds. We'll go over the query syntax is the next section, but for simplicity, run the following:

```
curl -i "http://localhost:8080/temp/stream1?query=true&ts_timestamp=gt:now-10000"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
```

```
Date: Thu, 19 Jan 2012 13:52:39 GMT

{
    "path": "/temp/stream1/",
    "ts_query_results": [
        {
            "value": 123,
            "ts": 1326980315
        }
    ],
    "props_query_results": {
        "errors": [
            "Empty or invalid query"
        ]
    }
}
```

The "ts_query_results" is really what we care about here. It's an array of json objects, where each json object is the object that was saved the underlying database holding data for this stream. Notice, the timestamp is also included. We can obtain much larger results using this functionality. We'll go more in depth in the next few sections.

We're fundamentally dealing with timeseries data and there's a very simple syntax for acquiring the data you need. The following is a list of URL parameters that can be set to run a timeseries query.

Parameter	**Description**
query	Must be set to "true" for the query to be processed.
ts_timestamp	A logical object that represents a physical or logical entity.
Sub-parameters	
gt	greater than.
lt	less than.
gte	greater than or equal to.
lte	less than or equal to.
now	time on streamfs server when query is submitted.

Table B.2: Parameters

Sub-parameter gt greater than. lt less than. gte greater than or equal to. lte less than or equal to. now time on streamfs server when query is submitted.

The parameters specified above are URL parameters that should be included in the GET request URL to StreamFS. Notice that the "query" parameter must be set to "true" in order

for the query to be processed. Below we have included several query examples.

```
1. curl -i
   "http://localhost:8080/temp/stream1?query=true&ts_timestamp=lt:1327017501"
2. curl -i "http://localhost:8080/temp/stream1?query=true&ts_timestamp=lte:now+1"
3. curl -i
   "http://localhost:8080/temp/stream1?query=true&ts_timestamp=gte:1326980040,lt:1326980315"
```

The first query fetches all values less than 1327017501. The second query fetches all values less than or equal to "now+1". The keyword "now" is used by streamfs to be the current time on the streamfs server when the query is submitted and can be used as a variable in the query. The third query is a typical range query, where we want values greater than or equal to 1326980040 and less than 1326980315. Notice the use of the comma in the URL parameter value. The comma implies an "AND" condition for the timeseries query.

Properties

Querying properties is a complex and you have more options. There's essentially two ways. The first sets the "props_" URL parameter while the other submits a set of keywords. The latter is the one we'll go over in this tutorial.

```
echo "{\"\$or\":[{\"_keywords\":\"units\"}]}" >props_query.json
curl -i -X POST "http://localhost:8080/*?query=true" -d@props_query.json
```

B.7 Bulk default/stream file creation

Creating each file at a time can incur high overhead when there are many files to create, mainly because each request is established over a new HTTP connection. Therefore, StreamFS support a bulk-creation request:

```
{
    "operation":"create_resources",
    "list":[
            {
                "path":"/temp/one/two/stream3",
                "type":"stream"
            },
            {
                "path":"/temp/one/three/stream4",
                "type":"stream"
            },
            {
                "path":"/temp/one_four/two/",
                "type":"default"
            }
```

```
    ]
}
```

The operation name is 'create_resources' and is includes a list array where each element in the list if an object with the path and type attributes set. The path is the path of the resource you want to create and the type is it's type. For each path in the list, it will create the necessary files that eventually lead to the creation of the file listed. For example, if only `/temp` exists, `/temp/one` and `/temp/two` will be created as "default" files and `/temp/one/two/stream3` will finally be created as a stream file.

The response is shown below:

```
HTTP/1.1 201 OK
Content-Type: application/json
Connection: close
Last-Modified: Sun, 09 Sep 2012 21:14:46 GMT
Server: StreamFS/2.0 (Simple 4.0)
Date: Sun, 09 Sep 2012 21:14:46 GMT

{
    "/temp/one": {},
    "/temp/one/two": {},
    "/temp/one/two/stream3": {
        "status": "success",
        "is4_uri": "/temp/one/two/stream3",
        "PubId": "22ee31ce-5975-434e-9836-762050544d3e"
    },
    "/temp/one/three": {},
    "/temp/one/three/stream4": {
        "status": "success",
        "is4_uri": "/temp/one/three/stream4",
        "PubId": "864a4dd7-fff3-4fc5-8cfb-ca55f86aed7f"
    },
    "/temp/one_four": {},
    "/temp/one_four/two": {}
}
```

This shows the results of creating the file(s). Each value is the body of the response upon creation.

Subscribing to a stream

StreamFS offers a subscription facility. StreamFS uses the url specified by the "target" field in the subscription request to POST the data to the url as soon as it comes into StreamFS. Lets set one up. First, lets create the subscription request object and POST it to the `/sub` file, which is where the subscription handler lives.

```
echo "{\"s_uri\":\"/temp/stream1\", \
\"target\":\"http://localhost:1337\"}" > subreq.json
curl -i -X POST "http://localhost:8080/sub" -d@subreq.json
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 20 Jan 2012 03:51:49 GMT

{
    "operation": "subscribe",
    "status": "success",
    "subid": "eda7f7ee-99a1-4808-8d68-4be2562dd3bd"
}
```

Notice, the reply includes a unique identifier for this subscription. It also creates a file in the /sub directory as an active file for managing the subscription and attaining information about it.

```
curl -i -X GET "http://localhost:8080/sub"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 20 Jan 2012 03:53:07 GMT
{
    "status": "success",
    "type": "DEFAULT",
    "properties": {},
    "children": [
        "0828106",
        "all"
    ]
}
```

```
curl -i -X GET "http://localhost:8080/sub/0828106"
```

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 20 Jan 2012 03:53:31 GMT
```

```
{
    "status": "success",
    "subid": "eda7f7ee-99a1-4808-8d68-4be2562dd3bd",
    "destination": "http://localhost:1337",
    "sourceId": "789cf943-bbc8-428e-97ce-03e7cfe5fc12",
    "sourcePath": "/temp/stream1"
}
```

Notice, by calling GET on the subscription resource we can obtain information about it. Now lets test the the subscription. Here's a small Node.js script you can use to act as a simple receiver for the incoming data from the stream.

```
var http = require('http');

function handlePost(req, res){
    req.on('data', function(chunk) {
                  console.log("got some data here");
                //console.log("Receive_Event::" + chunk.toString());
               });
    res.writeHead(200, {'Content-Type': 'text/json'});
    res.end("{\"status\":\"success\"}");
}

var server= http.createServer(function(req,res){
    req.setEncoding('utf8');

    console.log(req.headers);

    req.on('data', function(chunk) {
        console.log("Receive_Event::" + chunk);
    });

    req.on('end', function() {
        console.log('on end');
        console.log("Bytes received: " + req.socket.bytesRead);
        if(req.method=='POST'){
            handlePost(req,res);
        } else{
            res.writeHead(200, {'Content-Type': 'text/plain'});
            res.end();
        }
    });
});
server.listen(1337, "localhost");
```

```
console.log('Server running at http://localhost:1337/');
```

You can copy-paste this code in a file and running using it Node.js. After you've installed, start it and POST data to the stream resource.

```
curl -i -X POST "http://localhost:8080/temp/stream1\
?type=generic&pubid=789cf943-bbc8-428e-97ce-03e7cfe5fc12" \
-d "{\"data\":123}"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 20 Jan 2012 03:59:32 GMT

{"status":"success"}
```

```
$ node dumblistener.js

Server running at http://127.0.0.1:1337/

Receive_Event::
{
    "data": 123,
    "ts": 1327032612,
    "pubid": "789cf943-bbc8-428e-97ce-03e7cfe5fc12",
    "timestamp": 1327032612,
    "PubId": "789cf943-bbc8-428e-97ce-03e7cfe5fc12",
    "is4_uri": "/temp/stream1/"
}
```

Notice, the data object is received successfully by the listener script. You write code to consume incoming data through an HTTP POST and run, collect data for whatever streams you have in your StreamFS instance.

Now, if we want to cancel the subscription to stop incoming data from being forwarded to the external script, we simple delete the resource with an HTTP DELETE call to it.

```
curl -i -X DELETE "http://localhost:8080/sub/0828106"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 20 Jan 2012 04:14:48 GMT

{"status":"success"}
```

B.8 Creating symbolic links

A very important feature in StreamFS is the ability to create symbolic links – files that point to other files. Why is this important? In the case of sensor data management, we want to be able to access the data source through multiple names. So if we have a temperature sensor that is both inside a room and belongs to a set of things I own, I can place it in the room directory and symbolically link to it from my personal directory.

Lets go ahead and create a symbolic link. As usual, we create a json documen with the operation that will be carried out by StreamFS. Lets name the symlink "stream1_link".

```
echo "{\"operation\":\"create_symlink\",\
 \"uri\":\"/temp/stream1\", \"name\":\"stream1_link\"}" > \
 create_symlink.json
```

Now lets POST it to the "temp" file and to create it.

```
curl -i -X POST "http://184.106.109.119:8080/temp" -d@create_symlink.json

HTTP/1.1 201 Created
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 20 Jan 2012 23:21:24 GMT
```

If everything went well, you should get a response with the HTTP status code 201. Now that it's up and created, lets go ahead and see how it is listed when we call the parent directory.

```
curl -i "http://184.106.109.119:8080/temp"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 20 Jan 2012 23:21:26 GMT

{
    "status": "success",
    "type": "DEFAULT",
    "properties": {},
    "children": [
        "stream1",
        "stream1_link -> /temp/stream1"
    ]
}
```

Notice the child named "stream1_link". The arrow indicate that it is a symbolic link that points to `/temp/stream1`. Therefore all HTTP requests to `/temp/stream1_link` will be forwarded to the `/temp/stream1` file and the response will look as if it had come from there. Lets check that first hand.

```
curl -i "http://184.106.109.119:8080/temp/stream1_link"
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 20 Jan 2012 23:21:38 GMT

{
    "status": "success",
    "pubid": "789cf943-bbc8-428e-97ce-03e7cfe5fc12",
    "head": {},
    "properties": {
        "units": "psi"
    }
}
```

Compare this to the response we received from `/temp/stream1` when we created it. Notice, it's the same response. Again, this is a useful way to give multiple names to the same file and we use it in other features in StreamFS, for example to perform various aggregation procedures.

Moving a resource

Sometimes you either make a mistake in naming a file or simply want to place it in another directory. StreamFS supports the move operation that allows you to move any file from one location to another or to rename an existing file.

```
echo "{\"operation\":\"move\",\
 \"src\":\"/temp/stream1\", \"dst\":\"/temp/stream2\"}" > \
 move.json
curl -i -X PUT "http://localhost:8080/temp" -d@move.json
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Tue, 10 Apr 2012 03:07:24 GMT

{"status":"success"}
```

B.9 Stream Processing

This section starts getting into the stream processing components of StreamFS. StreamFS supports the ability the run javascript processing scripts on streaming data and return the result via a stream file/resource. The user can either query or subscribe to the output of the resource to see the results. In this section we're going to use the previously created stream file in /temp/stream1 as the source stream to feed through the processing element. We will define a processing element, install it, pipe /temp/stream1 through it and obverse the output as it's being processed.

B.10 Configuration

The processing engine in StreamFS communicates with processing elements meant to run on remote machines. Note the contents of the configuration file in lib/local/rest/resources/proc/config/serverlist.json.

```
{
    "procservers":[
        {
        "name":"proc1",
        "host":"127.0.0.1",
        "port":1337
        }
    ]
}
```

This configuration file consists of a list of process element servers. StreamFS automatically load balances between the servers while trying to maintain the highest leve of efficiency. In other words, it will use all the resources of a single machine until it decides to spawn jobs on a new one due to decreasing performance. For testing, lets use the default configuration which starts a proces-element server on the localhost.

B.11 Start the processing element

If you're administering your own copy of StreamFS you're going to have to fire up the processing layer and update the configuration files in StreamFS to point to their location. Each processing element runs in node. Lets starts it with the following command in a new terminal:

```
node lib/local/rest/resources/proc/js/runner.js
```

B.12 Creating a processing job

The script below is a request to to create a new process. The name is how the name of the file/resource to be created in /proc. The winsize is the window size that will induce the process to run. A winsize of 10 means that when the window has 10 elements it in, pass the window of values to the function defined by func. The materialize keyword is a boolean that sets the output of the function to be saved in the database for querying or not. The timeout parameters is the time out in milliseconds for the process to run. This is a special case where both the winsize and the timeout are specified. The timeout beats out the winsize parameter. In other words, if the timer fires before the buffer has reached winsize, the function runs on the data that is currently in the buffer. Lets save this in a file called saveproc.json.

```
{
    "operation":"save_proc",
    "name":"testproc",
    "script":{
            "winsize":10,
            "materialize":"false",
            "timeout":20000,
            "func":function(buffer, state){
                var outObj = new Object();
                outObj.tag = "processed";
                return outObj;
            }
    }
}
```

You can use it as a template for getting started with creating different types of processing scripts. Make sure that the processing script doesn't have any errors in it before running. Do not surround the function definition in quotes. The function must take the "buffer" variable is an Array of Objects, it also accepts an optional parameter 'state' which holds state associated with the process. This is a generic object that is passed to the function each time it runs. The returned element must be an Object. It could be an object with any number of elements in it. This objects will be sent back to StreamFS and made available through the associated stream element for this process output.

The script that you wish to run on the buffer is defined by func. This function takes a buffer, where buffer.length ¡= winsize, performs some operation on it, and outputs a buffer. If the materialize option is set to true and the output object contains the "timestamp" and "value" fields, it is saved for processing later on. We create it by POSTing it to /proc.

```
curl -i -X POST http://localhost:8080/proc -d@saveproc.json
```

This returns a HTTP 201 response. We can check that the new resource has been created with a GET request to /proc.

```
curl -i http://localhost:8080/proc
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 11 May 2012 12:24:41 GMT

{
    "status": "success",
    "type": "DEFAULT",
    "properties": {
        "status": "active"
    },
    "children": [
        "testproc"
    ]
}
```

Notice that one of the children is named testproc. Lets call GET on that new resource. Notice the properties object. It contains a variant of the script that was entered. It's the basic code that will be run on the data directed through a running instance of the script.

```
curl -i http://localhost:8080/proc/testproc


HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 11 May 2012 12:25:06 GMT

{
    "status": "success",
    "type": "PROCESS_CODE",
    "properties": {
        "operation": "save_proc",
        "name": "testproc2",
        "script": {
            "winsize": 10,
            "materialize": "false",
            "timeout": 20000,
            "func": {
```

```
                "params": [
                    "inbuf"
                ],
                "text": "var outObj = new Object();
                         outObj.tag = \"processed\";
                         return outObj;"
            }
        }
    },
    "children": [
        "15a73498cfed"
    ]
}
```

B.13 Start the process

Lets get a test process started. First lets create a request object and POST it to StreamFS to get the process 'installed'.

```
echo '{"path":"/temp/stream1", "target":"/proc/testproc"}' > subreq.json
```

Now lets post it to the subscription path in /sub to create it.

```
curl -i -X POST "http://localhost:8080/sub/" -d@subreq.json
```

The response looks like a regular subscription response. However, this is a special subscription. It's a subscription that directs incoming data through a running process.

```
HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 11 May 2012 12:23:37 GMT

{
    "operation": "subscribe",
    "status": "success",
    "subid": "bc2e1035-7f19-4a34-baa2-60e49470a988"
}
```

Below I have included a nodejs script that you can copy-paste in order to view the output of the running process. Copy-paste it and fire it up with node.

```
var http = require('http');
```

```
function handlePost(req, res){
    console.log("Handling post event");
    res.writeHead(200, {'Content-Type': 'text/json'});
    res.end("{\"status\":\"success\"}");
}
var server= http.createServer(function(req,res){
    req.setEncoding('utf8');
    console.log(req.headers);
    req.on('data', function(chunk) {
               console.log("Receive_Event::" + chunk);
               if(req.method=='POST'){
                   handlePost(req,res);
               }
            });

    req.on('end', function() {
        console.log('on end');
    });

    console.log("Bytes received: " + req.socket.bytesRead);
    if(req.method!='POST'){
        res.writeHead(200, {'Content-Type': 'text/plain'});
        res.end();
    }
}).listen(1338, "localhost");
console.log('Server running at http://localhost:1338/');
```

Notice, the stream resource is a child of the testproc resource that was created when you saved the script. The stream resource was created after the subscription was installed (the initial one) that is piping incoming data through a running instance of the process.

```
echo '{"path":"/proc/testproc/15a73498cfed/", "target":"http://localhost:1338"}'
    > subreq2.json
```

Now, we want to subscribe to this resource in order to observe the output of the process as data runs through it.

```
curl -i -X POST "http://localhost:8080/sub/" -d@subreq2.json

HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 11 May 2012 12:30:00 GMT
```

```
{
    "operation": "subscribe",
    "status": "success",
    "subid": "14637d35-68d2-4757-bdf8-4438b12fce1f"
}
```

Viewing the output

As the process run, the output should look like similar to the output shown below:

```
{ 'content-type': 'application/json',
  'cache-control': 'no-cache',
  pragma: 'no-cache',
  'user-agent': 'Java/1.7.0_03',
  host: 'localhost:1338',
  accept: 'text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2',
  connection: 'close',
  'content-length': '66' }
Bytes received: 247
Receive_Event::{
              "tag":"processed",
              "PubId":"a58227c6-2324-4a6a-bca8-72411d27340f"
                       }
Handling post event
on end
```

B.14 Stopping the process

To stop the process altogether, simple delete the stream resource, the subscription, or even the source, /temp/stream1. Any of those will stop the process altogether.

```
curl -i -X DELETE http://localhost:8080/proc/testproc2/15a73498cfed

HTTP/1.1 200 OK
Transfer-encoding: chunked
Content-type: application/json
Connection: close
Date: Fri, 11 May 2012 12:35:41 GMT
```

Bibliography

[1] Yuvraj Agarwal et al. "BuildingDepot: an extensible and distributed architecture for building data storage, access and sharing". In: *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. BuildSys '12. Toronto, Ontario, Canada: ACM, 2012, pp. 64–71. ISBN: 978-1-4503-1170-0. DOI: 10.1145/2422531.2422545. URL: http://doi.acm.org/10.1145/2422531.2422545.

[2] Yuvraj Agarwal et al. "Enabling Building Energy Auditing Using Adapted Occupancy Models". In: Buildsys'11. Seattle, WA, 2011, p. 6.

[3] American Society of Heating, Refrigerating and Air-Conditioning Engineers. *ASHRAE Standard 135-1995: BACnet*. ASHRAE, Inc., 1995.

[4] *Apache HBase*. http://hbase.apache.org/.

[5] A. Arafat and T. Hasan. "Automatic detection of ECG wave boundaries using empirical mode decomposition". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2009.

[6] A. Aswani et al. "Reducing Transient and Steady State Electricity Consumption in HVAC Using Learning-Based Model-Predictive Control". In: *Proceedings of the IEEE* 100.1 (2012), pp. 240–253. ISSN: 0018-9219. DOI: 10.1109/JPROC.2011.2161242.

[7] M. Balazinska et al. "Data Management in the Worldwide Sensor Web". In: *Pervasive Computing, IEEE* 6.2 (2007), pp. 30–40. ISSN: 1536-1268. DOI: 10.1109/MPRV.2007.27.

[8] Gowtham Bellala et al. "Towards an Understanding of Campus-Scale Power Consumption". In: Buildsys'11. Seattle, WA, 2011, p. 6.

[9] Vincent D. Blondel et al. "Fast unfolding of communities in large networks". In: *J.STAT.MECH.* (2008).

[10] Christopher Brooks et al. *Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)*. Tech. rep. UCB/EECS-2007-7. EECS Department, University of California, Berkeley, 2007. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-7.html.

[11] Paul Castro et al. "A Probabilistic Room Location Service for Wireless Networked Environments". In: 2001, pp. 18–34.

[12] Chen Chen et al. "Graph OLAP: Towards Online Analytical Processing on Graphs". In: *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*. ICDM '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 103–112. ISBN: 978-0-7695-3502-9. DOI: 10.1109/ICDM.2008.30. URL: http://dx.doi.org/10.1109/ICDM.2008.30.

[13] Yixin Chen and Li Tu. "Density-based clustering for real-time stream data". In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD '07. 2007.

[14] Henrik Baerbak Christensen and Jakob Bardram. "Supporting Human Activities - Exploring Activity-Centered Computing". In: *Proceedings of the 4th international conference on Ubiquitous Computing*. UbiComp '02. Goteborg, Sweden: Springer-Verlag, 2002, pp. 107–116. ISBN: 3-540-44267-7. URL: http://dl.acm.org/citation.cfm?id=647988.741475.

[15] W. Steven Conner, Lakshman Krishnamurthy, and Roy Want. "Making Everyday Life Easier Using Dense Sensor Networks". In: *Proceedings of the 3rd international conference on Ubiquitous Computing*. UbiComp '01. Atlanta, Georgia, USA: Springer-Verlag, 2001, pp. 49–55. ISBN: 3-540-42614-0. URL: http://dl.acm.org/citation.cfm?id=647987.741329.

[16] Drury B. Crawley and Linda K. Lawrie. *ENERGYPLUS: NEW CAPABILITIES IN A WHOLE-BUILDING ENERGY SIMULATION PROGRAM*.

[17] David E. Culler, Klaus Erik Schauser, and Thorsten von Eicken. *Two Fundamental Limits on Dataflow Multiprocessing*. Tech. rep. UCB/CSD-92-716. EECS Department, University of California, Berkeley, 1992. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/6259.html.

[18] Stephen Dawson-Haggerty et al. "BOSS: building operating system services". In: *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 443–458. URL: http://dl.acm.org/citation.cfm?id=2482626.2482669.

[19] Stephen Dawson-Haggerty et al. "sMAP: a simple measurement and actuation profile for physical information". In: *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. SenSys '10. Zürich, Switzerland, 2010.

[20] Department of Energy. *2011 Buildings Energy Data Book*. http://buildingsdatabook.eren.doe.gov/.

[21] Department of Energy. *USGBC Exploring A New Kind of LEED Plaque*. http://www.leeduser.com/blogs/usgbc-exploring-new-kind-leed-plaque-v4-gbci-certification-platinum.

[22] Colin Dixon et al. "An operating system for the home". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 25–25. URL: http://dl.acm.org/citation.cfm?id=2228298.2228332.

[23] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*. Wiley, 2001.

[24] Echelon Corporation. *LonTalk Protocol Specification*. Echelon Corp. 1994.

[25] Patrick Th. Eugster and Rachid Guerraoui. *Content-Based Publish/Subscribe with Structural Reflection*. 2001.

[26] Patrick Th. Eugster et al. "The many faces of publish/subscribe". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: http://doi.acm.org/10.1145/857076.857078.

[27] Romain Fontugne et al. "Empirical mode decomposition for intrinsic-relationship extraction in large sensor deployments". In: *Workshop on Internet of Things Applications*. IoT-App'12. 2012.

[28] Romain Fontugne et al. "Strip, bind, and search: a method for identifying abnormal energy consumption in buildings". In: *Proceedings of the 12th international conference on Information processing in sensor networks*. IPSN '13. 2013.

[29] Ge Gao and Kamin Whitehouse. "The self-programming thermostat: optimizing setback schedules based on home occupancy patterns". In: BuildSys'09. Berkeley, California, 2009, pp. 67–72. ISBN: 978-1-60558-824-7. DOI: http://doi.acm.org/10.1145/1810279.1810294. URL: http://doi.acm.org/10.1145/1810279.1810294.

[30] P. Gardner and L. Ward. *Energy Management Systems in Buildings: The Practical Lessons*. Energy Publications, 1987. ISBN: 9780905332543. URL: http://books.google.com/books?id=_1wKfAEACAAJ.

[31] Neil Gershenfeld, Stephen Samouhos, and Bruce Nordman. "Intelligent Infrastructure for Energy Efficiency". In: *Science* 327.5969 (2010), p. 3.

[32] Neil Gershenfeld, Stephen Samouhos, and Bruce Nordman3. "Intelligent Infrastructure for Energy Efficiency". In: *Science Magazine* (2010).

[33] Jim Gray et al. "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total". In: *Proceedings of the Twelfth International Conference on Data Engineering*. ICDE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 152–159. ISBN: 0-8186-7240-4. URL: http://dl.acm.org/citation.cfm?id=645481.655593.

[34] *HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer*. http://haproxy.1wt.eu/.

[35] Ismail Haritaoglu. "InfoScope: Link from Real World to Digital Information Space". In: *Proceedings of the 3rd international conference on Ubiquitous Computing*. UbiComp '01. Atlanta, Georgia, USA: Springer-Verlag, 2001, pp. 247–255. ISBN: 3-540-42614-0. URL: http://dl.acm.org/citation.cfm?id=647987.741331.

[36] Lars Erik Holmquist et al. *Smart-Its Friends: A Technique for Users to Easily Establish Connections between Smart Artefacts*. 2001.

[37] Jeff Hsu et al. "HBCI: human-building-computer interaction". In: *Proceedings of the 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Building*. BuildSys '10. Zurich, Switzerland: ACM, 2010, pp. 55–60. ISBN: 978-1-4503-0458-0. DOI: 10.1145/1878431.1878444. URL: http://doi.acm.org/10.1145/1878431.1878444.

[38] Hai Huang and Jiaqiang Pan. "Speech pitch determination based on Hilbert-Huang transform". In: *Signal Processing* 86.4 (2006), pp. 792 –803. ISSN: 0165-1684. DOI: http://dx.doi.org/10.1016/j.sigpro.2005.06.011. URL: http://www.sciencedirect.com/science/article/pii/S0165168405002367.

[39] Norden E. Huang. "Computing frequency by using generalized zero-crossing applied to intrinsic mode functions". Pat. 6990436. 2006.

[40] Norden E. Huang et al. "On Instantaneous Frequency". In: *Advances in Adaptive Data Analysis* 1.2 (2009).

[41] Norden E. Huang et al. "The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis". In: *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 454.1971 (1998), pp. 903–995. DOI: 10.1098/rspa.1998.0193. eprint: http://rspa.royalsocietypublishing.org/content/454/1971/903.full.pdf+html. URL: http://rspa.royalsocietypublishing.org/content/454/1971/903.abstract.

[42] Jonathan W. Hui and David E. Culler. "Extending IP to Low-Power, Wireless Personal Area Networks". In: *Internet Computing, IEEE* 12.4 (2008).

[43] *Java Universal Network/Graph Framework*. http://jung.sourceforge.net/.

[44] Xiaofan Jiang et al. "Design and implementation of a high-fidelity AC metering network". In: *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*. IPSN '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 253–264. ISBN: 978-1-4244-5108-1. URL: http://dl.acm.org/citation.cfm?id=1602165.1602189.

[45] Yifei Jiang et al. "MAQS: a personalized mobile sensing system for indoor air quality monitoring". In: *Proceedings of the 13th international conference on Ubiquitous computing*. UbiComp '11. Beijing, China: ACM, 2011, pp. 271–280. ISBN: 978-1-4503-0630-0. DOI: 10.1145/2030112.2030150. URL: http://doi.acm.org/10.1145/2030112.2030150.

[46] Krasimira Kapitanova et al. "Being SMART about failures: assessing repairs in SMART homes". In: *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. UbiComp '12. 2012.

[47] Younghun Kim et al. "Granger causality analysis on IP traffic and circuit-level energy monitoring". In: BuildSys'10. Zurich, Switzerland, 2010, pp. 43–48. ISBN: 978-1-4503-0458-0. DOI: http://doi.acm.org/10.1145/1878431.1878442. URL: http://doi.acm.org/10.1145/1878431.1878442.

[48] Jahyoung Koo, Jiyoung Yi, and Hojung Cha. "Localization in mobile ad hoc networks using cumulative route information". In: *Proceedings of the 10th international conference on Ubiquitous computing*. UbiComp '08. Seoul, Korea: ACM, 2008, pp. 124–133. ISBN: 978-1-60558-136-1. DOI: 10.1145/1409635.1409652. URL: http://doi.acm.org/10.1145/1409635.1409652.

[49] Andrew Krioukov and David Culler. "Personal building controls". In: *Proceedings of the 11th international conference on Information Processing in Sensor Networks*. IPSN '12. Beijing, China: ACM, 2012, pp. 157–158. ISBN: 978-1-4503-1227-1. DOI: 10.1145/2185677.2185726. URL: http://doi.acm.org/10.1145/2185677.2185726.

[50] Edward A. Lee et al. *Overview of the Ptolemy Project*. Tech. rep. UCB/ERL M01/11. EECS Department, University of California, Berkeley, 2001. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2001/3947.html.

[51] T. Lee and T. B. M. J. Ouarda. "Prediction of climate nonstationary oscillation processes with empirical mode decomposition". In: *Journal of Geophysical Research: Atmospheres* 116.D6 (2011).

[52] Jiakang Lu and Kamin Whitehouse. "Smart blueprints: automatically generated maps of homes and the devices within them". In: *Proceedings of the 10th international conference on Pervasive Computing*. Pervasive'12. 2012.

[53] Xi Lu, Michael McElroy, and Juha Kiviluoma. "Global potential for wind-generated electricity". In: *Proceedings of the National Academy of Sciences* 106.27 (July 2009). URL: http://www.pnas.org/content/106/27/10933.full.

[54] Paul Lukowicz et al. "WearNET: A Distributed Multi-sensor System for Context Aware Wearables". In: *Proceedings of the 4th international conference on Ubiquitous Computing*. UbiComp '02. Goteborg, Sweden: Springer-Verlag, 2002, pp. 361–370. ISBN: 3-540-44267-7. URL: http://dl.acm.org/citation.cfm?id=647988.741494.

[55] Dimitrios Lymberopoulos. "A methodology for extracting temporal properties from sensor network data streams". In: *In Proceedings of the 7th ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys âĂŹ09*. 2009.

[56] Yudong Ma et al. "Predictive Control for Energy Efficient Buildings with Thermal Storage: Modeling, Stimulation, and Experiments". In: *Control Systems, IEEE* 32.1 (2012), pp. 44–64. ISSN: 1066-033X. DOI: 10.1109/MCS.2011.2172532.

[57] Sunil Mamidi, Yu-Han Chang, and Rajiv Maheswaran. "Improving building energy efficiency with a network of sensing, learning and prediction agents". In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS '12. Valencia, Spain: International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 45–52. ISBN: 0-9817381-1-7, 978-0-9817381-1-6. URL: http://dl.acm.org/citation.cfm?id=2343576.2343582.

[58] *Memcached*. http://memcached.org/.

[59] Sean Meyn et al. "Anomaly detection using projective Markov models in a distributed sensor network". In: CDC'09. Shanghai, China, 2009.

[60] Mark Modera et al. "Efficient Thermal Energy Distribution in Commercial Buildings Final Report to California Institute for Energy Efficiency". In: *Environmental Energy Technologies Division, LBNL Technical Report* (1994).

[61] H. Mohammadzade et al. "BEMD for expression transformation in face recognition". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2011.

[62] Next10. *Untapped Potential of Commericial Buildings: Energy Use and Emissions*. 2010. URL: http://next10.org/next10/pdf/NXT10_BuildingEfficiencies_final.pdf.

[63] H. Ochiai et al. "FIAP: Facility information access protocol for data-centric building automation systems". In: *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*. 2011, pp. 229–234. DOI: 10.1109/INFCOMW.2011.5928814.

[64] Jorge Ortiz et al. "Towards real-time, fine-grained energy analytics in buildings through mobile phones". In: *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. BuildSys '12. Toronto, Ontario, Canada: ACM, 2012, pp. 42–44. ISBN: 978-1-4503-1170-0. DOI: 10.1145/2422531.2422540. URL: http://doi.acm.org/10.1145/2422531.2422540.

[65] R. Richards. ""Representational State Transfer (REST)" in "Pro PHP XML and Web Services"". In: Springer Publishing, 2006. Chap. 17, pp. 633–672. URL: http://dx.doi.org/10.1007/978-1-4302-0139-7_17.

[66] D. M. Ritchie and K. Thompson. "The Unix Time-Sharing System". In: *Communications of the ACM* 17 (1974), pp. 365–375.

[67] Kay Romer. *The Lighthouse Location System for Smart Dust*. 2003.

[68] David S. Rosenblum and Alexander L. Wolf. "A Design Framework for Internet-Scale Event Observation and Notification". In: *In Proc. of the 6 th European Software Engineering Conf. held jointly with the 5 th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE97), number 1301 in LNCS*. Springer, 1997.

[69] Mirco Rossi, Burcu Cinaz, and Gerhard Tröster. "Ready-to-live: wearable computing meets fashion". In: *Proceedings of the 13th international conference on Ubiquitous computing*. UbiComp '11. Beijing, China: ACM, 2011, pp. 609–610. ISBN: 978-1-4503-0630-0. DOI: 10.1145/2030112.2030238. URL: http://doi.acm.org/10.1145/2030112.2030238.

[70] Margo Seltzer and Nicholas Murphy. "Hierarchical file systems are dead". In: *Proceedings of the 12th conference on Hot topics in operating systems*. HotOS'09. Monte Verità, Switzerland: USENIX Association, 2009, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1855568.1855569.

[71] Jay Taneja, Randy Katz, and David Culler. "Defining CPS Challenges in a Sustainable Electricity Grid". In: *Proceedings of the 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*. ICCPS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 119–128. ISBN: 978-0-7695-4695-7. DOI: 10.1109/ICCPS.2012.20. URL: http://dx.doi.org/10.1109/ICCPS.2012.20.

[72] Jay Taneja et al. *Enabling Advanced Environmental Conditioning with a Building Application Stack*. Tech. rep. UCB/EECS-2013-14. EECS Department, University of California, Berkeley, 2013. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-14.html.

[73] TIBCO. *Using TIBCO Enterprise Message Service with Storage Foundation Cluster File System Increasing Availability and Performance*. 2006.

[74] Sameer Tilak et al. "A file system abstraction for sense and respond systems". In: *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*. EESR '05. Seattle, Washington: USENIX Association, 2005, pp. 1–6. ISBN: 1-931971-32-3. URL: http://dl.acm.org/citation.cfm?id=1072530.1072532.

[75] U.S. Environmental Protection Agency. *Buildings Energy Data Book*. 2010. URL: http://buildingsdatabook.eren.doe.gov/docs/DataBooks/2010_BEDB.pdf.

[76] *U.S. Green Building Council Leadership in Energy and Environmental Design*. http://www.usgbc.org/leed/.

[77] Lingfeng Wang, Zhu Wang, and Rui Yang. "Intelligent Multiagent Control System for Energy and Comfort Management in Smart and Sustainable Buildings". In: *Smart Grid, IEEE Transactions on* 3.2 (2012), pp. 605–617. ISSN: 1949-3053. DOI: 10.1109/TSG.2011.2178044.

[78] Roy Want et al. "The Personal Server: Changing the Way We Think about Ubiquitous Computing". In: *In Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing, Goteborg*. Springer-Verlag, 2002, pp. 194–209.

[79] Mark Weiser. "Human-computer interaction". In: ed. by Ronald M. Baecker et al. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995. Chap. The computer for the 21st century, pp. 933–940. ISBN: 1-55860-246-1. URL: http://dl.acm.org/citation.cfm?id=212925.213017.

[80] *WSNFuse: accessing Wireless Sensor Networks as a Filesystem*. http://sourceforge.net/projects/wsnfuse/.

[81] P. Wyckoff et al. "T spaces". In: *IBM Syst. J.* 37.3 (July 1998), pp. 454–474. ISSN: 0018-8670. DOI: 10.1147/sj.373.0454. URL: http://dx.doi.org/10.1147/sj.373.0454.

[82] Sungro Yoon, Kyunghan Lee, and Injong Rhee. "FM-based indoor localization via automatic fingerprint DB construction and matching". In: *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. MobiSys '13. Taipei, Taiwan: ACM, 2013, pp. 207–220. ISBN: 978-1-4503-1672-9. DOI: 10.1145/2462456.2464445. URL: http://doi.acm.org/10.1145/2462456.2464445.

[83] Tina Yu. "Modeling Occupancy Behavior for Energy Efficiency and Occupants Comfort Management in Intelligent Buildings". In: *Proceedings of the 2010 Ninth International Conference on Machine Learning and Applications*. ICMLA '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 726–731. ISBN: 978-0-7695-4300-0. DOI: 10.1109/ICMLA.2010.111. URL: http://dx.doi.org/10.1109/ICMLA.2010.111.