Building Operating Systems Services: An Architecture for Programmable Buildings

Stephen Dawson-Haggerty

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-96
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-96.html

May 16, 2014

Building Operating Systems Services:
An Architecture for Programmable Buildings

by

Stephen Dawson-Haggerty

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 Culler, Chair
Professor Randy Katz
Professor Edward Arens

Spring 2014

The dissertation of Stephen Dawson-Haggerty, titled Building Operating Systems Services: An Architecture for Programmable Buildings, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

Building Operating Systems Services:
An Architecture for Programmable Buildings

Abstract

Building Operating Systems Services:
An Architecture for Programmable Buildings

by

Stephen Dawson-Haggerty

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Culler, Chair

Commercial buildings use 73% of all electricity consumed in the United States [30], and numerous studies suggest that there is a significant unrealized opportunity for savings [69, 72, 81]. One of the many reasons this problem persists in the face of financial incentives is that owners and operators have very poor visibility into the operation of their buildings. Making changes to operations often requires expensive consultants, and the technological capacity for change is unnecessarily limited. Our thesis is that some of these issues are not simply failures of incentives and organization but failures of technology and imagination: with a better software framework, many aspects of building operation would be improved by innovative software applications.

To evaluate this hypothesis, we develop an architecture for implementing building applications in a flexible and portable way, called the Building Operating System Services. BOSS allows software to reliability and portably collect, process, and act on the large volumes of data present in a large building. The minimal elements of this architecture are hardware abstraction, data management and processing, and control design; in this thesis we present a detailed design study for each of these components and consider various tradeoffs and findings. Unlike previous systems, we directly tackle the challenges of opening the building control stack at each level, providing interfaces for programming and extensibility while considering properties like scale and fault-tolerance.

Our contributions consist of a principled factoring of functionality onto an architecture which permits the type of application we are interested in, and the implementation and evaluation of the three key components. This work has included significant real-world experience, collecting over 45,000 streams of data from a large variety of instrumentation sources in multiple buildings, and taking direct control of several test buildings for a period of time. We evaluate our approach using focused benchmarks and case studies on individual architectural components, and holistically by looking at applications built using the framework.

Contents

List of Figures

List of Tables

Acknowledgments

Being at Berkeley these past few years has been a joyful experience. Mostly, this is due to a great group of people to who are probably too numerous to name. David Culler has the clearest mind for architectural thinking I've yet to meet, and couples it with an appreciation for the act of creating the artifact that yields a very interesting research approach. I've learned a lot. Randy Katz has always been a source of encouragement, useful feedback, and amusing British costumes. Ed Arens has been very supportive over the past few years as we started to dig into buildings, and knowing someone who deeply understands and cares about buildings has been extremely helpful.

Prabal Dutta was a wonderful mentor during my first few years here; his way of picking the one important thing out of the soup of irrelevancies and then writing a paper that makes it seem completely obvious is still something I marvel at. I've greatly enjoyed working with Andrew Krioukov as we broke open buildings, figured out what makes them tick, and put them back together better. The whole 410 crew – originally Jay Taneja, Jorge Ortiz, Jaein Jeong, Arsalan Tavakoli, and Xiaofan Jiang – was a great way to get started in grad school. Some of us made the transition from motes to energy together, which was frustrating and rewarding all at the same time. I've also greatly benefited from discussions with many other people at Berkeley, notably Ariel Rabkin, Tyler Hoyt, Arka Bhattacharya, Jason Trager, and Michael Anderson.

Albert Goto bas been a constant presence and help, always good for a birthday cake, a late-night chat about NFS over `6loWPAN`, or help with whatever needed doing. Scott Mcnally and Domenico Caramagno gave generously of their time in helping to understand how our buildings work, and indulging us as we learned how to change them. I also valued the time spent working with collaborators at other institutions, especially Steven Lanzisera and Rich Brown at LBNL, and JeongGil Ko and Andreas Terzis at Johns Hopkins. Together, we were able to really bring TinyOS to a satisfying point.

Finally, of course, the people who made life outside of school a blast. Abe Othman, Savitar Sundaresan, Chris Yetter, James McNeer and all of the doppels have a special place in my life. Sara Alspaugh has been a friend, a partner, and a source of great conversations to whom I will always be grateful. Most importantly, I'd like to thank my parents, John Haggerty and Sally Dawson. They are always supportive and loving, and somehow let us think that everyone has a Ph.D.! Well, I guess it's close to true. My brother Mike is an inspiration and an example of how to know what you love and go do it.

Chapter 1

Introduction and Motivation

According to a 2010 Energy Information Administration (EIA) report, the commercial sector accounts for 19% of all energy consumption in the United States [30], much of which is spent in buildings and much of which is thought to be wasted. Buildings are already some of the largest and most prevalent deployments of "sensor networks" in the world, although they are not typically recognized as such. Locked in proprietary stovepipe solutions or behind closed interfaces, a modern commercial building contains thousands of sensors and actuators that are more or less the same as those used in a typical sensor network deployment: temperature, humidity, and power are the most common transducers. Many commercial buildings already contain the dense instrumentation often posited as the goal of sensor network deployments, but combine it with a relatively unsophisticated approach to applying those data to a host of different problems and applications. Through better use of existing systems, we may be able to make a dent in buildings' energy usage.

The overarching goal of this thesis is to lay the groundwork for **building applications**: a platform siting on top of the sensing and actuation already present in existing buildings, providing interesting new functionality. Specifically, we are interested in enabling applications with a few key properties; applications which are **portable**, able to be easily moved from one building to another so as to enable changes to building operation which scale like software, rather than hardware on building renovation. These applications are often also **integrative**, in the sense that they bring together sources of data and information which are very diverse. They take advantage of efficiencies and new capabilities possible without expensive hardware retrofits. Finally, these new applications must exhibit **robustness** in the face of a variety of failure modes, some of which were introduced by extending the scope of what is thought of as a "building application."

1.1 Underlying Trends

Despite consuming 70% of U.S. electricity, the building sector exhibits surprisingly little innovation for reducing its consumption. The reasons include low levels of investment in

R&D and a general focus on performance metrics like comfort and reliability over energy efficiency. Optimizing and maintaining the performance of a building requires initial commissioning to ensure design goals are met and continuous attention to operational issues to ensure that efficiencies are not lost as the physical building evolves. Monitoring-based continuous commissioning only begins to address the challenges for maintaining peak building performance.

Efficiency is not yet evaluated to the same standard as comfort and reliability, but with better user input, control policies, and visibility into the buildings state, energy consumption can be intelligently reduced. Computer systems can deliver wide-scale, robust, and highly sophisticated management services at low marginal cost. However, applying software techniques to millions of commercial buildings with hundreds of millions of occupants demands a rethinking of how such software is procured, delivered, and used. Building Software today is something of a misnomer, as it is typically embedded in a proprietary Building Management System (BMS) or Building Automation System (BAS), providing rudimentary plotting, alarming, and visualization tools with few capabilities for extensibility or continuous innovation.

Due to increased efforts of combat climate change, it has become important to deploy renewable energy assets, such as wind and solar, onto the electric electric grid. In fact, the past few years have seen impressive growth in both categories [16, 58]. Because electricity storage is relatively expensive and the schedule of when these new resources produce electricity depends on uncontrollable natural factors, a key challenge to increasing the use of renewables is the development of zero-emission load balancing. This allows consumers of electricity to respond to a shortfall in generation by reducing demand, rather than producers attempting to increase generation. Buildings are an important target for this approach because they contain large thermal masses and thus have significant flexibility as to when they consume electricity. For instance, a building could reduce cooling load for a certain period, allowing it to "coast" on it's existing thermal mass. Applications which perform this service require both access to the operation of the buildings as well as communication with the electric grid operator, and make up the second broad class of application we enable.

What is needed is a shift to Software-Defined Buildings: flexible, multi-service, and open Building Operating System Services (BOSS) that allows third-party applications to run securely and reliably in a sandboxed environment. A BOSS is not limited to a single building but may be distributed among multi-building campuses. It provides the core functionality of sensor and actuator access, access management, metadata, archiving, and discovery. The runtime environment enables multiple simultaneously running programs. As in a computer OS, these run with various privilege levels, with access to different resources, yet are multiplexed onto the same physical resources. It can extend to the Cloud or to other buildings, outsourcing expensive or proprietary operations as well as load sharing, but does so safely with fail-over to local systems when connectivity is disrupted. Building operators have supervisory control over all programs, controlling the separation physically (access different controls), temporally (change controls at different times), informationally (what information leaves the building), and logically (what actions or sequences thereof are allowable).

1.2 Motivating Applications

Broadly speaking, applications for buildings fall into three basic categories. First, some applications involve analysis, optimization, or visualization of the operation of existing systems. Second, others involve the integration of the building into wider scale control loops such as the electric grid. A third class connects occupants of buildings to the operation of their buildings.

An example of the first class of application is a coordinated *HVAC optimization* application. Ordinarily, the temperature within an HVAC zone is controlled to within a small range using a PID controller. The drive to reach an exact set-point is actually quite inefficient, because it means that nearly every zone is heating or cooling at all times. A more relaxed strategy is one of *floating*: not attempting to effect the temperature of the room except within a much wider band. However this is not one of the control policies available in typical commercial systems even though numerous studies indicate that occupants can tolerate far more than the typical 6F variation allowed [5]. Furthermore, the minimum amount of ventilation air provided to each zone is also configured statically as a function of expected occupancy; however the actual requirement in building codes are often stated in terms of fresh, outside air per occupant. Optimization applications might use occupancy information derived from network activity, combined with information about the mix of fresh and return air currently in use to dynamically adjust the volume of ventilation air to each zone.

A second class of application converts a commercial building from a static asset on the electric grid passively consuming electricity to an active participant, making decisions about when and how to consume energy so as to co-optimize both the services delivered within the building but also its behavior as part of a wide-scale electric grid control system. Two strategies which have become more commonplace in recent years are time-of-use pricing, and demand response. In an electric grid with time-of-use pricing the tariffs electricity consumers are charged vary based on a schedule; for instance, electricity during peak demand hours could be significantly more expensive than the rate at night. In a demand response system, utilities gain the ability to send a signal to electricity consumers commanding them to scale back their demand. Buildings are prime targets for participating in both of these strategies, because they consume most of the electricity but also can have significant flexibility about when they choose to consume due to their large thermal mass and slow-changing nature of many of their loads.

Finally, an example of a user responsive application is one that improves comfort by giving occupants direct control of their spaces, inspired by [35]. Using a smart-phone interface, the *personalized control* application gives occupants direct control of the lighting and HVAC systems in their workspaces. The application requires the ability to command the lights and thermostats in the space. The *personalized climate control* application highlights the need for the ability to outsource control, at least, temporarily, to a mobile web interface in a way that falls gracefully back to the local control. It also integrates control over multiple subsystems which are frequently physically and logically separate in a building: HVAC and lighting. This type of interface can improve occupant comfort, as well as save energy through

better understanding of occupant preferences.

1.3 Contributions and Thesis Roadmap

In this thesis, we first present in Chapter 2 a tutorial on building design, looking especially at common of mechanical systems and the computer systems controlling them, so as to give a solid background in the existing state of buildings. We also develop a set of existing, example applications in some detail so as to provide a set of running examples throughout the thesis. Building on this in Chapter 3, we synthesize the essential patterns of application design, and propose an architecture for creating building applications which addresses the key challenges in this area while meeting our overall goals of integration, portability, and robustness. Following this, we develop three key architectural components in details: in Chapter 4 we develop the Simple Measurement and Actuation Profile (sMAP), a system for collecting and organizing the heterogeneous set of devices present at the device layer of buildings. Next, we develop in Chapters 5, 6, and 7 a system for collecting, storing, and processing large volumes of time series data generated from building system. In doing so, we build a system which can store and process tens of billions of data points with millisecond latency within the archive. Finally in Chapter 8, we develop control transactions, an abstraction for making changes to building operations with well-defined semantics around failure.

Our approach at each of these layers is similar: a survey of requirements followed by design, implementation, and evaluation of the layer as a single unit. Finally, we conduct a holistic evaluation of the overall architecture by breaking down the implementation of several representative applications.

Chapter 2

Background and Related Work

Before launching into a discussion of how to redesign the control of building systems, we first present background material on building systems to allow our presentation to be self-contained. For the reader unfamiliar with how large buildings work, we provide a brief primer on their mechanical systems, how their existing computing, communication, and control resources are designed and used, and the few types of computer applications that existing systems are equipped to perform. We conclude this chapter by synthesizing a motivation for what we believe is possible and desirable for building systems: a case for Building Operating System Services.

2.1 Building Physical Design

A large modern commercial building represents the work of thousands of individuals and tens or hundreds of millions of dollars of investment. Most of these buildings contain extensive internal systems to manufacture a comfortable indoor environment: to provide thermal comfort (heating and cooling), good air quality (ventilation), and sufficient lighting; other systems provide for life safety (fire alarms, security), connectivity (networking) and transport (elevators). These systems are frequently provided by different vendors, function separately, and have little interoperability or extensibility beyond the scope of the original system design. As we explore the systems, we keep a close eye on their capabilities, limitations, and design motivations. We also explore some alternative architectures.

Within a building, systems are often separated into separate vertical stovepipes; for instance, those presented in Table 2.1. The logical separation of functions into vertical systems pervades many facets their existence in a building; they are often specified, designed, purchased, installed, and maintained separately from other systems in the building. For this reason, we present a brief overview of a few relevant systems and design considerations, describing the design of systems as they currently exist before launching into a more comprehensive discussion of how they could be more effectively integrated.

Vertical	Description	Example equipment present
HVAC	Responsible for manufacturing a comfortable indoor thermal environment.	Air handlers; fans; dampers; cooling towers; chillers; boilers; heating coils; radiant panels.
Lighting	Provides illumination to indoor areas needed for work, mobility, and emergencies.	Incandescent, fluorescent, LED lighting elements; ballasts; lighting switches and controllers; daylight detectors.
Security	Guards against unauthorized physical access to the space.	Cameras; motion detectors; door locks; smart cards; alarm panels.
Transportation	Moves individuals within the spaces.	Elevators, escalators, automatic doors.
Networking	Moves data within the space.	Cabling; switch gear; telephone PBX, wireless access points.
Life safety	Protects life and property from fire, water, carbon monoxide, and other eventualities.	Smoke detectors; alarm annunciators, standpipes.

Table 2.1: Vertically integrated systems which may be present in a large building, along with key equipment

2.1.1 Heating, Ventilation, and Air Conditioning

Heating, ventilation, and air condition systems (HVAC) are responsible for manufacturing a comfortable thermal environment for occupants of the building. While heating systems have been a common feature of construction for centuries, air conditioning only became possible with the advent of refrigeration in the early 20th century, and only became common in postwar construction. According to the DOE, HVAC accounts for around a third of the energy consumed by an average commercial building [109]. These systems are relatively diverse, with many different refrigeration technologies as well as an assortment of techniques to improve their efficiency.

As an example of one common design point, Figure 2.1 shows a heating, ventilation, and air conditioning (HVAC) system for a large building. Four process loops are evident.

- In the air loop, air is both chilled and blown through ducts within a unit called an air handler, after which it passes through variable air-volume (VAV) boxes into internal rooms and other spaces. The VAV box has a damper, allowing it to adjust the flow of air into each space. After circulating through the rooms, the air returns through a return air plenum where a portion is exhausted and the remaining portion is recirculated. The recirculated air is also mixed with fresh outside air, before being heated or cooled to an appropriate temperature, completing the loop.

- The hot water loop circulates hot water through heat exchangers present in most

VAV boxes; supply air is reheated there before being discharged through diffusers into different rooms. A valve controls the degree to which the air is reheated. The hot water itself is typically heated in a centralized boiler or heating plant. In a slight variant of this architecture, reheat is sometimes provided through the use of electrical (resistive) heating elements, eliminating the hot water loop.

- The cold water loop circulates water from the chiller through a heat exchanger, which chills the supply air and rejects it to the atmosphere using a cooling tower on the room.
- Finally, a secondary cold water loop rejects heat from the chiller to the atmosphere, by circulating water through a cooling tower.

To provide dehumidification, it is common to chill the supply air to a relatively cool temperature (*e.g.*, 55F) before being reheated at the VAV boxes; this is a so-called "reheat" system. There are many other designs for each part of the system, and designs evolve over time to meet different requirements. As such, this system should be considered as an example of a large class of different system designs.

Many different control loops are present; the predominant control type is PID controllers[1], used to meet set-point targets for air pressure, temperature, volume. A few of the most important loops are:

VAV control: VAV control takes as input each zone's temperature and set point, and produces as output a position for the damper and heating coil. The most modern type of VAV control is the so-called "dual max" zone, in which both airflow and heating valve position are adjusted continuously whenever the zone temperature is outside of the zone's "dead band". In such a system, there are actually two temperature set-points: a heating set point and a cooling set point. The system is said to be "floating" within the dead band whenever the temperature is between these two set points, and thus is neither being heated nor cooled.

Duct static pressure: the air coming out of the air handler is pressurized so as to force it through the duct network. Increasing pressure increases airflow to the zones, and thus increases the capability of the system to provide cooling; however, it also increases the energy expended by the fan[2] and deposits more heat into the air from the fan motor. Since the VAV dampers adjust independently of the central air handler, it is necessary to have a control loop that maintains a constant static pressure of the supply air.

[1]Proportional-Integral-Derivative or PID control is a widely used form of process control very common in building plants. A PID controller continuously computes an actuator position (the "loop output") as a function of some input variable and a set point; for instance, an airflow controller will compute the damper position in a duct by observing the current airflow in order to achieve some desired airflow (the set point). Each term in the controller ("P", "I", and "D") refer to an error term computed based on the process's past, present, or future state.

[2]Fan affinity laws relate the flow through a fan to various other properties such as fan diameter and power needed. An important outcome is that the relationship between flow and power for a particular fan is cubic – doubling the volume of air moved requires eight times the power.

Supply air temperature: The air coming out of the air handler is also chilled; the supply air temperature loop adjusts the amount of cooling so as to maintain a constant supply air temperature.

Economizer : the economizer is a damper controlling the mixing of outside air with return air. This mixed air is blown through the air handler, cooled, and becomes the supply air to zones. The economizer setting has a significant energy impact, because the temperature of the air entering the cooling coil determines how much it needs to be chilled. Weather conditions change the outside air temperature, but the return air temperature is typically constant; generally a few degrees warmer than the supply air temperature. This control loop often operates using a fixed mapping from outside air temperature to economizer position.

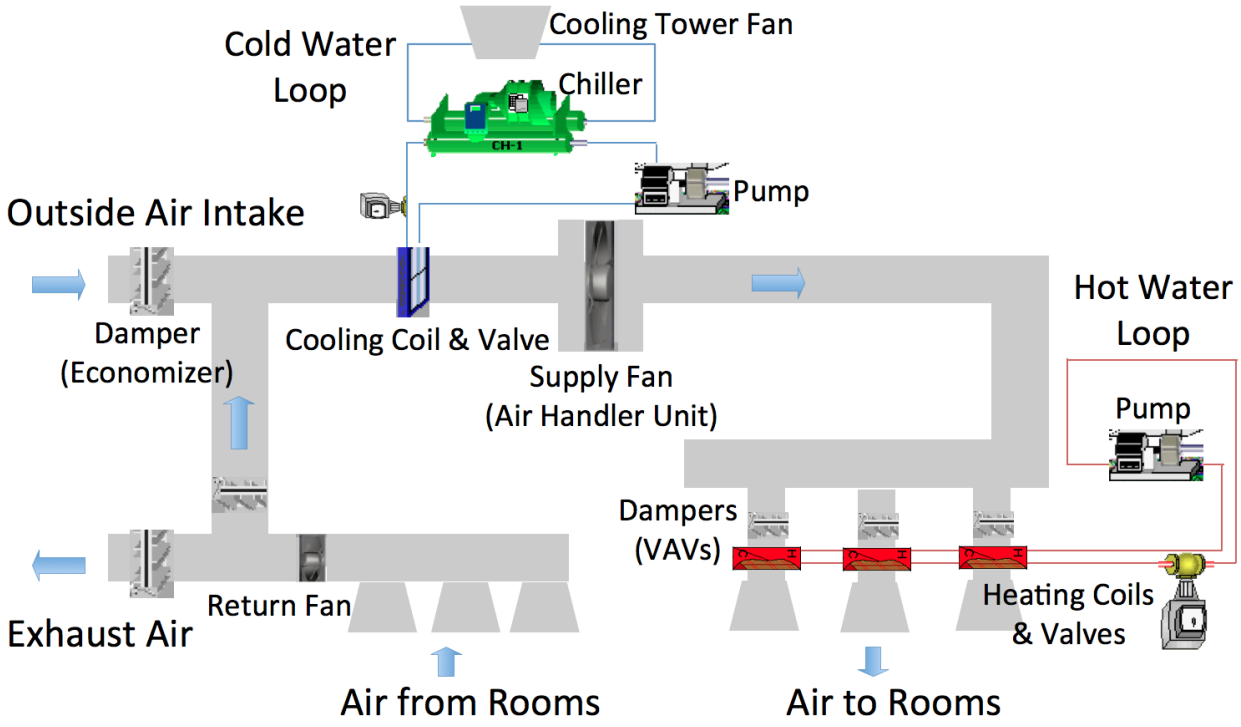


Figure 2.1: A typical process diagram of an HVAC system loop in a commercial building.

2.1.2 Lighting Systems

Lighting system design was once a simple matter of providing the design amount of illumination to interior spaces, typically measured in lumens per square foot. The designer simply computed the footprint of each lighting fixture and ensured that the resulting installation provided sufficient light to each space. Some buildings have only one light switch per floor, resulting in very simple control since the entire space must be lit if anyone requires lighting.

Lighting today is considerably complicated by the overriding design goal of providing sufficient illumination to spaces while minimizing energy costs. For this reason, modern architectural designs emphasize increased use of daylighting, reducing the energy consumption consumed by lighting figures when illumination can be provided by the sun or low-power task lighting. A key challenge in lighting control is mediating the interaction between active lighting elements with passive architectural features. A building designed to take advantage of daylight may have a range of passive features, ranging from exposures, window shades, skylights, and reflective elements designed to bring light into the space as the sun moves across the sky in different seasons while limiting the solar heat gain. Potential active features include, in addition to the obvious lighting elements, photochromic windows and mechanical shading elements that allow the control system to adjust the amount of light brought in. Energy codes have also increased the adoption of dimable ballasts, while technologies like LED also allow for the adjustment of color in addition to brightness.

Figure 2.2: A building interior at the National Renewable Resources Laboratory exhibiting many active and passive features designed for daylighting.

Managing all of this hardware to provide consistent illumination while also reducing energy consumption and solar heat gain is another significant area where control loops play a role in buildings. One vendor implements eight different strategies for managing the lighting energy consumption [70]:

High-end tune and trim: reduce the maximum illumination ever provided in a space.

Occupancy sensing: dim or turn off lights when no one is present.

Daylight harvesting: reduce illumination when the space is lit by insolation.

Personal dimming control: allow occupants to reduce lighting levels as desired.

Controllable window shades: limit solar heat gain during peak solar hours while allowing additional illumination during the morning and evening.

Scheduling: automatically shut off lights according to a schedule.

Demand response: provide a temporary reduction in illumination in response to an external signal.

Overall, these strategies, while individually simple, speak to the range of considerations and interactions present in something as seemingly-simple as adjusting the lights, and how information from different systems (scheduling, occupancy, solar and weather data) are brought together to optimize lighting.

2.1.3 Other Building Systems

Many of the other systems noted in Table 2.1 are just as or even more complex than lighting and HVAC. The data are very diverse; network switches may be able to observe packet flows in a wired network, or even localize clients in a wireless network; security systems have detailed data about the entry and exit of building occupants at the whole-building granularity. Functionality is often duplicated as integration costs are high; for instance, multiple systems may attempt to monitor occupancy, a key control input, by installing separate sensors.

2.2 Monitoring and Control

Knitting together all of the hardware embodied in each of these different systems are monitoring and control systems. Here, we present a brief overview of the most common architecture for these systems: Supervisory Control and Data Acquisition (SCADA), as well as a few alternative paradigms which have been applied in other settings.

2.2.1 Direct and Supervisory Control

Control in existing building systems operates as two logical levels, shown in Figure 2.3. Direct control is performed in open and closed control loops between sensors and actuators: a piece of logic examines a set of input values, and computes a control decision which commands an actuator. These direct control loops frequently have configuration parameters that govern their operation known as set points; they are set by the building operator, installer, or engineer. Adjusting set points and schedules forms an outer logical loop, known as supervisory control. This logical distinction between types of control is typically reflected physically in the components and networking elements making up the system: direct control is performed

by embedded devices, called Programmable Logic Controllers (PLCs) wired directly to sensors and actuators, while supervisory control and management of data for historical use is performed by operator workstations over a shared bus between the PLCs. This architecture is natural for implementing local control loops since it minimizes the number of pieces of equipment and network links information must traverse to affect a particular control policy, making the system more robust, but provides no coordinated control of distinct elements. It imposes hard boundaries which are difficult to overcome.

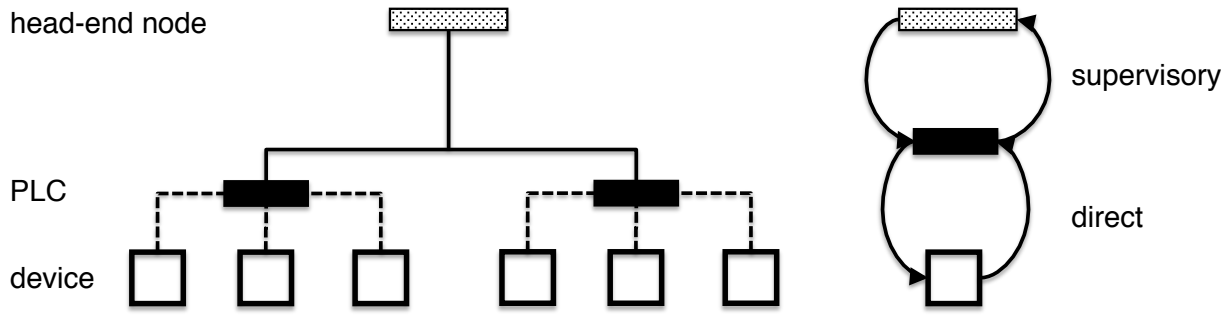


Figure 2.3: The two-level architecture of many existing building separations, with a logical and physical distinction between direct and physical control. This is shared with a typical Supervisory Control and Data Acquisition (SCADA) system.

Existing control models in buildings are relatively simple, even in the best-performing buildings. One characteristic of how existing buildings are architected is that each vertical system has many decoupled or loosely coupled local control loops, which interact only through the media they control, but not directly via signaling.

Each vertical system within the building uses specific algorithms to make the building work. For instance, within the HVAC system, Proportional-Derivative-Integral (PID) controllers are used at the VAV level to maintain temperature and airflow targets, and at the central plant level to maintain constant temperatures and pressure in the ducts and water loops. PID loops compute a *control output* from one or a several input variables. For the purposes of our discussion here, it is simply necessary to know that PID control is a relatively simple, robust, and widely-deployed way of performing direct control in many applications, not just HVAC; they do however require parameter tuning, and there is a deep literature exploring methods of doing so [15, 44, 56, 101, 118]

The cutting edge of building control attempts to replace the many decoupled control loops throughout a building with a more integrated control strategy. For instance, Model-Predictive Control (MPC) is popular in process industries and holds the potential of efficiency improvements by coordinating control of many different elements within the system [3, 7, 8].

2.2.2 Communications Protocols

Today's typical building management system consists of front-end sensors, usually closely associated with actuators, that periodically report their data to a back-end database over one or more link technologies: RS-485, raw Ethernet frames, and IP networks are common. Several computers are typically also present, and provide an interface to users such as facilities managers for adjusting set points and setting schedules; these are then enacted by sending commands back to the front-end devices (Remote Terminal Units, in Modbus terminology). This straightforward architecture is simple and attractive when computing is expensive, because it minimizes the functionality placed at the actual sense points. As processing gets ever cheaper, it makes sense to re-evaluate these design decisions, especially as systems converge on IP as their network layer.

The design space for a web service for physical information consists of three interlocking areas:

Metrology the study of measurement; *what* is necessary to represent a datum.

Syndication concerns *how* a data is propagated out from the sensor into a larger system.

Scalability relates to the *range of devices* and uses the service can support, from small embedded systems to huge Internet data centers.

Each of these concerns presents a set of design issues, some of which have been previously addressed in the academic literature or by industrial efforts. In this work, we examine these previous solutions and build from them a single set of solutions which are designed to solve a specific problem: representing and transmitting physical information.

BACnet

The most important existing protocol in buildings is known as BACnet, which was developed beginning in 1987, and was released as Version 1 in 1995 [4]. "BACnet – A Data Communication Protocol for Building Automation and Control Networks," is managed by a committee of ASHRAE, the American Society of Heating, Refrigeration, and Air-Conditioning Engineers and has been standardized as ISO 16484. The aim of BACnet is relatively simple: to provide a common communication protocol for control-level networks within buildings, with the goal of allowing components from different manufacturers to interoperate, and begin to breaking open some of the stovepipes presents within existing vertical market segments.

As a protocol, BACnet can be best thought of as a protocol which specifies the physical, link, and application layers of the OSI link model. At the physical and link layers, the standard has five options; fully compliant implementations must use either an IP network, Ethernet (without IP), ARCnet (a token-ring serial protocol), MS-TP (master-slave/token-passing, another serial protocol), Echelon LonTalk, or any Point-to-Point link (such as an RS-485 line or a phone line). This diversity of media leads to a certain confusion within the protocol since adaptations must be made to account for some of these modes – for instance,

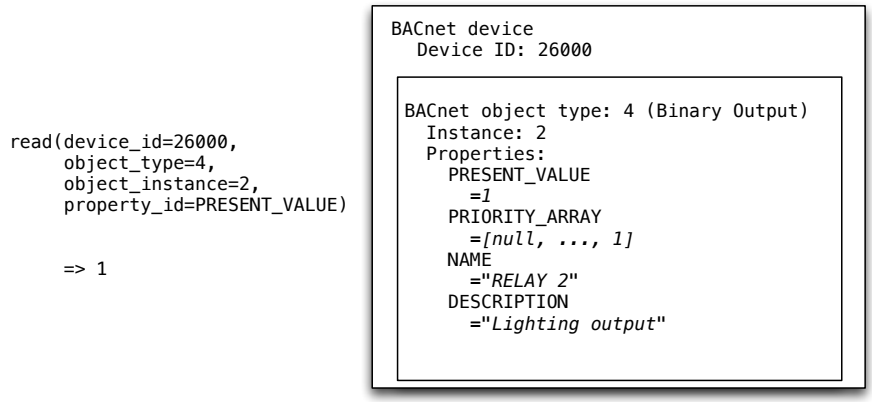


Figure 2.4: Key primitives in BACnet are "devices", "objects," and "properties.". Devices represent physical controllers, or logical network hosts. Objects are somewhat general, but may represent individual points of instrumentation such as a relay switch or point of measurement. Properties on objects are individual values; for instance reading `PROP_PRESENT_VALUE` on a switch will yield the current switch position.

BACnet contains its own network model and addressing system to make up for shortcomings in some of these links.

At an application *metrology* level, BACnet represents the world as a set of "objects," that have "properties" which the protocol manipulates. BACnet objects are not true objects (in the Object-Oriented Programming sense) because they are not associated with any code or executable elements. Instead, the BACnet standard defines a limited taxonomy of objects, properties, and actions to be performed on properties; for instance, it defines a standard "Analog Output" type of object, which represents a physical transducer that can take on a floating point value on its output. BACnet specifies properties requiring the object to expose the range of values it can take on, the present value, the engineering units of the value, as well as optional name and description fields.

For scalability to larger networks, BACnet builds in scoped broadcast-based service discovery via "Who-Is" and "Who-Has" messages which allow clients to discover the names of devices on the network, and to ask them to filter the list of objects they contain using a simple predicate. These services are built on top of link-layer broadcast functionality. It also contains basic features for data *syndication* both in "pull" mode where objects are periodically polled for the latest data, and "push" where data are send to a receiver whenever the value changes by more than a threshold (change-of-value triggering).

BACnet also provides rudimentary support for mediation between different processes or clients within the system, through the use of a static prioritization scheme. For certain types of objects such as Analog Outputs, changing the value of the point is accomplished not by directly setting the `present-value` property, but by placing a value within a priority array. This priority array, shown in Figure 2.5 contains 16 elements. The controller determines

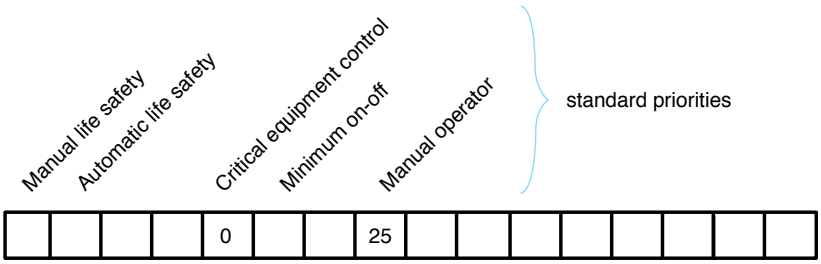


Figure 2.5: An example state of a BACnet priority array. In this case, the present value for this controller would take on the value 0 since that is the highest-priority, non-null value present.

the actual value of of the output by examining this array, and giving the output the value of the highest priority (lowest non-null value) in the array. A separate `relinquish-value` property determines the output value when no elements are present in the array.

Other Protocols

Many other communications protocols are in use within buildings as well. In fact, even control systems using BACnet often contain gateways linking BACnet devices to other legacy protocols such as LonTalk, Modbus, oBIX, [51, 76, 80] or one of many proprietary protocols used by legacy equipment. These protocols span the gamut of sophistication; Modbus is a popular serial protocol that provides only limited framing and a simple register-based data model on top of RS-485 links, while oBIX (the Open Building Information eXchange) is a set of XML schema managed by OASIS designed to promote building interoperability, at quite a high level. Although much could be said about these protocols, BACnet serves as a useful point in the design space. Proprietary protocols also see extensive, although diminishing use. Protocols like Siemens N2, Johnson N1 and N2, and many others are prevalent in buildings of a certain age; although the latest generation of system from large vendors have migrated towards BACnet, legacy systems often require an adaptor or emulator in order to communicate.

2.2.3 Component Modeling

Although protocols like BACnet begin to establish basic interoperability between building controllers, the semantic content of these interactions is still low. Building control protocols expose points which indicate analog or binary values, which may or may not be writable, but they are not tied to a higher level equipment or systems model of the building; it can be very difficult to tell what the point actually means and does. Within a BACnet system, points are often still identified by convention – Figure 2.6 has an example of a BACnet point name from Sutardja Dai Hall, a building on Berkeley's campus. The name is accessed by

reading the `name` property of a VAV controller. A lot is encoded in this string; however, it is by convention and clients or users must understand the conventions in use in the particular building they are in to correctly interpret the names.

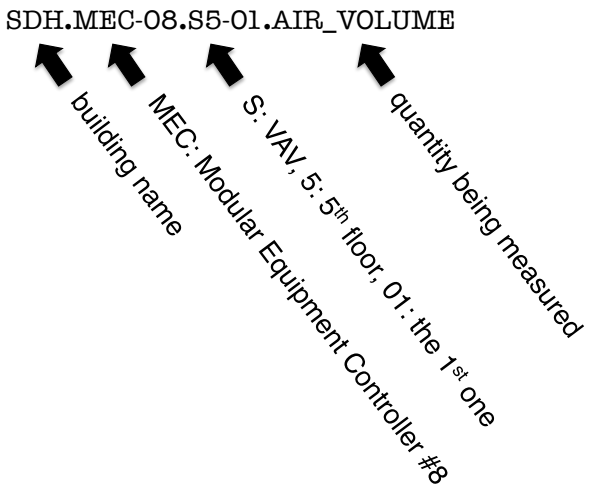


Figure 2.6: An example of a BACnet point name from Sutardja Dai Hall, on the Berkeley campus. In this case, the point name includes both spatial (building, floor, and zone) information, network information (the controllers' address), and functional information (air volume). Interpreting this requires knowing the convention in use.

Even knowing how to interpret these tags only begins to address the underlying issue here, which is the need for software in the building to be able to interpret the relationship between the components of the building. For instance, that tag name gives us no information about which process loops that VAV is part of, or how it relates to other components in the building. This problem of mapping a physical resources into separate views of the space begins to shift us from a controls perspective, focused on the actuation of various building systems, to Building Information Modeling, an offshoot of CAD focused on the digital representation of assets within the building. The goal of BIM is to maintain a digital representation of the entire building from construction through commissioning, allowing programatic inspection of a digital representation of the physical instance. The main set of standards and schema from the construction industry are the Industry Foundation Classes (IFC) and the related Extended Environments Markup Language (EEML) [34, 65]. IFC and EEML files allow architects and engineers to create features representing architectural elements, process loops, equipment, and other assets present within a building.

Alternative modeling languages are coming from the Green Building XML (gbXML) Consortium, mainly consisting of BIM developers. A major goal of this enabling interoperability between BIM packages and various energy modeling packages used to evaluate the energy use of the space once constructed. The major shortfall we find with most of these efforts is that they specify either too little or too much. Both IFC/EEML and gbXML specify

physical representations of spaces and components, but it's currently very difficult to link that model back to a set of controls governing the operation of the space, or match up the virtual model with actual data.

2.2.4 Interfaces for Programming

Programming these systems is a challenge, and how to do it parallels the two-tiered architecture. At the supervisory control level, it is possible to change set points and schedules through the use of one of the existing controls protocols such as BACnet. An implementer may make use of a wide range of BACnet (or other protocol-specific) features within a controller, and issue network commands so as to implement some control strategy. Most systems also have a way to change the logic present in the controllers themselves. Historically, these programming systems have come from the electrical and mechanical engineering communities, and have been based on relay ladder logic, rather than other alternatives. A significant problem with these systems is that they, for the most part do not make use of any of the advances in programming language design. Even basic structured programming constructs are often missing making it difficult to write reusable code, and making static analysis a very difficult problem.

2.2.5 Alternatives to SCADA

Although dominant in buildings as well as in other process industries, the two-tiered SCADA architecture is not the only attempt to provide coordinated management of distributed physical resources. One alternative is distributed object systems, as exemplified by implementations such as CORBA and Tridium Niagara [42, 108]. In a distributed object system, objects performing tasks can be accessed (mostly) transparently over the network, and applications are written by injecting new objects into the system, which call methods on other existing objects. These systems have had success in several domains; for instance, the U.S. Navy AGIES system uses CORBA internally for communication between radars, command and control elements, and weapons. In the building space, the most successful distributed object-based system is the Tridium Niagra architecture. The Niagra system defines a Java object model for many standard components in a building, and provides device integration functionality for interfacing with external devices using protocols like this discussed in Section 2.2.2. Applications access resources in the building through a global object namespace. This architecture is attractive because it mitigates some of the inherent problems with lower-level protocols, as objects must implement well-defined interfaces. Therefore, there is less semantic information lost when application requests are translated into low-level actions – for instance, an application may call a `lights_off` method in a well defined interface to turn the lights off, instead of writing a `0` to a vendor-specific register on a controller.

Another alternative architecture for distributed control systems are systems designed around message buses. A message bus implements point-to-multipoint communication between groups of senders and receivers; receivers (sometimes known as subscribers) receive

messages on a set of topics they subscribe to. This paradigm has seen some success in control systems, notable in CAN bus (Controller Area Network). Unlike a distributed-object system, a message bus system has looser binding between parties – a client can subscribe to a channel or send messages on a topic, without knowing exactly which or how many parties will receive the message. In some ways, BACnet inherits more for these message-bus based systems than from a true distributed object system.

Both of these systems are in some ways "flatter" than the SCADA architecture, with it's two-tier division of direct and supervisory control. This is in many ways liberating, because it frees applications from direct physical constraints on where they must run; as in the internet, they may theoretically run nearly anywhere. However, the separation of direct and supervisory control has important implications for reliability, and it is not obvious how the need to reason about which communication links and controllers are in the critical path of different applications maps into either distributed object or message systems.

2.3 Management and Optimization Applications

Existing building do, in a real way, run applications. The programs for controlling the HVAC and lighting systems described in Section 2.1 are one example; in many vendors' systems, they are implemented as a logical flow diagram, which are then synthesized into executable code and downloaded into the various controllers present in the system. Beyond this narrow scope however, most "programs" which are run in the context of the building either have hard separations between those meant for design-time analysis, optimization, or additional functionality. For instance, design tools such as EnergyPlus which are used for whole-building energy simulation require extensive model building and make assumptions about how the building will be operated, but these assumptions are not programmatically capture and transferred to the design of the control systems. Here, we briefly present a description of several classes of applications currently in wide use within buildings; the first class, energy analysis only makes use of stored data, while the second, demand response, also contains an element of control.

2.3.1 Electricity Consumption Analysis

Electricity usage across a large building is difficult to analyze. One of our testbed buildings on campus, Cory Hall, is typical: electricity enters in the basement through a 12.5kV main. Once it is in the building, it is stepped down to 480V at the building substation, from where it is distributed into 12 main circuits. These circuits are stepped down to 240V and 120V using additional transformers located in electrical closets throughout the building, from where they make their way to lighting, receptacles, computer server rooms, elevators, chillers, and other loads within the building.

As part of a monitoring project, we installed over 120 3-phase electrical meters into the building, and networked them together so as to enable centralized data collection. Figure

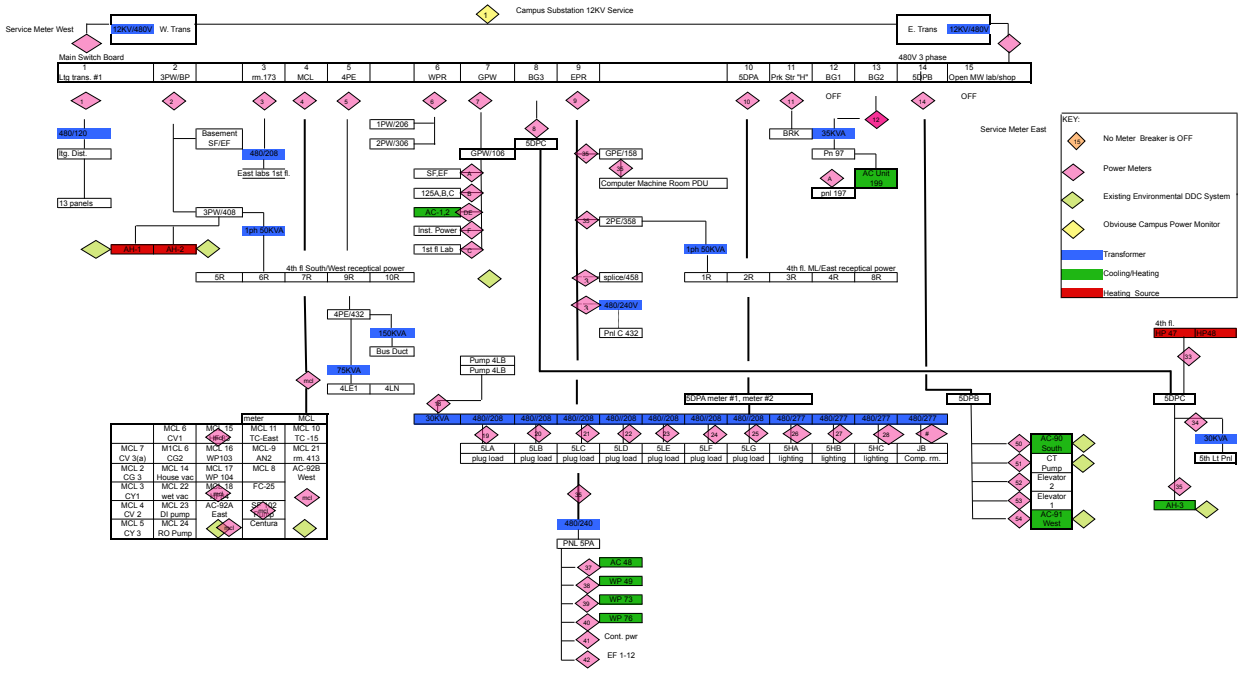


Figure 2.7: The electrical distribution system within Cory Hall, UC Berkeley. To better understand how electricity was used in the building, we installed around 120 three-phase electric meters at various points in the system. Analyzing this data requires both the ability to deal with larger quantities of data, and the metadata to allow automated interpretation.

2.7 shows where these meters were installed within the electrical distribution system in Cory Hall; in fact, this does not include the entire picture since individual end-uses, which make up around 40% of many buildings, are not individually monitored in this case. Making sense of the data generated by all of the metering elements present in this application requires multiple kinds of metadata, including information about the type of meter, where it sits within the electrical load tree, and which loads that branch serves. We will use this project as a running example throughout the thesis as we investigate the challenges of instrumenting, processing, and acting upon this data.

2.3.2 Energy Modeling and Analysis

Energy analysis is potentially performed at all phases of a building's lifecycle. Before construction, engineers analyze the expected energy consumption in view of expected weather, occupancy, and the building technologies in use. This type of analysis is used to optimize the building envelope and employs open-source tools like EnergyPlus [25] for simulating overall energy use, as well as packages like Radiance [110] for lighting simulation. In the design phase, the goal is normally to predict how the proposed design will perform once constructed,

as well as to act in a decision-support role to answer hypotheticals around the performance or economics of various architectural features, equipment sizing, and selection of components. Tools at this phase are relatively well developed; furthermore, at this stage, designers collaborate by sharing design files enumerating the geometry and physical properties of the materials.

2.3.3 Demand Responsive Energy Consumption

Demand response is the only existing class of building application that extends a control loop beyond the building itself. The underlying premise of demand response is to invert the traditional relationship between electricity generators and consumers in the electric grid; in the "legacy" electric grid, when loads begin consuming more power, generators must compensate by generating additional power. This leads to certain capital inefficiencies, since generation resources must be built to supply the the peak load experienced, even though average load is significantly less than that.

Demand response allows utilities to respond to increases in demand by asking certain loads to reduce their consumption, rather than simply continuing to produce more. The most prominent effort to standardize and deploy these approaches is OpenADR, developed at the Lawrence-Berkeley National Laboratory [89]. Fundamentally, the architecture and communication pattern in use; a central operator notes the need for a "demand-response event," and computes how much load they would like to shed. They communicate the need for the demand response event to an automation server, which then signals the loads integrated into the OpenADR system.

2.4 The Case for BOSS

Looking forward, several clear underlying trends drive the case for a unified programming environment for building applications. First, the sheer amount of sensing and actuation present in buildings has been increasing and it is increasingly networked, driven by the quest for both energy efficiency and increased quality of spaces, as well as the declining cost of computation and communication. Even a few years ago, the case for local control of systems was strong – networking was expensive, and Internet connections were unreliable or untrusted. Furthermore, storing and processing the data when extracted would have been difficult to imagine at scale. Secondly, many of the efficiency gains that building engineers seek will be enabled through *better analysis* of data from existing systems, and the ultimate *integration* of data from disparate systems. This trend is enabled by the ability to network everything. Finally, the ultimate goal is implementing *wide-scale feedback and control loops*, co-optimizing systems which today are nearly completely separate – allowing the building to interact with the electric grid, external analysis service providers, and occupant-facing applications in a way which is currently nearly impossible.

The goal of BOSS is to provide an architecture which enables experimentation with these new classes of applications, while taking building supervisory control to new places, where data inputs and actuators are spread broadly over physical and network expanses. The most significant design challenges involved in making this a reality are in rethinking *fault tolerance* for a distributed world, and examining device naming and semantic modeling so that the resulting applications are *portable*.

Chapter 3

BOSS Design

Our goal in designing BOSS is to provide a unified programming environment enabling coordinated control of many different resources within a building. We begin in this chapter by examining our requirements in some detail using a top-down approach. We extract from several applications common patterns used when writing applications for buildings. We then develop an overall architecture which allows these pattens to be simply and meaningfully implemented. Since an architecture is a decomposition of a large system into smaller pieces, with a principled placement of functionally and interconnection, we develop the key building blocks making up the BOSS architecture. This allows us to inspect the architecture at a high level, without needing to consider the detailed implementation of each component from the outset.

In succeeding chapters, we dive deeply into the design and implementation of three critical components of the BOSS architecture, using a similar methodology of reviewing patterns and use cases for that component alone, synthesizing from those uses a set of demands for our design, implementing, and then taking a step back to evaluate the result. In this way, we are able to consider both lower-level performance questions ("Is it fast enough?"), as well as address high-level architectural questions ("Does the composite system allow the implementation of the applications we were interested in?").

3.1 Design Patterns for Building Applications

When creating applications which relate to physical infrastructure in some way, there is typically a progression along the lines of "monitor-model-mitigate." This pipeline refers to an organization and operational paradigm supporting the goal of ultimate energy reduction and improved programability. Of course, this is not a single stage pipeline, but a cycle, with the results of the last mitigation effort ideally feeding into the next set of analyses and operation so that changes are made continuously based on an improved understanding of the system.

This three-phase pipeline has implications for how to design computer systems to support

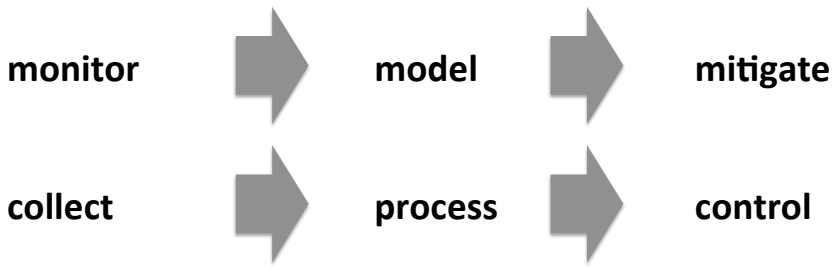


Figure 3.1: Organizations often follow a three-phase pipeline for the implementation energy efficiency strategies. In the first phase, they *monitor* systems to gain a better understanding of their operation and dynamics. Second, they create *models* to support decision making around which measures are most effective. Finally, they implement *mitigating measures* to reduce the energy spend.

this flow. Because the steps taken to understand and change building operation is staged and incremental, systems can be designed to parallel this workflow; as we gain a better understanding of the system through monitoring, we are then better equipped to build systems to support the modeling and control aspects of the process. The lower set of arrows in Figure 3.1 represent the active analogs to the monitor-model-mitigate paradigm: collect-process-control.

3.1.1 The Collect Pattern

The first phase of analysis of any large system is simply to gain visibility into its operation, though the collection of volumes of data. Although only limited changes are typically made to the system in this phase, it is not an entirely passive process. For instance, it may be necessary to install additional instrumentation, gain access to existing systems, reconfigure existing data collection infrastructure to collect or transmit more data, or to estimate unmonitored or unobservable values from collections of monitored through modeling.

Unless data collection is accomplished with an entirely new system, collecting data from an existing system typically requires a significant integration effort. In all cases, it is key to have a good understanding of what is to be monitored to inform the analysis process. Generally the first step is to decide on an initial list of points to collect data from. For instance, an energy usage study might require electrical meters to be installed at a circuit level granularity reporting kWh; a power-quality study might require additional data on power factor or harmonic content of the electricity.

Once the data required is known, we can decide how to obtain the data. Controls systems often offer multiple locations at which one can integrate. For instance, field-level devices might communicate over Modbus to a data concentrator, which then exposes that data using, say, OPC to an OSISoft PI historian. Many different architectures are possible

and different sets of points may be available through different systems, requiring multiple integrations. Practically, we need to consider overlapping considerations, including:

1. The availability of existing interfaces to the controls protocol,
2. The rate of data to be collected: existing upstream systems may collect data at a fixed rate, which may or may not be acceptable based on the integration requirements,
3. Whether actuation will be required: actuation may be only possible by integrating directly with field-level controllers rather than upstream data concentrators,
4. Number of integration points: it may be possible to get all required data from a single integration with a concentrator, as compared to integration with hundreds of field controllers. In general, fewer devices will be simpler,
5. Reliability: integration through a data concentrator which is not considered to be a critical piece of equipment will probably be less reliable than integration directly with the PLCs, as it introduces additional points of failure. Furthermore, high volume data collection may drive the concentrator out of its designed operating regime, and
6. Network access: the availability of a location to install software which can access both the controls system and the data collector.

Collecting data also requires a repository, or historian, to manage the data. Operational data is often in the form of time series, and specialized databases exist to support the collection of large volumes of data. Because the data is often repetitive or has low entropy, it compresses well allowing large amounts of it to be saved at relatively low cost. As part of the data collection process, the data as extracted from the system is transmitted either within the site or over a WAN to the ultimate data repository, where it can be analyzed and visualized.

3.1.2 The Process Pattern

The next step after collecting monitoring data is to begin to extract higher-level, meaningful events from the raw signal. The raw data in many cases will be of relatively high volume, but with a low information content. For instance, examining a raw feed of room temperature by hand may not reveal much about the dynamics of the system, but by correlating it with things known about the space a savvy investigator could uncover an HVAC system which is unable to meet demand during peak times: a malfunctioning chiller, opportunities for savings due to overcooling, or many other common problems or opportunities.

We have found three key requirements for working with building time series data at a large scale and that these are not well-served by existing systems. These requirements were developed from a study of existing and potential application for buildings: automated fault

detection and diagnosis [87, 94], personalized feedback and control for building occupants about their energy footprint [31, 62], demand response [88], and energy optimization [47, 74].

The first task of applications is simply locating the time series data in question using the time series metadata. Today, we have hundreds of thousands of streams in our system; this scale is relatively common-place in medium to large process control environments. In the future, the number of streams will only increase. To efficiently locate streams, one must make reference to an underlying schema or ontology which organizes the streams with reference to a model of the world. The issue is that no one schema is appropriate for all time series; data being extracted from a building management system (BMS) may reference a vendor-specific schema, while other time series may be organized by machine, system, or many other schema.

Time series data is array- or matrix-valued; sequences of scalar or vector valued readings. These readings contain a timestamp along with the raw data; they are often converted to sequences where the readings occur notionally at a fixed period for further processing through the use of interpolation or windowing operators. Because cleaning starts with the raw data series and produces (generally smaller) derivative series, a pipeline of processing operators (without any real branching) is appropriate.

3.1.3 The Control Pattern

In the final stage of the monitor-model-mitigate pipeline, the results of the data collection and modeling are put to use to inform changes to the operation of the building system. Traditionally in buildings, changing the control strategy is a relatively costly operation. There are several root causes; control sequences are hand-coded for each piece of each building, requiring significant manual effort to change. Underlying mechanical, regulatory, and human constraints on system operation are only implicit within the system, rather then made explicit; therefore, someone who attempts to change the operation has difficulty understanding the reasoning behind the current state of the system. Because no one is sure if a particular mode of operation was done intentionally to meet some requirement or accidentally, based on an oversight, justifying changes is difficult. Furthermore, because systems operate unattended for long periods of time, managers are hesitant to introduce changes which have uncertain effect on operations.

Altering or replacing control strategies can occur in differing levels of complexity. For instance, significant energy savings can often be obtained through supervisory changes: trimming operating hours, adjusting set-points within different control loops, or adjusting parameters underlying control loops' operation. More involved changes require replacing direct control loops with new logic: replacing a PID controller with a model-predictive controller, coupling the operation of previously decoupled loops, or accounting for additional variables in system operation. The two types of modifications have different requirements on the underlying system, in terms of how close to real-time they operate and the volume of data required, and have different implications in terms of system reliability.

Therefore, the two most important requirements for implementing control broadly are that of *reliability* and *portability*. Reliability is driven by considerations around meeting all of the constraints which have been engineered into the system, without imposing additional constraints due to poor management. Portability refers to the ability to repeat pieces of code and control sequences in many places, gaining from the economy of scale and eliminating the need for custom programming in what are fundamentally highly parallel systems.

3.2 BOSS Design: a Functional Decomposition

Based on experience with proceeding through the three stages of monitoring, modeling, and mitigation, we concluded that better abstractions and shared services would admit faster, easier, and richer application development, as well as a more fault tolerant system. Moreover, one needs to consider issues of privacy and controlled access to data and actuators, and more broadly provide mechanisms that provide isolation and fault tolerance in an environment where there may be many applications running on the same physical resources.

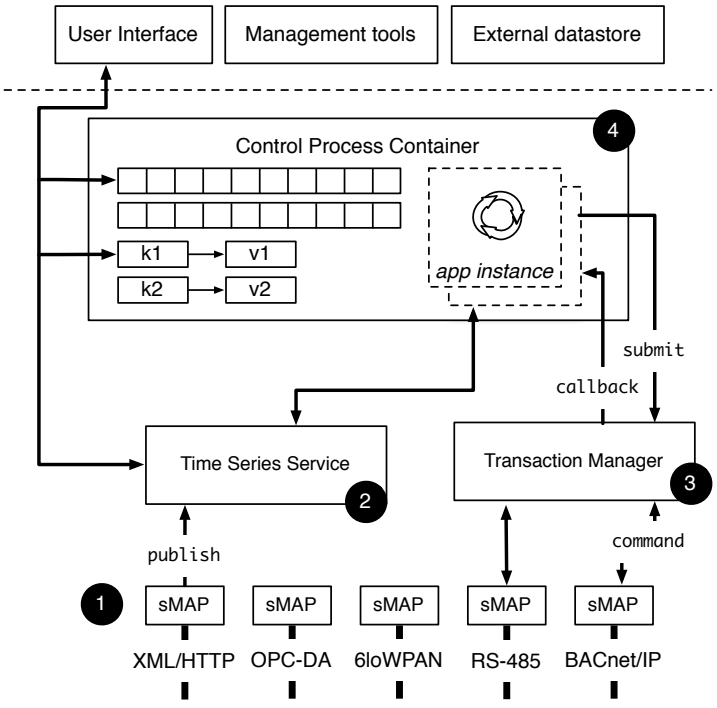


Figure 3.2: A schematic of important pieces in the system. BOSS consists of (1) the hardware presentation layer, the (2) time series service, and the (3) control transaction component. Finally, (4) control processes consume these services in order make changes to building operator.

The BOSS architecture has been developed to meet the needs of an organization as it proceeds through the stages of monitoring, modeling, and mitigation. It consists of six main subsystems: (1) hardware presentation; (2) real-time time series processing and archiving; (3) a control-transaction system; and (4) containers for running applications. Additionally, authorization, and naming and semantic modeling are present in the design. A high-level is shown in Figure 3.2 and described in detail below. The hardware presentation layer elevates underlying sensors and actuators to a shared, RESTful service and places all data within a shared global namespace, while the semantic modeling system allows for the description of relationships between the underlying sensors, actuators, and equipment. The time series processing system provides real-time access to all underlying sensor data, stored historical data, and common analytical operators for cleaning and processing the data. The control-transaction layer defines a robust interface for external processes wishing to control the system which is tolerant of failure and applies security policies. Lastly, "user processes" make up the application layer.

Execution in this architecture is distributed across three conceptual domains: the lowest, the sensor and actuator plane, building-level controllers, and Internet services. One purpose of distinguishing these domains is not because of a difference in capability (although there are surely huge differences), but rather because we wish to allow these to be reasoned about in terms of the implications of a failure; we call them "failure domains." For instance, a failure of the network connecting a floor-level panel to other building controls does not compromise that panel's ability to actuate based on the directly-connected inputs and outputs, but it does prevent it from contacting an Internet service for instructions on what to do. The tolerance of a particular control loop to failures can be determined by examining which data are needed as inputs, and from which fault boundaries they cross.

3.2.1 Hardware Presentation

At the lowest level of the hardware interface stack is a Hardware Presentation Layer (HPL). The HPL hides the complexity and diversity of the underlying devices and communications protocols and presents hardware capabilities through a uniform, self-describing interface. Building systems contain a huge number of specialized sensors, actuators, communications links, and controller architectures. A significant challenge is overcoming this heterogeneity and providing uniform access to these resources and mapping them into corresponding virtual representations of underlying physical hardware. It abstracts all sensing and actuation by mapping each individual sensor or actuator into a *point*: for instance, the temperature readings from a thermostat would be one sense point, while the damper position in a duct would be represented by an actuation point. These points produce *time series*, or streams, consisting of a timestamped sequence of readings of the current value of that point. The HPL provides a small set of common services for each sense and actuation point: the ability to read and write the point; the ability to subscribe to changes or receive periodic notifications about the point's value, and the ability to include simple key-value structured metadata describing the point.

Providing the right level of abstraction (for efficiency) while representing many different types of legacy devices is the key tradeoff. Systems we integrate with are constrained in many different ways; Modbus [76] provides only extremely simple master/slave polling, while some `6lowpan` wireless systems are heavily bandwidth-constrained. Some systems provide highly-functional interfaces (for instance, OPC [85] or BACnet [4]), but implement it incompletely or utilize proprietary extensions. Although nothing in our design prevents devices from implementing our HPL natively, in most cases today it is implemented as a proxy service. This provides a place for dealing with the idiosyncrasies of legacy devices while also providing a clean path forward.

To provide the right building blocks for higher level functionality, it's important to include specific functionality in this layer:

Naming: each sense or actuation point is named with a single, globally unique identifier. This provides canonical names for all data generated by that point for higher layers to use.

Metadata: most traditional protocols have limited or no metadata included about themselves, or their installation; however metadata, is important for the interpretation of data. The HPL allows us to include metadata describing the data being collected to consumers.

Buffering: many sources of data have the capability to buffer data for a period of time in case of the failure of the consumer; the HPL uses this to guard against missing data wherever possible.

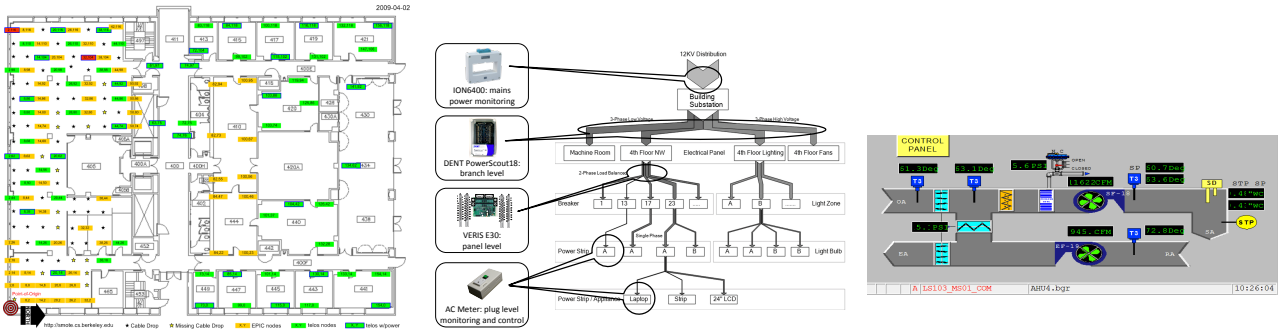
Discovery and Aggregation: sensors and their associated computing resources are often physically distributed with low-powered hardware. To support scalability, the HPL provides a mechanism to discover and aggregate many sensors into a single source on a platform with more resources.

This functionality is distributed across the computing resources closest to each sensor and actuator; ideally it is implemented natively by each device.

3.2.2 Hardware Abstraction

The hardware abstraction layer is responsible for mapping low-level points from the HPL into objects with higher-level semantics and functionality, and providing a way to discover and reference these objects. Unlike computer systems, buildings are nearly always custom-designed with unique architecture, layout, mechanical and electrical systems, and control logic adapted to occupancy and local weather expectations. This introduces challenges for writing portable software, because the operation of the system depends not only on the exact piece of equipment being controlled, but its relationship to numbers site- and installation-specific factors. The current state-of-the-practice is for software to be manually configured for each piece of equipment. For instance, every zone temperature controller may use a

standard control algorithm, which is manually "wired" to the sense and actuation points in this zone. Changing this requires manual intervention in every zone, and the replacement of the standard control algorithm would require even more significant and costly modifications.



(a) spatial view (b) electrical view (c) system (HVAC) view

Figure 3.3: Multiple different views of the relationships between system components exist and interact in physical space. The HAL allows applications to be written in terms of these relationships rather than low-level point names.

The solution is to allow applications to inspect these relationships between components and alter their operation accordingly. An important concept in the HAL is the idea that there are multiple, separate "views" of the underlying building systems: a *spatial* view deals with where objects are located in three-dimensional space, while a *systems* view records the relationship between elements of a common system. An *operational* view deals with issues like class schedules and room utilizations, while an *electrical* view deals the electrical distribution tree; more views are possible to define. Furthermore, these views overlap through the physical world: for instance, a chilled-water pump's relationship to other components is part of the systems view, while its location in space is part of the physical view and in the electrical distribution tree the electrical view as shown in Figure 3.3.

The HAL maintains representations of these views, and provides an approximate query language, allowing authors to describe the particular sensor or actuator that the application requires based on that component's relationship to other items in the building, rather than hardcoding a name or tag. Applications can be written in terms of high level queries such as "lights in room 410," rather than needing the exact network address of that point. The query language allows authors to search through multiple views of underlying building systems, including *spatial*, where objects are located in three-dimensional space; *electrical*, describing the electrical distribution tree; *HVAC*, describing how the mechanical systems interact; and *lighting*.

A second problem is that building components often do not provide standardized methods for controlling them. Different pieces of equipment require different control sequences to achieve essentially the same result. To abstract these differences, the HAL also allows us to create higher-level objects on top of the low-level HPL points and the relationships between

them. The HAL consists of a set of drivers with standard interfaces for controlling common building components, such as pumps, fans, dampers, chillers, *etc.* Drivers provide high-level methods such as `set_speed` and `set_temperature` which are implemented using command sequences and control loops on the relevant HPL points. These drivers provide a place to implement device-specific logic that is needed present a shared upper abstraction on top of eclectic hardware systems.

Drivers and applications also use the view functionality to determine locking sets, necessary for coexisting with other applications. For instance, an application which is characterizing the air handler behavior might want to ensure that the default control policy is in use, while varying a single input. It would use the approximate query language to find all points on the air handler and lock them in the transaction manager. Since different models of the same piece of equipment may have different points even though they perform the same function, it is essential that applications can control sharing at the level of functional component rather than raw point name.

3.2.3 Time Series Data

Most sensors and embedded devices do not have the ability to store large quantities of historical data nor the processing resources to make use of them; such data are extremely important for historical analyses, model training, fault detection, and visualization. The challenge is storing large quantities of this data efficiently, while allowing applications to make the best use of it; most historical data currently is usually unused because it is difficult to extract value from. Typical access patterns for historical data are also different than those that traditional relational databases are optimized for: data are mostly accessed either by performing range-queries over timestamps and streams or finding the latest values; for instance a typical query might extract all room light-level readings for a period of one month, touching millions of values. Even a modest-sized installation will easily have tens of billions of readings stored and even simple queries have the potential to touch millions of readings. New data are mostly appended to the end of the time series, because the data come from individual measurements taken by the sensors and published by the HPL. Finally, the data are usually dirty, often having desynchronized timestamps requiring outlier detection and recalibration before use.

The time series service (TSS) provides the application interface for accessing stored data, specifically designed to address these concerns. It consists of two parts: a stream selection language and a data transformation language. Using the stream selection language, applications can inspect and retrieve metadata about time series; the data-transformation language allows clients to apply a pipeline of operators to the retrieved data to perform common data-cleaning operations. This both moves common yet complex processing logic out of the applications, allowing them to focus on making the best use of the data, and also enables the possibility of optimizing common access patterns.

This service moves two important shared capabilities into a service, which are shared by application at each stage of the monitor-model-mitigate cycle:

Cleaning: data often contains errors and holes, causing analysts and applications to painstakingly clean the data; removing outliers, fixing timestamps, and filling missing data. Although many well-known techniques exists to perform these techniques, they are often employed manually, limiting sharing between applications. We define *data cleaning* as the task of applying appropriate, sensor- and application-specific operations to the raw data so as to produce the normalized, corrected dataset needed for higher-level analysis.

Streaming: although Stonebraker clearly identifies the need to treat streaming and historical data uniformly [100], this idea is not present in most building data systems. Not doing so creates significant friction when moving from evaluation on stored data to production. Streaming computation fed into an actuator can also be thought of as forming a control loop, forming a CPS.

3.2.4 Transaction Manager

The final stage of improving a facilities' operation is making changes to the control strategies in use. This active part of the cycle is performed by applications. BOSS applications typically take the form of either coordinating control between multiple resources, which would otherwise operate independently as in the HVAC optimization, or extending control beyond the building to incorporate other systems or data, as in the personalized control application. The challenge is doing so in a way that is expressive enough to implement innovative new control algorithms, while also is robust in the face of the failure of network elements and controllers; existing building control systems are not designed to be extended in this way. Control algorithms that involve users or Internet-based data feeds should survive the failure of the parts of the control loop that run outside of the building without leaving any building equipment in an uncertain state. It is desirable that control policies be extended or modified across multiple failure domains; Internet-based services may have more processing and storage than is available in the building or may wish to implement proprietary logic. However there are real problems with performing automatic direct or supervisory control over the Internet or the building network. For direct control, latency may be an issue. There can be concurrency issues when changes are made by multiple parties. Furthermore, a failure of one domain can leave the system in an uncertain state. To resolve these issues, we use a *transaction* metaphor for effecting changes to control state. Transactions in database systems are a way of reasoning about the guarantees made when modifying complex objects. In this control system, we use transactions as a way of reasoning about what happens when collections of control inputs are made.

Control transactions operate conceptually at the supervisory level, but expose significantly richer semantics than simple set-point adjustment. A control transaction consists of a set of actions to be taken at a particular time; for instance, a coordinated write to multiple set points. In addition to the action to be performed, a control transaction also requires a *lease time* during which the control action is to be valid, and a *revert sequence* specifying how

to undo the action. When the lease expires either because it is not renewed or some failure occurs, the transaction manager will execute the revert sequence, which restores control of the system to the next scheduled direct controller. The ability to revert transactions provides the fundamental building block for allowing us to turn control of the building over to less-trusted applications. These segments of revert actions are implemented for each action and can be thought of as the "inverse action" which undoes the control input. We require there to be a "lowest common denominator" control loop present which is able to run the building in default (although potentially inefficient) way. In this way, applications can always simply release control and move the building back into default control regime.

To support multiple applications, each operation also is associated with a *priority level* and a *locking strategy*. These allow multiple higher-level processes or drivers to access the underlying points, while providing a mechanism for implicit coordination. Using a concept borrowed from BACnet, writes are performed into a "priority array" – a set of values which have been written to the point at each priority level. The actual output value is determined by taking the highest priority write. Although it provides for basic multiprocessing, the BACnet scheme has several problems. Because there are no leases, a crashing application can leave the system locked in an uncertain state until its writes are manually cleared. There are also no notifications, making it difficult to determine if a particular write has been preempted by another process at a higher priority without periodically polling the array. The transaction manager augments the basic prioritization scheme, adding *leases*, *notifications*, and *locks*. Leased writes cause the transaction manager to undo writes once the lease has expired, protecting against the case of network or application failure. Notifications provide applications information about when changes they are made have been overridden by other applications, while locking allows higher-priority actions to receive exclusive access.

3.2.5 Authorization and Safety

In addition to providing high-level, expressive access to building systems, BOSS seeks to limit the ability of applications to manipulate the physical plant in order to ensure safety. Most building operators will not turn over control to just anyone, and even for a trusted application developer, safeguards against runaway behavior are needed. The authorization service provides a means of authorizing principals to perform actions; they may be restricted by location (only lights on the fourth floor), value (cannot dim the lights below 50%), or schedule (access is only provided at night). Because it is also difficult to foresee all possible interactions between control strategies, BOSS also provides for run-time safety checks. By observing the real-time data from the system maintaining models of system state, it can attempt to determine if the system is being pushed in an unsafe or impermissible direction.

Authorization

The authorization system proactively enforces access constants on control input to the system. We provide access controls at the same semantic level as the operations to be performed.

Therefore, access may be restricted both at the driver interfaces which are part of BOSS, as well as at the level of bare HPL points. Before an application begins to run on the system, it must submit a manifest to the authorization server, which lists which points or classes of points it wishes to access. As with other parts of the system like the driver interface, these points may be specified either explicitly by unique name, or implicitly through the use of a HAL query. Attached to each resource request is an associated contract the application agrees to follow; generally, lists of points that the application will read, and input range limits and schedules for points it will write.

These application manifests may either be automatically approved, or presented to a building manager for approval. As part of a manual review, the building manager may further restrict the access granted to the application; restricting the range of input the building might accept, or the scale of the rollout. Once the manifest is approved, the application is free to use the access granted to it as specified in the manifest. Security and safety checks are performed at time-of-use on each BOSS method call; although clients may cache permissions granted by the authorization server for short periods in order avoid excessive round-trips.

Verifying access permissions at time-of-use using an online server rather than at time-of-issue using signed capabilities has implications for availability and scalability, as it places the authorization service on the critical path of all application actions. However, we found the ability to provide definitive revocation a critical functionality necessary to convince building managers that our system is safe. This is one place where practical considerations of the domain won over our bias against adding more complexity to the command pathway. Because we only cache application permissions for a (configurable) short time, we are able to guarantee that certain permissions can be revoked at run-time.

Safety

Another compelling difference between computer systems and physical systems is that in a physical system, different inputs may interact in a way which is difficult or impossible to foresee, but drives the system into an undesired state. This is because the inputs interact in physical space, beyond the control and visibility of the management system. For instance, a lighting controller may reduce lighting because it is a sunny day, but if it doesn't coordinate its lighting control with the behavior of shades the result may be darkness. Therefore, it is also important to have the ability to identify when the system is moving towards an undesired state and correct. The reversion system provided by the transaction manager, as well as the real-time data access from the time series service provide the functionality needed to build in active safety.

To provide this "active" safety, we provide for special, privileged watcher processes that have the ability to override applications' control of the system, and return control to default, assumed-good control strategies. Because the system is governed by physical laws, it is possible to maintain a model of the system state, and abort offending applications when they exceed limits or push the system outside of safe limits. Watcher processes observe real-time data from system operation, and use it to inform their model, which they compare

against various constraints. When the system is in danger of violating a constraint and behaving in an unforeseen way, the watcher must choose what action to take; essentially, which transactions to abort to return the system to a safe state. Aborting the transaction returns control to the next-lower-priority controller and ultimately back to the building's default control scheme.

As a simple example, a HVAC zone watcher maintains a model of the thermal flows into and out of a heating and cooling zone. The heating or cooling inputs to the zone are caused by one controllable source (the building's HVAC system), and multiple uncontrollable sources (occupants, equipments, weather). Based on historical data, the watcher can also infer models for the system's capacity to heat or cool the system, and the maximum expected change in heating load as a function of occupants and weather. Using these models, the watcher can calculate in real-time if the zone is on a trajectory to violate a manager-provided temperature constraint, and abort any transactions which are controlling that zone if it is.

3.2.6 Building Applications

Updates to the building control state are made atomically using control transactions; however, these are often part of larger, more complicated and long-lived blocks of logic. This is known as a "control process" (CP) and is analogous to a user process. CPs connect to services they require, such as the time series service, HAL, and transaction managers, and manage the input of a new control action. Because of the careful design of transactions and the archiver, there are few restraints on where CPs can be placed in the computing infrastructure; if they fail or become partitioned from the actuator they control, the transaction manager will simply "roll back" their changes and revert to a different CP which has not experienced partition or failure.

A control process also offers the point at which authentication and authorization is performed in the system. To receive authorization, a CP is required to provide a manifest file describing the types of changes it will make to the system – what points will be controlled, within which values, and when. For some users, we then require administrative approval of these manifests before granting them; once granted we provide a signed version of the manifest to the CP. This serves as a capability which can be provided to other parties proving that the CP has been authorized to make a control action.

Control processes run application-provided logic within a container managed by the BOSS runtime. The application container manages the control process lifecycle, instantiating new processes when needed, and tracking resources utilized by the code. The process container also mediates the application's access the outside world, allowing external parties to communicate with a running application through message-passing, as well as for the application to externalize state about its operation through key-value pairs published to an application namespace. This design allows application writers to encapsulate long-lived control logic into a concise, isolated piece of logic with well-defined failure semantics and a narrow interface to the outside world.

3.3 Perspectives

Having developed the architecture in detail, we now briefly consider the overall service composition from a few different perspectives.

3.3.1 Runtime Service Partitioning and Scaling

At runtime, all of the services developed in Section 3.2 must be mapped to physical computing resources, either within the building or remotely, accessed via a network connection. The assignment of these services has important implications for the availability of the system as a whole; Table 3.1 begins to summarize placement restrictions on these services.

Architectural component	Functional requirements	Placement
Hardware presentation layer	Expose the primitive low-level operations of hardware using a common set of interfaces.	Positioned as close to the physical sensors and actuators as possible (ideally, co-locate with transducers).
Control transaction manager	Provide "all or nothing" semantics when applying control inputs; provide rollback of actions on failure, cancellation, or expiration.	In the same failure domain as the HAL used to affect the changes.
Hardware abstraction layer	Map the low-level functions of the physical hardware to higher level abstractions.	Anywhere; should be persistent.
Time series service	Maintain a history of readings from the sensors and actuators.	Replicated; may be offsite.
Authorization service	Approve application requests for access to building resources.	Anywhere
Control processes	"User processes" implementing custom control logic.	Anywhere

Table 3.1: Architectural components of a Building Operating System

The HPL and control transaction manager are the most constrained; they must remain in communication with the sensors and actuators to which they are proving access. If partitioned from the underlying hardware, the system could be left in an uncertain state, where a transactional command was aborted by the transition manager but lost before reaching the actual actuator resulting in partial execution of the transaction. The servers making up the HPL therefore must be distributed to be as close to the underlying points they expose as possible. The transaction manger communicates with these resources, and therefore sits within the same fault domain; if its connectivity to the underlying resources is severed, it will be forced to abort transactions and recover.

Other components have significantly more flexibility as to their placement; their failure or partition will affect the availability of the system, but will not impact correctness. They may be replicated so as to increase availability, or provide additional scalability. For instance, the time series service may be required by various applications for model building or visualization, but may have multiple instantiations, with one service on-site collecting a smaller amount of data, used for simple model building and a large, more scalable service placed offsite and used for data aggregation and wide-scale analysis.

Control processes in particular are designed to have the most flexibility as to placement; while partition from any of the services they use may cause them to fail, the underlying system is prepared to deal with this eventuality. Because they will depend on external data in many cases; for instance, a connection to a user-feedback web site, or access to a satellite data feed for additional data, we are unable to make strong assumptions about their availability.

3.3.2 Reliability

Multiple different layers contain mechanisms to deal with network partition and intermittent failure. The hardware presentation layer is designed to allow other components of the system to reliably collect data from the underling sensor and actuators by providing a reliable buffer for outgoing data. It also forms the basis for the leases provided by the transaction manager by implementing simple timeouts on writes. Control processes can submit batched actions or participate in control loops through the transaction manager, while maintaining the guarantee that their actions will be undone in the case of application failure, network partition, or other eventuality.

3.3.3 Portability

Application portability from system to system is aided by a few different features of the design. By mapping low-level control points to a uniform namespace at the presentation layer, applications which refer to points by referring to an underlying set of tags rather than a controller's network address can be moved from building to building, provided that a consistent schema has been applied. The hardware presentation layer also makes the applications agnostic to the peculiarities of any underlying protocols, since they are elevated to a uniform service type. Finally, the transaction manager reliability model relaxes constraints on where applications need to be hosted, making it potentially practical to host controllers in the Internet or on mobile devices.

3.4 Next Steps

Having developed the BOSS architecture, we now proceed with a detailed discussion of three key components: the hardware presentation layer, the system for time series processing,

and the transaction manager. We are able to extensively evaluate tradeoffs in each of these components because our systems have seen extensive deployment both by us and by external users. Each component is useful on its own without the rest of the architecture, which allows us to evaluate each somewhat independently before bringing them back together and showing the composite system performance; we also feel that this is good evidence of a well-factored design.

Specifically, we do not present several of the other key elements of a BOSS: in particular, we leave treatment of the hardware abstraction layer, known as the Building Application Stack to other work[63], and limit the implementation and evaluation of the security and authorization service to future work.

Chapter 4

Hardware Presentation

The hardware presentation system is responsible for elevating the many arcane industrial protocols to a common point, which allows rapid application development despite the huge heterogeneity which exists at lower layers. It fits neatly into the "measure" part of the three-stage model of system analysis. In order to ground this system's design in concrete applications and use cases, we first review some of our deployments with an eye towards extracting the underlying systems requirements. The result of this design and implementation work has been a protocol called sMAP: the Simple Measurement and Actuation Profile.

In its final form, sMAP has become a useful system on its own, allowing implementors to quickly and easily write data sources that obtain data from instruments and reliably and efficiently publish that data over the Internet. In various incarnations, it has been scaled up to publish hundreds of readings per second, while other implementations have been squeezed to fit on embedded devices. sMAP is designed with these use cases in mind as a result of a careful design process which considered a broad range of uses from the beginning, and constant iteration on the design starting in 2010. The result is a mature system which has seen external adoption, including by other groups at Berkeley, several companies, and the national laboratories.

We use several different deployments of sMAP as our motivation, and as a basis for evaluation. They include:

Residential plug-load deployment: we deployed 455 plug-load meters across four floors of an office building at the Lawrence Berkeley National Laboratory, and around 50 meters to each of four different residences to account for usage by plug-in appliances.

Personal Comfort Tool: the Center for the Built Environment developed a portable sensing cart collecting many different pieces of data relating to personal comfort.

Sutardja Dai Hall: using BACnet, we imported around 2500 different time series from the Center for Information Technology Research the Interest of Society (CITRIS) building.

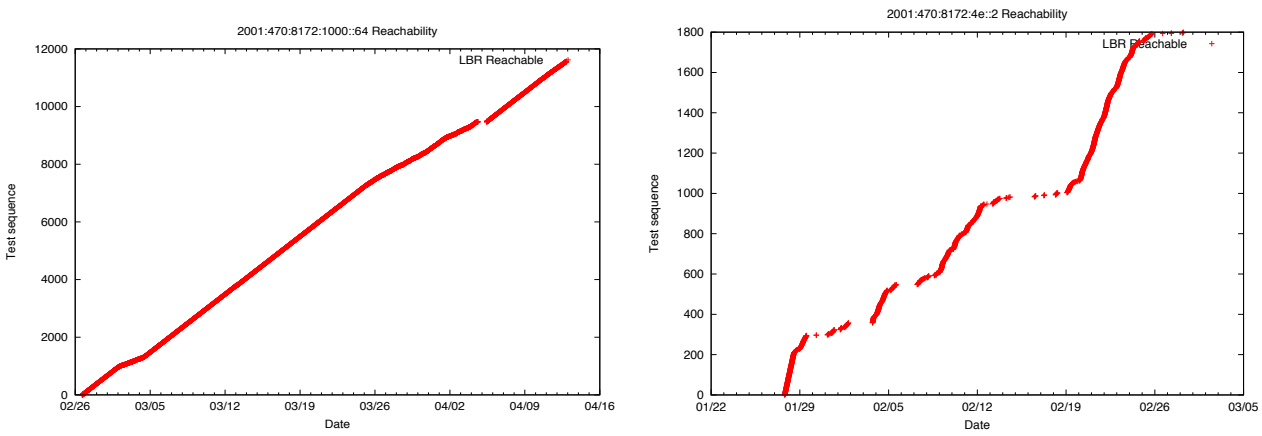
Cory Hall Monitoring: we installed sub-metering elements on around 120 circuits inside of the Electrical Engineering building at Berkeley.

Berkeley Campus: we imported whole-building energy consumption information from a preexisting project collecting energy usage data from across the campus.

4.1 Design Motivation

4.1.1 Residential Deployment

As part of the Miscellaneous Electrical Loads (MELS) monitoring project collaboration with LBNL between 2009 and 2011 we conducted a number of residential and commercial pilot studies using the ACme plug-load metering system. A significant challenge in this environment is the fact that home Internet connections can be very unreliable; As Figure 4.1 shows, in one deployment, external connectivity was available for only a few hours a day, on average. This implies that, despite a need for real-time data in many cases, sMAP should also support disconnected and intermittent operation. Since at the time, sMAP did not support any sort of reliability beyond standard network-layer retries from TCP, we were forced to build around this limitation in the application. This is both redundant and error-prone. This is particularly relevant for IP-based sensors; even though they may be globally routable allowing for a direct connection between the sensor and collector, it is likely that frequent outages will be experienced on that path.



(a) A home router with an adequate uplink

(b) A home router with poor connectivity

Figure 4.1: The results of five-minute external connectivity tests over a period of weeks for two residential deployments. A connection with no problems would be shown as a straight line.

In another application, the Center for the Built Environment (CBE) developed a set of tools for characterizing indoor environment. This includes a cart which collects data on room air temperature stratification and reports it to a backend. In order to avoid requiring

on-site infrastructure support the cart uses a cellular modem to connect to the Internet. Cellular internet connections are also notoriously unreliable, as well as exhibiting dynamic behavior in terms of both throughput and latency [48, 113]. Therefore, in addition to being tolerant to interruptionss it should additionally support efficient publication of data over links with limited bandwidth; even a residential DSL line can be saturated relatively easily as the number of sensors increases without some attention to efficiency.

> **Takeaway:** both *local buffering* of data as well as *transport efficiency* should be priorities in order to account for intermittent, slow links.

4.1.2 Building Management System Integration

Much building data and actuation is currently locked up inside of existing building management systems. These systems often are silos, with the vendor providing all equipment and software as a single integrated solution. The problem with such a solution is that the existing architecture imposes constraints on the ways the system can be accessed or programmed: very often, data from sensors and actuators enter the system at the bottom, and emerge at the top as the final products which the manufacturer foresaw would be required; intermediate layers are often either arcane or completely inaccessible. The HPL must break open these closed systems, and provide a uniform interface for upper layers to build on.

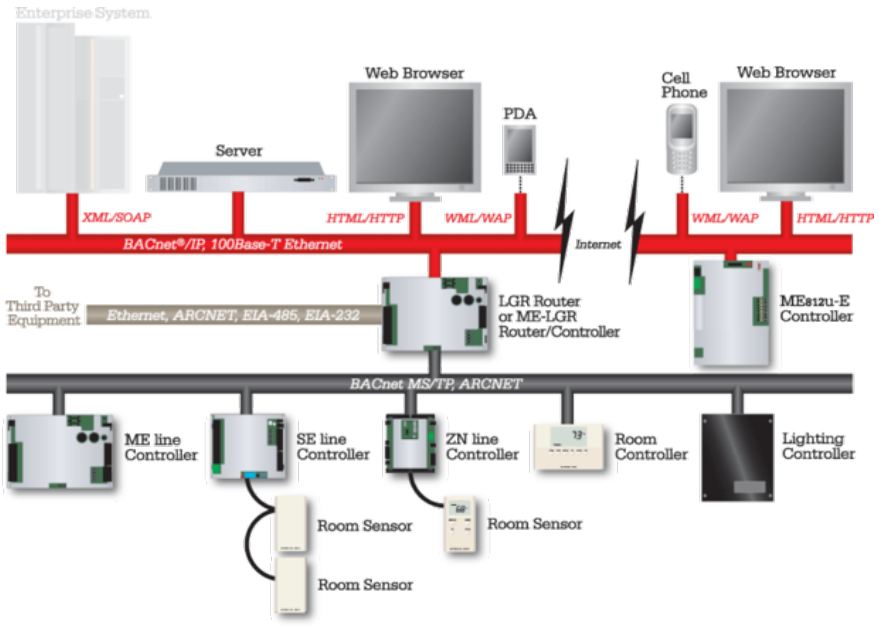


Figure 4.2: The system architecture of an Automated Logic building management system. [24]

Figure 4.2 shows the architecture of one common vendor's BMS product. Much as in our schematic representation in Figure 2.3, the system has a field-level bus with specialized physical and link layers over which controllers communicate, as well as head-end nodes which perform management functions. In this case, the controllers communicate using an open standard, the BACnet protocol over an ARCnet (a token-ring serial protocol) link-layer.

Location	Vendor	Controls protocol	Number of points	Data rate
Sutardja Dai Hall	Siemens	BACnet	1400	Up to every 2s
Bancroft Library	Automated Logic	BACnet	300	20s
David Brower Center	Schneider Electric	BACnet	2000	1 minute
U.C. Davis Campus	Schneider Electric	OPC	20	1 minute
New York Times Building	Unknown	CSV import	400	15 minute

Table 4.1: Building management system and data concentrator integrations performed.

When integrating such systems, there are essentially two challenges. The first is simply data rate; to an extent greater than many other existing systems, BMS systems have the capacity to generate data at thousands of points per second. Moreover, the systems are in some cases themselves bandwidth limited, depending on the communication technology used for the controller bus. While some newer buildings use Ethernet, many older building communicate using a serial protocol, such as RS-485, that imposes bandwidth constraints. sMAP must be able to efficiently collect and transmit this volume of data to external collectors, while being sensitive to the limitations of the underlying system.

The second challenge is incorporating actuation. Existing controls protocols like BACnet or OPC have specific functionality to allow them to perform actuation, and expose a somewhat richer interface than just reads and writes of individual actuation positions. Since we are designing a protocol translation layer, we should be able to expose the underlying functionality when appropriate, so that the system composed of the original BMS and the protocol translation is still reliable.

Takeaways: Data publishing at *high aggregate rates* in the range of thousands of points a second should be possible. When *actuation* is present, the capability should be exposed in a way that preserves the overall system reliability.

4.1.3 Building Retrofit

As part of the California Electric Commission (CEC) project to develop a testbed for building-to-grid integration, we conducted an in-depth instrumentation retrofit of an existing building: Cory Hall. Cory Hall, first opened in 1955 is the home of the electrical engineering department and contained a mix of laboratory, classroom, and micro fabrication spaces at the time of the retrofit. To better understand the building operation, we added a signification amount of instrumentation to the building, shown in Table 4.2.

Meter type	Communication	Number installed
3-phase electric	Modbus	65
Whole-building electric	XML/HTTP	1
ACme plug-load	`6loWPAN`/IPv6	20
Steam	Modbus	1
Condensate	Modbus	1
Weather	SDI-12 (serial)	1
Thermostats	HTTP	1

Table 4.2: Instrumentation added to Cory Hall as part of the Building-to-Grid testbed project.

Dealing with this type of data stresses the system in several ways. Clearly one challenge is simply integrating data using a large number of separate protocols. Even though several of the devices used in this project communicate using Modbus, it is a very low-level protocol and provides no data model. It not self-describing and therefore requires separate mapping layers to be written for each device, converting register-level representations into a more useful high level format. Therefore, providing an extremely simple framework for integrating new instrumentation sources is desirable.

Secondly, this deployment contains a large number of three-phase electric meters. Electric sub metering is a good example how placing measurements in context requires multiple different views of the infrastructure. These meters are all related to each other through the electrical load tree, pictured in Figure 2.7. Electricity flows into the building from the 12.5kV grid feed into a large step-down transformer, which reduces the voltage to 480V and distributes that through 12 sub-circuits to loads within the building. The electric meters are installed at different points within this distribution tree in order to disaggreate the total energy consumed by the building into smaller chunks. Capturing the point in the tree at which these meters are installed is essential for most analyses of the data, and some information is available at the HAL layer which can aid in interpretations of the data.

Takeaways: As with the building management system case, *high aggregate data rates* are often possible. Furthermore, *metadata* should be represented in a way which allows consumers to make sense of the multiple different views of the building.

4.1.4 External Data

Another common use case has been integration or importing pre-existing databases. For instance, the `obvius.com` site contains whole-building energy data from much of the UC campus from the past several years; `wunderground.com` provides access to many years' worth of historical weather data. It is typical for implementors to first acquire real-time access to the data, and then obtain access to a database which contains historical readings. Many existing databases contain relatively large volumes of stored readings, and so efficient import of these data is an important use case.

Takeaways: Some data will be *bulk loaded* and therefore should not require an on-line server; furthermore, this bulk-load case should be efficient and differs from the other cases where data is trickled out as it is generated.

4.2 The Simple Measurement and Actuation Profile

The presentation layer allows higher layers to retrieve data and command actuators in a uniform way. The realization of the hardware presentation layer is called the Simple Measurement and Actuation Profile (sMAP), which has included two major versions and many minor revisions starting in 2009. It provides resource-oriented access to data sources and actuators, includes support for many different device types, and has seen adoption in many other groups.

sMAP's design is the result of multiple iterations and significant field experience, including that detailed in Section 4.1 and as a result includes features needed to satisfy various deployment requirements; however, we have also made it simple for novice programmers to produce high-quality, reusable code. Some of its key features are

Metrology : support for large number of relatively high-rate time series, with structured metadata.

Syndication: simple integration into a larger system of time series processing and archiving, with built-in compression and local buffering of data when connection quality is poor.

Breadth: a library of existing drivers for many different types of existing devices, and built-in support for integrating with new ones by providing pre-built packages for important lower-level controls protocols like Modbus and BACnet.

Actuation: support for exposing actuation of underlying devices in a consistent, safe way.

Programmer's interface: significant time went into making it convenient for programmers including a *driver package format*, *logging features*, *clean APIs*, *configuration and deployment management*, and the other niceties of a real software package.

sMAP's original design comprises two underlying aspects: the *metrology* deals with what abstract quantities should be represented and how they should be organized; the *architecture* has to do with how the metrology is implemented in a real system. At its core, sMAP is a method of making available streams of *discrete*, *scalar* data and control points. Although other data sources such as imagery and acoustic data are also common, we do not include them in the the design; they are addressed by existing work on multimodal sensor data repositories.

Although scalar measurements consist of a single number, their interpretation depends on knowing how to convert that number into engineering units as well as a host of other

information. The value is normally a digitized signal from an instrument, measuring a property of the physical world. By providing an indication about what that property is and what units were used, a client can tell the basic type of instrument present. A timestamp and sequence number place the value within a stream of discrete values.

4.2.1 sMAP Time Series

The organizing principle behind sMAP is to represent each instrument channel as a single `Timeseries` object. Time series consist of sequences of readings from a single channel of an instrument and associated metadata. They may be organized by the sMAP implementer into `Collections`, representing logical organizations of the instrumentation; this organization is reflected in the resource hierarchy exposed by the sMAP HTTP server. `Timeseries` objects are durably identified by `uuid`s, which place all streams into a single, global namespace.

sMAP supports adding metadata to timeseries and collections to better support integrating existing data sources where the metadata should be captured along with the data. sMAP also requires a minimum amount of metadata, required for the interpretation of the data. These include *engineering units*, *data type* and *time zone*. Using HTTP, sMAP exposes all time series as resources at standardized URLs with respect to a sMAP root and follow the paradigm of Representational State Transfer (REST) [33]. Haphazard use of this design pattern is much maligned, but a resource-oriented approach to web services design is characterized by a systematic use of its conventions. Because the abstractions and data model we developed map neatly onto resources, we hold that this is a good fit for physical information. By locating them consistently we make it easy for clients to automatically discover use sMAP resources.

The top-level resources in the sMAP profile are:

`/data` contains all of the time series from the source. These are typically organized by the driver implementor into a sensible hierarchy of collections and time series objects, representing the underlying instrumentation. Each time series or collection may be tagged with key-value metadata.

`/reporting` allows control of periodic reports for syndication, discussed in Section 4.2.3. Each reporting destination is configured either statically in a configuration file, or dynamically using a resource-oriented interface.

`/jobs` allows clients to submit multiple requests for actuation simultaneously, with more control over temporal semantics.

Underneath the `/data/` resource, individual sMAP time series are organized into collections, with the time series as leaf resources; and example time series object is shown in Figure 4.4, taken from the Cory Hall monitoring project. Each time series is an object which contains both data and metadata allowing for the data's interpretation. When retrieved using HTTP, each of these objects is represented using a JSON structure following

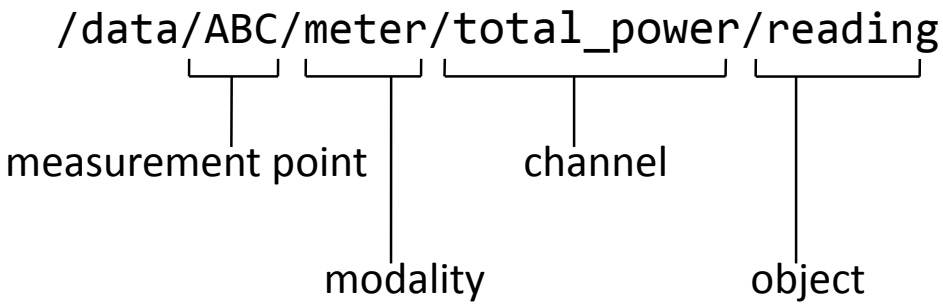


Figure 4.3: Canonical sMAP URL for the "total power" meter of a three-phase electric meter.

the sMAP specification; this also allows for combining multiple sMAP objects into a single larger object.

Description a string description of the channel.

Properties information about the channel required to store or display the data. This is made up of the data type, engineering units, and time zone.

Metadata additional information about the channel which is not necessary to archive or properly display readings. This is provided to facilitate the integration of existing sources. The full set of metadata for a time series also includes all the metadata for Collections in its recursive parent set.

Actuator if present, the channel includes an actuation component. The object describes what actuation is possible.

Readings a vector made up of the latest readings from the instrument.

uuid a globally unique identifier.

Each reading must include a timestamp, the reading, and an optional sequence number. The timestamp must be in units of Unix milliseconds.[1]

4.2.2 Metadata

Originally, sMAP supported very limited metadata. We extended the metadata model in order to better support integrating existing systems, as well as building systems where the

[1]Unix timestamps aren't actually unambiguous, due to complications arising from leap seconds. It seems like the alternative to using this time representation would be to use the ISO 8601:2004 time format as suggested by RFC3339. However, these string values are rather large when transfering a large number of readings so it's unclear whether the resulting object would be compact enough to satisfy our needs. If it is required, it additionally imposes higher burden on embedded devices who must maintain a calendar instead of simply a RTC since 1970.

```
{
  "Description": "Real power",
  "Metadata": {
    "Location": {
      "Building": "Cory Hall",
      "Campus": "UCB"
    },
    "SourceName": "Cory Hall Dent Meters",
    "Instrument": {
      "Manufacturer": "Dent Industries",
      "Model": "PowerScout 18",
      "SamplingPeriod": "20"
    },
    "Extra": {
      "Driver": "smap.drivers.dent.Dent18",
      "MeterName": "basement-1",
      "SystemType": "Electrical",
      "Service": "PNL1"
    },
  },
  "Properties": {
    "ReadingType": "double",
    "Timezone": "America/Los_Angeles",
    "UnitofMeasure": "kW"
  },
  "Readings": [
    [1397344709000, 72.3]
  ],
  "uuid": "dc7a9fde-9ea4-5bc4-b97c-fd7fc1de5d85"
}
```

Figure 4.4: An example time series object exported by sMAP. Sensors and actuators are mapped to time series resources identified by UUIDs. Meta-data from underlying systems are added as key-value tags associated with time series or collections of time series.

Metadata	**Use**
Value	The quantity
Units	Interpretation
Measured Quantity	Type of measurement: water, electric, *etc*
Global timestamp	Interpretation and alignment
Sequence number	Missing data detection; computing actual sampling period
Time zone	Interpretation in local time
Instrument range (min-max)	Establish dynamic range
Instrument identifier	Establish traceability

Table 4.3: Example metadata required to interpret a scalar data stream.

implementors have significant metadata. We make several simplifying assumptions.

Because metadata changes are rare relative to the data changes, we can both efficiently compress the metadata when transmitted over the network and also resolve many issues dealing with what happens when resources move: we do not need to track the move since instead all timeseries which were moved simply receive new metadata. Since the metadata

applies to time series which are identified by UUID, there is never an ambiguity about which piece of metadata applies to a point, and it is always safe to re-send all metadata.

Both `Timeseries` and `Collections` objects support the inclusion of metadata to provide additional information about measurements. Ideally this metadata is programmatically obtained from an existing system, or entered by the implementor; it is not intended that this metadata will change frequently. This metadata is structured as a set of hierarchical key-value pairs within a sMAP object. This allows qualifying pieces of metadata with their source; for instance, the `Location/` namespace defines a common set of location metadata such as city, building, postal code, *etc.* For convenience, sMAP defines types of metadata to facilitate information exchange: `Instrument` and `Location` metadata. Additionally, other information must be placed into the `Extra` fields, and may encapsulate any other metadata description in use.

This application of metadata is illustrated in Figure 4.4, where the time series object contains metadata point about the series being collected. The metadata keys, contained under the top-level `Metadata` key, contain information about the point. In this example, the metadata keys are used to relate the point back into one of the underlying views of Cory Hall. For instance, the `Metadata/Location/Building` tag allows us to include this feed in rollups of data from that building, while the `Metadata/Instrument/Model` tag would allow us to apply any instrument-specific corrections needed. The `Metadata/Extra/Service` tag begins to relate the data feed to the electrical distribution tree from Figure 2.7 – where in the electrical distribution tree the feed fits in.

4.2.3 Syndication

The purpose of reporting is to allow consumers to receive timely, reliable notifications of changes to the sMAP tree. These allow for a sMAP source to function both as an event source for control applications, providing notifications of changes, and also as a source for archival data. Because of these design goals, there are two aspects to the design of syndication: first, avoiding polling due to the need for timeliness, and second, the need for local buffering to provide reliability in the face of outages. To meet these needs, sMAP uses a webhook-style callback mechanism, combined with a local buffer for outgoing data. In this pattern, there are at a minimum two parties involved – the sMAP source and the data consumer. To start a syndication session, the data consumer registers a web hook URI along with other parameters with the sMAP source. Because the registration requires the allocation of local resources on the sMAP source for buffering data, this process gives the source the opportunity to reject the request if it is fully subscribed. Later, when new data are available, the sMAP source sends the data to the web hook URI using a POST request, where the body contains collections of time series objects.

This pattern has a few advantages compared to other approaches to syndication, specifically message bus approaches. First, because the data are buffered at the source, it respects the end-to-end principle and is tolerant of many different networking failures that can occur – link failures at the edge, name resolution errors, misbehaving routers, and man-in-the-middle

attacks (if using SSL). Furthermore, although message brokers often provide a "reliable transport" function, it still must include local buffering at the source since failures in the edge network are possible – indeed, in our residential deployments discussed in Section 4.1.1, the majority of failures were at the edge link.

Registration

Clients wishing to receive notifications of new data or metadata changes do so by creating a reporting instance. To get started, they post a reporting object to the `/reports` resource on the server; an example request body is shown in Figure 4.5.

```
{
  "MaxPeriod": 2147483647,
  "MinPeriod": null,
  "ExpireTime": null,
  "ReportDeliveryLocation": [
    "http://db1:8079/add/3gSL2xtZqbHPKr3ATybkxigwPpfTMp2JNQGm",
    "http://db2:8079/add/3gSL2xtZqbHPKr3ATybkxigwPpfTMp2JNQGm"
  ],
  "ReportResource": "/+",
  "Format": "gzip-avro",
  "uuid": "72c2fc09-cece-5e9b-9b98-1ebf60d375ff"
}
```

Figure 4.5: An example reporting endpoint installed in a sMAP source. The reporting object allows multiple destination for failover, as well has various options controlling how and when data are published; `gzip-avro` specifies that the outbound data are to be compressed with a gzip codec after first being compressed using Apache Avro.

A copy of one of these objects installed in a server is known as a **reporting instance**. When a reporting instance is installed, a sMAP server will periodically deliver updates to time series objects to one of the delivery location URIs specified in the instance, according to what resources the source has requested. The reporting instance on the server also allocates a buffer on the source where pending data are placed if the source is unable to communicate with the delivery location.

The most common use case is that a sMAP client subscribes to a resource like `/+` to receive all new data from the source. sMAP implementations may choose not to deliver the entire timeseries object but instead only include keys that have changed – typically only `Readings`. The fields each address a consideration discussed in Section 4.1:

`ReportResource` identifies the set of resource on the server which the client is interested in receiving, relative to the `/data` resource. In the example, "`/+`" refers to the recursive child set, therefore subscribing to all time series on the source. It is also possible to use wildcards (*e.g.*, `/*/voltage`), to subscribe to a subset of time series on the source.

`ReportDeliveryLocation` is a list of URIs specifying where report data should be delivered. The sMAP server will continue attempting to deliver data until it receives an HTTP success response from one of these servers. In the example, we keep attempting to send data until either `db1` or `db2` is successfully contacted; the long random string component is used by this endpoint to demultiplex incoming data.

`MinPeriod` specifies the minimum interval between reports. A sMAP server should not deliver reports more frequently than this. If the minimum period is so long as to prevent the sMAP server from buffering all the data accumulated in the period, it prefers deliver the latest data. If not included, the default is 0.

`MaxPeriod` specifies the maximum period between reports. After this much time elapses, the sMAP server should deliver a report regardless of whether there is new data to indicate liveness. If not included, the default is infinite; this can provide a keep alive message in the case sMAP is being used in an event-triggered mode rather than periodic reporting.

`ExpireTime` time in UTC milliseconds after which reports should be stopped, undelivered data dropped, and the report removed. Default is "never."

New reports are created by submitting the object as in Figure 4.5 to the `/reports` resource using an HTTP `POST` request. The sMAP server supports the range of create, update, modify, and delete operations on this resource using the appropriate HTTP verbs; furthermore, if authentication and authorization are used, the sMAP source may require the same authentication and authorization to modify or delete the report. sMAP servers implement a number of policies to deal with reporting instances where the delivery location is not accessible. They may keep trying for a fixed period of time before removing or deactivating the report instance, or buffer data while the recipient is down and retry with all of the accumulated data periodically.

Static Report Configuration

sMAP is mostly used in an "online" setting, where sources are automatically discovered and subscribed to by data consumers or other sMAP proxies. However, there are certain use cases where it is desirable for reporting to be configured manually on the sMAP server, rather than using the online system. For instance, if the sMAP server is behind a NAT, or if the data is actually not online but rather an import of an existing database. In these cases, a sMAP server may provide a provision for configuring report instances via a configuration file; for instance, our implementation of sMAP supports an INI file-like syntax specification of the server, including reporting instance like the one in Figure 4.6.

Differential Transmission

When a report instance is created, a sMAP source must send the entire resource which the report instance refers to, with all metadata. However, frequently only a small portion of the

```
; send data to the database
[report 0]
ReportDeliveryLocation = 'http://db1:8079/add/3gSL2xtZqbHPKr3ATybkxigwPpfTMp2JNQGm'
```

Figure 4.6: A configuration file snippet configuring a reporting destination.

resource changes; for instance, the metadata is often static and only the Readings part of a timeseries changes with each reading. For a particular report instance, a source may only send changes once it has successfully delivered the entire object. As an example, consider the object in Figure 4.4. When a new report instance is created, the sMAP source must send the entire object, including the full set of keys under `Metadata` since it cannot assume that the receiver has any information about the object. However, once that metadata has been sent, it can send a much smaller object containing only the changes; for instance, the one shown in Figure 4.7 where a new datum has been generated, but all other properties are unchanged.

```
{
  "Readings": [
    [1397344719000, 72.7]
  ],
  "uuid": "dc7a9fde-9ea4-5bc4-b97c-fd7fc1de5d85"
}
```

Figure 4.7: A differential version of the object in Figure 4.4 where a new datum has been generated. Only the new reading needs to be included, along with the UUID for identification of the series.

4.2.4 Actuation

Adding actuation support to sMAP is important, because closing the monitor-model-mitigate loop ultimately requires changing the way systems are operated; this generally means making real changes to the operation of equipment. Actuators are well-modeled as time series in many ways, since they generally have a state which it is meaningfully tracked over time. In order to represent an actuator as a sMAP object, we allow for inclusion of an *actuator model,* which defines which states the actuator can take on. In addition to recording the state over time and allowing clients to discover the type of actuation present, there are two other primitives useful for building higher-level controllers on top of sMAP sources. The first is *coordinated control* of several actuators at the same time, with well-defined semantics around when the underlying actions occur to help applications reason about when actions which require making multiple changes at once will occur. The second is a concept of the *write lease*, where a value is changed for a period of time, after which the write is undone in some way.

sMAP provides several default types of actuators: binary, N-state, and continuous. Binary actuators have two positions corresponding to logically "on" and "off"; N-state actuators have a number of discrete positions, while continuous actuators can be set to any position within a fixed interval. Aside from the minor detail that the state can be written in addition to read, actuators have much in common with other channels; such as units and a current value. Because they are extended time series, commands sent to the actuator are naturally logged through the same reporting framework used for all other time series data.

```
{
  "Actuator": {
    "Model": "binary",
    "States": [
      [ "0", "off"],
      [ "1", "on"]
    ]
  },
  "Properties": {
    "ReadingType": "long",
    "Timezone": "America/Los_Angeles",
    "UnitofMeasure": "Relay Position"
  },
  "Readings": [
    [ 1397412776000, 0]
  ],
  "uuid": "95fd532e-0356-5dc5-8fa5-07a9aab1b396"
}
```

Figure 4.8: An example `Timeseries` object representing a binary discrete actuator.

When actuation is present, the `Timeseries` object corresponding to the actuator must include a `Actuator` key. Figure 4.8 shows an example object representing a simple binary relay. For this actuator type, the `States` present contain aliases for different states the actuator can take on – in this case, we can refer to state "0" as "off." For the case of discrete actuators, being able to create aliases for states is useful when dealing with, for instance, the case when a switch position is inverted and switch state "0" (open) corresponds to the equipment being on.

When a `Timeseries` is used to represent an actuator, the `Readings` field is used to communicate the current actuator position. The sMAP implementor may choose to only generate a new reading when a control input is received, periodically, or when the state of the actuator changes.

Clients wishing to affect actuator position do so using the `POST` verb on the `Timeseries` representing the actuator. For instance, suppose the actuator time series in the example was available on the sMAP source at `/data/relay/1`; a client could send a simple state change request directly to that URL.

sMAP servers providing actuation may also provide the `/jobs` resource. This resource allows clients to control multiple actuators at the same time, by providing a `Job` object. When using the jobs interface, clients are able to submit multiple actions at the same time,

and gain additional control over what happens in the face of multiple writers. Figure 4.9 shows an example of a simple job, in which we command two actuators on the sMAP source for 900 seconds – 15 minutes. The effect of this job is somewhat different than what would be accomplished by writing to the two actuators separately; first, the writer receives a lock on the output value for the duration of the jobs. This is important since it allows the writer to receive exclusive access to those time series for some duration. Second, the actions occur concurrently, and are subject to the error policy specified by the `Error` key. This may be either `abort`, in which case an error performing any of the actions will cause the source to stop processing, undo any completed writes, and return an error, or `ignore`, which will ignore errors and continue processing.

```
{
  "jobid": "340f10fa-c340-11e3-8bd4-b8e856313136",
  "StartTime": 1397416647000,
  "Duration": 900,
  "Errors": "abort",
  "Actions": [
    {
      "uuid": "53d0ac14-c340-11e3-bbea-b8e856313136",
      "State": 1
    }, {
      "uuid": "615e80c2-c340-11e3-9e15-b8e856313136",
      "State": 1
    }
  ]
}
```

Figure 4.9: An example `Job` object, writing to two actuators at the same time. Because a duration is provided, the underlying actuators will be locked until the job completes or is canceled; additional writes will fail. The `uuid`s reference the underlying time series objects.

4.3 Implementation

A key contribution of sMAP is not just a description of a few simple objects which can be used to describe time series, but a set of software abstractions which can be used to quickly interface with new data sources, and promote code reuse. Figure 4.10 shows the general structure of the sMAP runtime, which implements the features discussed in Section 4.2. The system, implemented in Python, is structured so that most device-specific code (code interfacing with an underlying system providing the data) is structured as a driver class. The runtime provides hooks for starting, stopping, and configuring these drivers, and provides services for publishing their data to any number of consumers; generally it is a container for running small pieces of device-specific code while managing the rest of the application lifetime.

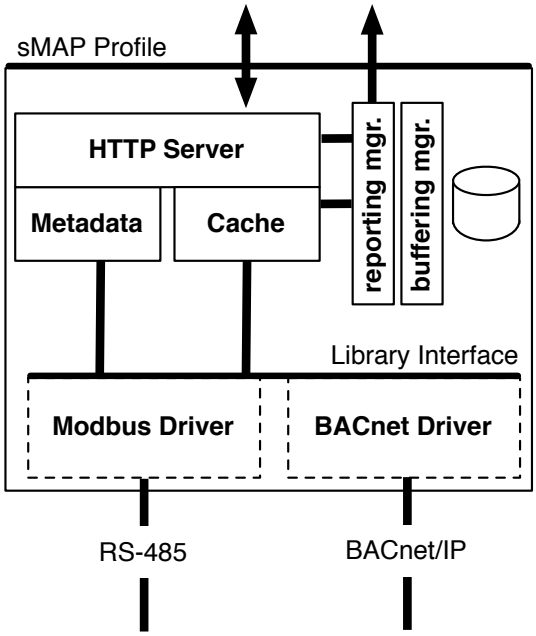


Figure 4.10: Our implementation of sMAP provides a clean runtime interface against which to write reusable "driver" components, which contain logic specific to the underlying system. The runtime is responsible for formatting sMAP objects, implementing the resource-oriented interface over HTTP, and managing on-disk buffering of outgoing data. It also provides configuration-management and logging facilities to users.

4.3.1 Drivers

Drivers provide encapsulation of device-specific code into a package that encourages reuse of code for interfacing with specific devices. The example driver shown in Figure 4.11 illustrates the core programmer's abstraction when using sMAP: the `SmapDriver`. Even this simple example makes use of many of the runtime's services. The `setup` method receives a parsed representation of the driver configuration, which is managed by the supporting runtime and allows operators to pass parameters such as network address and credentials to the driver. The method creates one time series with the path `/sensor0`; this time series implicitly receives a UUID. The driver adds a small amount of metadata; this is typical for drivers since they often can determine a make and model number either as a static value or by probing the underlying instrument. Finally, the `start` method results in period polling of the device for new data; in this case, calling an external library `examplelib.example_read` to retrieve data from the underlying device.

```
from smap.driver import SmapDriver
from smap.util import periodicSequentialCall
from smap.contrib import dtutil

# read a value from an example sensor
from examplelib import example_read

class ExampleDriver(SmapDriver):
    """Driver which creates a monotonic time series that increments every second."""
    def setup(self, opts):
        """Create time series and read configuration parameters"""
        self.add_timeseries('/temperature', 'V')
        self.set_metadata('/temperature', {
            'Instrument/ModelName' : 'ExampleInstrument'
            })
        self.rate = float(opts.get('Rate', 1))

    def start(self):
        # Call read every 'self.rate' seconds
        periodicSequentialCall(self.read).start(self.rate)

    def read(self):
        val = example_read()
        self.add('/sensor0', val)
```

Figure 4.11: An example sMAP driver. The `setup` method receives configuration parameters from the configuration file; `start` is called once all runtime services are available and the driver should begin publishing data. In this example, we call an external library, `examplelib.example_read` to retrieve data from a temperature sensor and publish the data every few seconds. The `opts` argument to setup, provided by sMAP, contains configuration information from the runtime container.

4.3.2 Configuration and namespaces

```
[/]
uuid = 90480f4f-d938-11e3-9f56-b8e856313136

[/example]
type= example.ExampleDriver
Metadata/Location/Building = "Soda Hall"
Metadata/Locaion/Campus = "UC Berkeley"
```

Figure 4.12: A configuration file setting up the example driver. Importantly, this contains a base UUID defining the namespace for all time series run within this container, and one or more sections loading drivers. In this case, only the `ExampleDriver` is loaded.

The sMAP runtime provides a number of conveniences to the implementor in addition to buffering data. Figure 4.12 shows an example of an INI-like configuration file read by the runtime and used to instantiate one or more drivers within a container. The runtime

reads this file at startup, and loads the drivers listed into the container's process, passing in any configuration options from the file; this is commonly used for parameters like network addresses of the underlying device, login credentials, and any other information needed to connect and retrieve data.

When creating new time series using the `add_timeseries` method, the runtime qualifies the name of any time series added by the driver to the name in the configuration file – for instance, this source would have a single time series named `/example/temperature`. This namespace qualification allows the same driver to be loaded multiple times, for instance when there are multiple copies of the same instrument type. The configuration also contains a *uuid namespace*, present in the `[/]` section; this is used to avoid the need to have large numbers of uuids present in either the configuration or code. When time series are added, as in line 12 of the example driver, its path on the local server is combined with the namespace uuid to produce the identifier of the relevant time series[2]. This method allows most users to only generate a single new unique uuid, place it inside of the driver config file, and have the runtime consistently generate a unique set of identifiers for all of their time series.

4.3.3 Utilities

The library also contains a robust set of utilities for taking a simple piece of driver code like the example and placing it into production. `smap-tool` provides a command-line interface to sMAP sources, making it easy to inspect a running source, install new report destinations and retrieve the latest data. `smap-reporting` and `smap-monitize` are utilities for demonizing a sMAP source on a server, configuring logging, process monitoring, and liveness checking to make sure the source reliably survives reboots and other system events. `smap-load` allows for running drivers in offline mode, for instance when performing an import of existing data, which simply requires delivering it to a set of reporting destinations. Finally `smap-load-csv` has a number of tools for converting data stored in comma-separated value files into a set of sMAP time series and publishing the result.

4.4 Evaluation

The three major considerations of this work are *metrology*, *syndication*, and *scalability*. In order to evaluate the success of our design, we use these to provide a set of questions we can use to evaluate sMAP.

Metrology → Completeness and Generality. To show that our design for metrology is simple yet effective, we show that a large variety of common data sources from electric, environmental, process control, and meteorological systems can be presented within the system. We also examine a set of deployed systems from the literature.

[2]In this example, we can generate the UUID of the temperature stream using python, *e.g.*, `uuid.uuid5(uuid.UUID("90480f4f-d938-11e3-9f56-b8e856313136"), "/example/temperature") => 'be57f975-d039-5c67-8cc5-9a6df575f993'`

Syndication → Application Use. sMAP is being used as the data plane of a host of higher-level applications. By having a common interchange format, these applications are "portable" from one collection of sensors to another.

Scalability → Practical Implementations. We show that sMAP can be practically implemented on devices ranging from powerful web farms to minuscule embedded devices without losing the essential benefits of the approach.

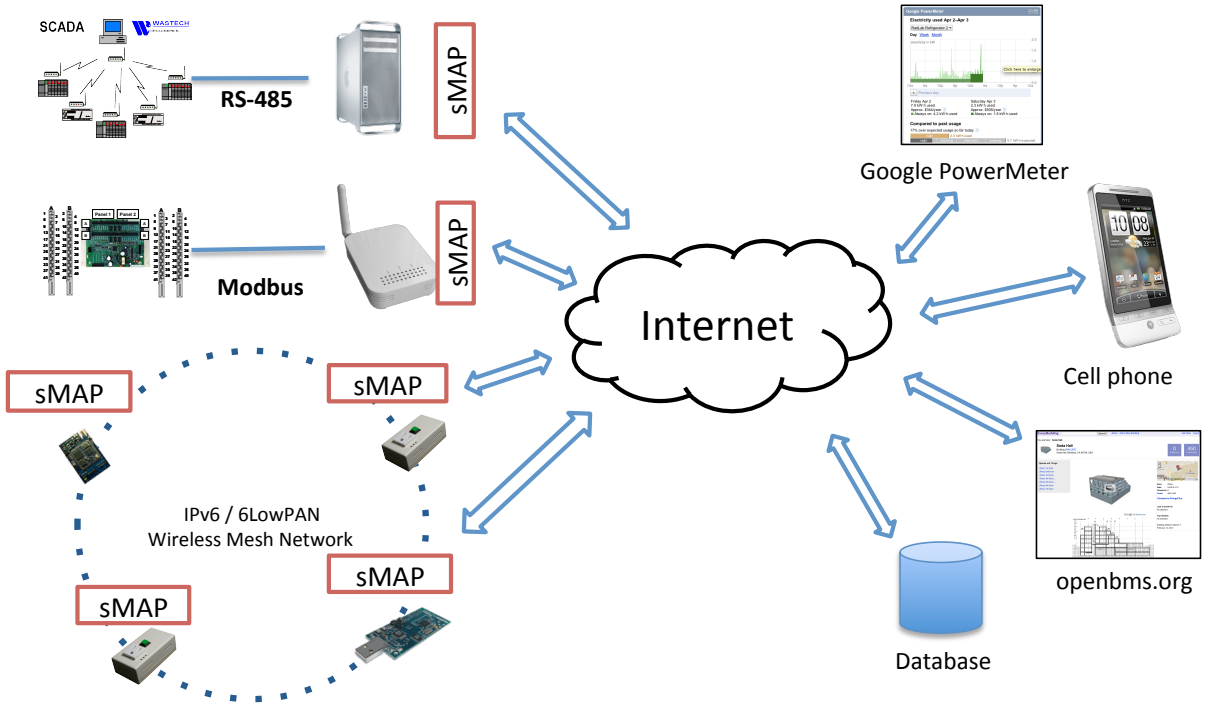


Figure 4.13: Part of the sMAP ecosystem. Many different sources of data send data through the Internet to a variety of recipients, including dashboards, repositories, and controllers.

4.4.1 Complete and General

The first question we examine is how general sMAP is: what is the diversity of data it can represent. sMAP was developed during a process to create a *building-to-grid* testbed, for experimenting with novel interfaces and feedback mechanisms between a commercial building and the electric grid. The building in question is Cory Hall, the electrical engineering building at Berkeley. First commissioned in 1955, Cory Hall consumes approximately 1MW of electricity in addition to steam and chilled water used for heating and cooling. Our

Name	Sensor Type	Access Method	Channels
ISO Data	CAISO, NYISO, PJM, MISO, ERCOT, ISO-NE	Web scrape	13849
ACme devices	Plug-load electric meter	Wireless `6lowpan` mesh	344
EECS submetering project	Dent Instruments PowerScout 18 electric meters	Modbus	4644
EECS steam and condensate	Cadillac condensate; Central Station steam meter	Modbus/TCP	13
UC Berkeley submetering feeds	ION 6200, Obvius Aquisuite; PSL pQube, Veris Industries E30	Modbus/Ethernet, HTTP	4269
Sutardja Dai, Brower Hall BMS	Siemens Apogee BMS, Legrand WattStopper, Johnson Control BMS	BACnet/IP	4064
UC Davis submetering feeds	Misc., Schneider Electric ION	OPC-DA	34
Weather feeds	Vaisala WXT520 rooftop weather station; Wunderground	SDI-12, LabJack/Modbus, web scrape	33
CBE Performance Management Package	Dust motes; New York Times BMS	CSV import; serial	874

Table 4.4: Hardware Presentation Layer adaptors currently feeding time series data into BOSS. Adaptors convert everything from simple Modbus device to complex controls protocols like OPC-DA and BACnet/IP to a uniform plane of presentation, naming, discovery, and publishing.

testbed construction commenced with the instrumentation of several hundred sense points with thousands of channels, capturing much of the energy spend in addition to environmental characteristics. Some of the feeds of sMAP data in Table 4.4 were developed as part of this project. Since that project, many others have provided drivers to the sMAP project which provide interfaces using BACnet (during a study on Sutardja Dai Hall), OPC (during a collaboration with UC Davis), several different types of wireless sensors (during a collaboration with the Center for the Built Environment and Lawrence Berkeley National Laboratory), as well as improved tools for accessing data stored in existing MySQL databases and CSV files. In this section, we review how sMAP applied to a few of these projects.

Building AC Power

Commercial buildings typically distribute power in a tree from a few feeds into the building substation, which are split into multiple three-phase branches and finally broken down to single-phase circuits at panels. We have instrumented each of these levels in our Electrical

Engineering building, and made the data available via sMAP. Figure 4.14 outlines the hierarchical nature of electricity distribution and locates the various meters used to instrument the building.

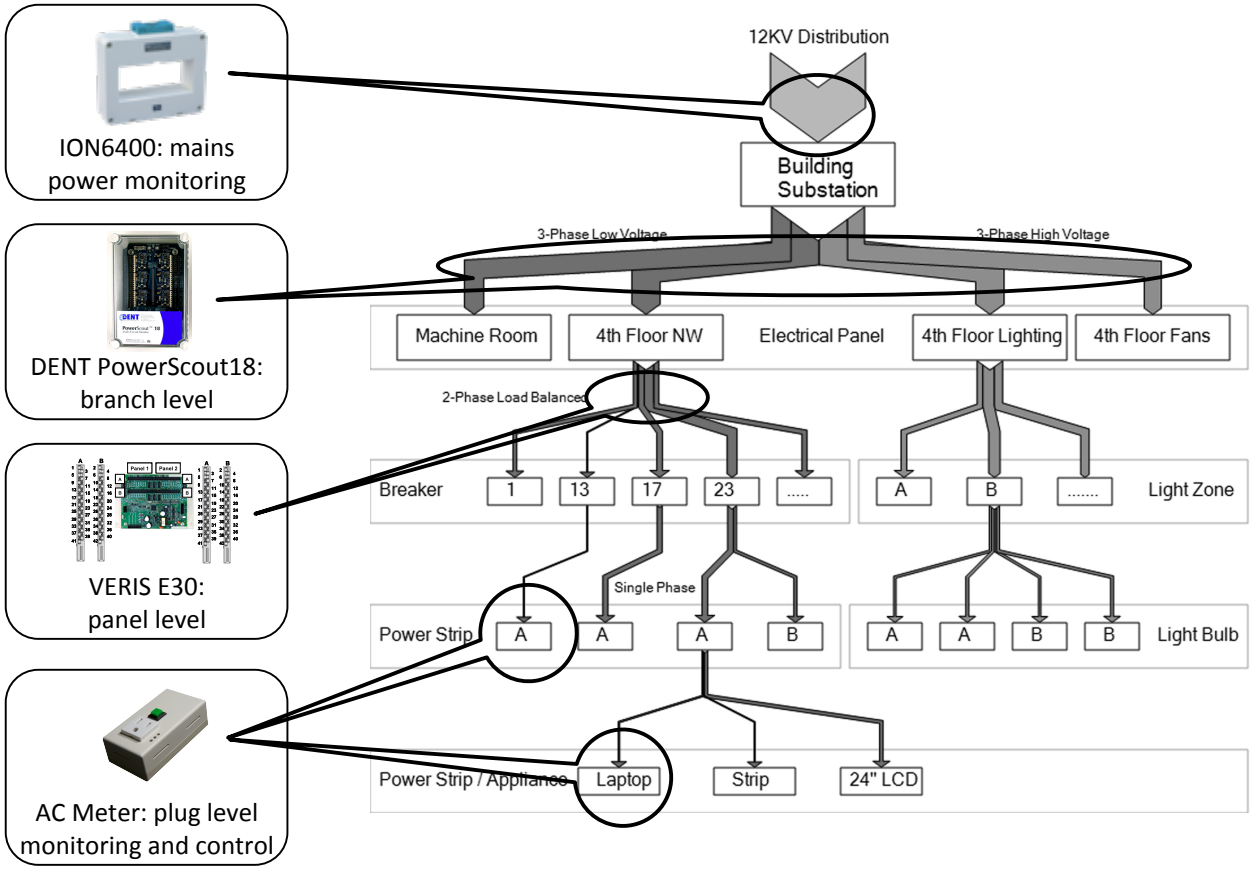


Figure 4.14: A Sankey diagram of the electrical distribution within a typical building, with monitoring solutions for each level broken out. All of these sources are presented as sMAP feeds.

At the top two levels, the substation and branch level, electricity is distributed in three phases through several transformers where it is stepped down from the the 12.5kV feed into the building. The instrumentation at these points consists of a large number of Current Transformers (CTs) and Rogowski Coils, which are present on the feed into the building and on individual phases of each branch. The building in question has two primary feeds from the grid, which are then split into 12 branches; this translates into 42 separate phase measurements (three per branch).

To distribute this data using sMAP, each branch or feed presents a sMAP interface to the world. Since each of these branches contains three phases, each sMAP instance presents several measurement points corresponding to each phase. In addition to these single-

phase measurements, there are several "virtual" measurement points which corresponds to measurements from different combinations of CTs like phase-to-phase measurements and total system data. Using the flexibility of sMAP to name measurement points by any valid URL resource, each branch or feed exports a total of seven measurement points: A, B, C, ABC, AB, BC, and AC. Furthermore, the driver for these instruments also applies a consistent set of tags to identify properties of the measurement – for instance, adding a `Metadata/Extra/Phase` tag indicating which leg of the three-phase circuit the measurement is from, and a `Metadata/Extra/ServiveDescription` tag relating the measurement to a point in the electrical tree.

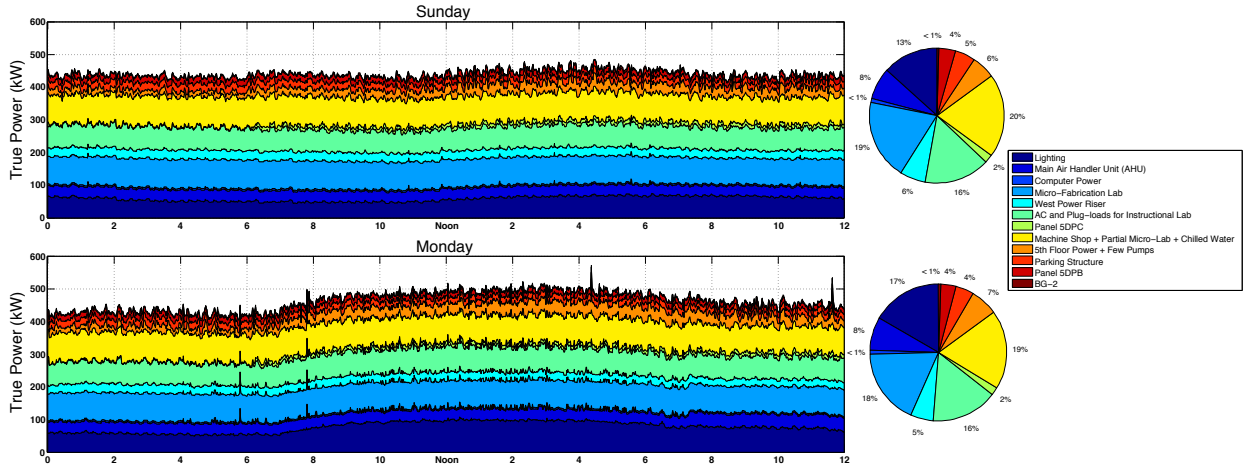


Figure 4.15: A detailed breakdown of electrical usage inside the Electrical Engineering building over two days. Data is pushed to a client database by a sMAP gateway connected to three Dent circuit meters, each with six channels. Power is used primarily by lighting, HVAC, and a micro-fabrication lab, as expected. Interestingly, the total power consumed on Sunday is $440kW$ while on Monday is $462kW$, an increase of less than 5% between a weekend and a weekday, indicative of an inefficient building. The difference between day and night is small as well. The only load with an obvious spike in power is lighting at around 7am on Monday, whereas most loads stay the same throughout the day and night.

Each of these points also contains multiple sensors and meters: for branch-level meters, there are seven sensors and three meters. These correspond to measurements like real, reactive, and apparent power, current, phase-to-phase voltages, power factor, and several other quantities.

Once electricity has been distributed through a branch, it is further stepped down to 120V for circuit-level distribution. These circuits split off from wiring panels located on each floor. To instrument this level of the distribution hierarchy, we used a device with 40 single-phase meters which is designed to be installed inside a breaker box. To map this arrangement onto sMAP, each circuit is treated as a single measurement point with several

Modality	Channel
meter	true energy
meter	reactive energy
meter	apparent energy
sensor	true power
sensor	reactive power
sensor	apparent power
sensor	current
sensor	displacement power factor
sensor	apparent power factor
sensor	line frequency

Table 4.5: Channels for each phase and total system measurement on a three-phase electric meter.

channels. Since this meter is much simpler, it only provides per-circuit energy consumption information (kWh).

Figure 4.16: An example Modbus-ethernet bridge. Modbus is run over a twisted pair serial cable (RS-485); converting it to Ethernet means removing any need to run new cable.

The meters in use are typical of modern electrical monitoring: they provide a Modbus interface running over RS485. In order to make them available over the Internet using sMAP, we use a Modbus – Ethernet adaptor (shown in Figure 4.16) to bridge to an IP subnet, and then run a gateway service on a server which periodically polls the devices and caches their last reading for use in sMAP. Since each manufacturer typically has their own map of Modbus registers, the gateway must be customized for each new brand of meter; of course, this effort is all transparent to clients who receive normally-formatted sMAP data.

Plug-load Power

The final level of electrical distribution is plug-level, where individual appliances are connected to the circuit. To monitor at this resolution, we used a custom device called the ACme. ACme's [53, 54] are single-phase plug-load meters that measure power and energy of typical AC appliances in households and offices. In addition, they are capable of controlling connected devices using an internal relay. ACme is a typical mote-class device based on a msp430 micro-controller and a cc2420 802.15.4 radio, shown in Figure 4.17. ACme is representative of a large class of sensors and meters found in commercial environments that measure physical phenomenons at a single point in some process. Examples of devices in this class include flow meters, temperature sensors, light sensors, and motion sensors.

ACmes use `blip`, our open-source IPv6 stack to form an ad-hoc network [105]. Since blip supports both TCP and UDP, there are multiple ways a protocol like sMAP can be scaled down to this device. We compare the options for this in Section 4.4.2.

From a metrological point of view, a single ACme is a device with a single measurement *point* – the plug – and multiple *modalities* – it senses power, meters energy, and actuates a relay. The actuation present on an ACme is particularly simple: a relay can switch the attached device. Since this fits into the library of sMAP actuators as a binary actuator, nothing new needed to be developed to enable this form of control.

External Data: CA ISO and WUnderground

sMAP can integrate existing, external data sources to help define the context in which our study building operates. Two forms of data relevant to our work were data from the California Independent System Operator (Cal ISO) concerning the total state-wide electricity demand, and weather data from various locations.

Weather data typically has multiple sensors and meters, instrumenting precipitation, wind, temperature, and pressure. These are easily translated to the appropriate abstractions as sMAP modalities; furthermore sMAP preserves some important metadata about the weather meter such as its station identifier, and make and model of the physical hardware. Numerous mote-class sensors would typically be incorporated to monitor interior environments. Because sMAP is relatively prescriptive – the hierarchy and object definition does not leave much freedom – the amount of non-shared code is very small; around 100 lines of Python each case. In fact, the amount of non-boilerplate code is even smaller; integrating the weather feed required only 43 new lines of code.

Other Deployments

To cast a wider net, we examined several years of SenSys and IPSN proceedings for papers describing the deployment of whole systems, or the development of new sensors; this listing is present in Table 4.6.

We found that although the *system design* typically addressed novel issues, the actual data retrieved tended to be very simple: typically, slow time series of a few parameters.

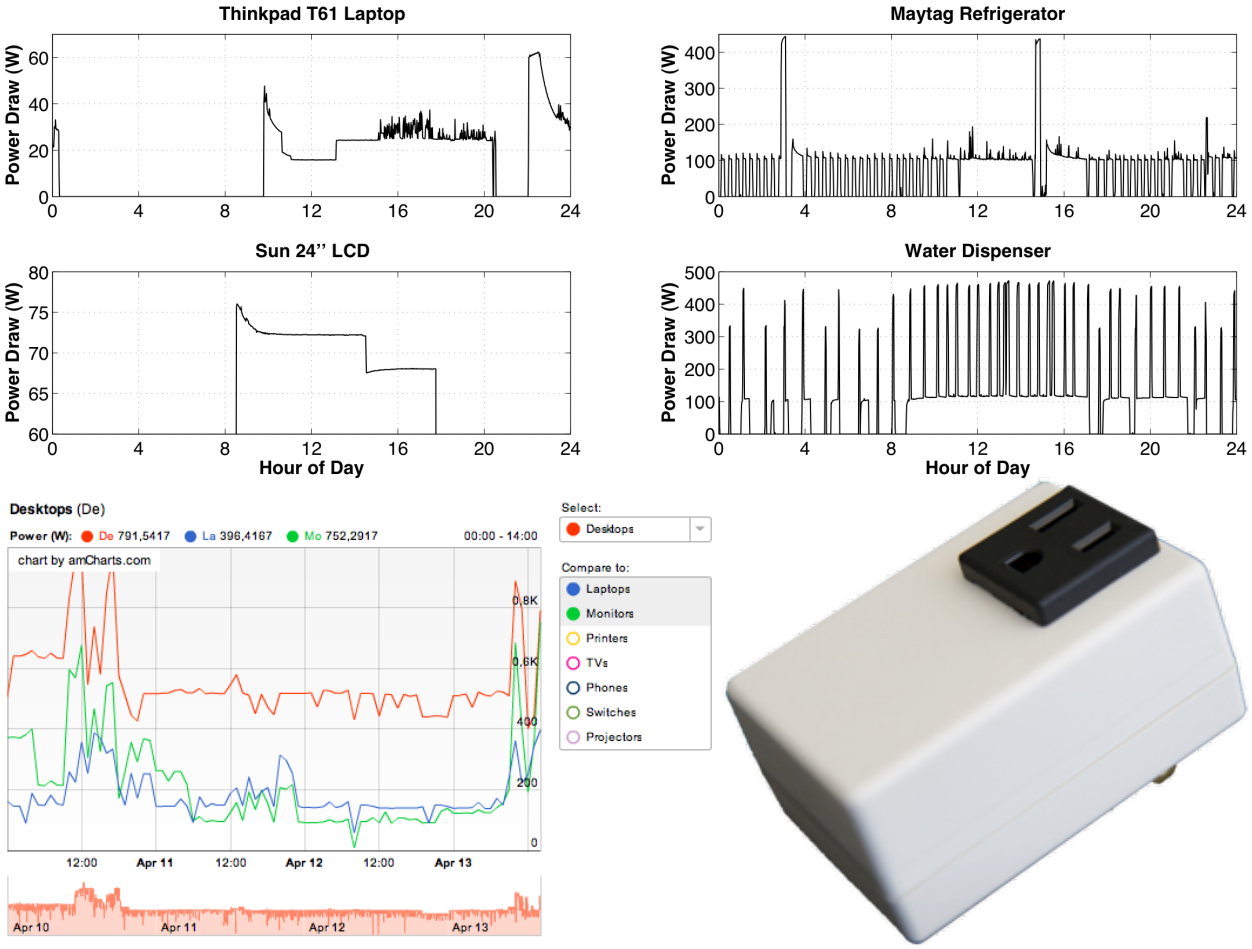


Figure 4.17: ACme wireless plug-load meters (bottom right) are used to monitor power consumption of various appliances, including a laptop and a LCD display in the top left, a refrigerator and a water dispenser in the top right, and aggregate consumptions in bottom left.

Since mote-class devices were the most common, the measurement point/channel abstraction appears to work well (as for the ACme). In fact, none of the deployments appeared to have a more complicated data model than a three-phase electric meter.

An important exception to this paradigm was best demonstrated by Lance (volcano monitoring) and the Heritage Building project: sMAP does not address high-frequency data like seismometer or accelerometer data. Although these are in principle time series of sensor data, it appears to us that sMAP is a sub-optimal design point for representing data from these applications; this is not a surprise since it was not an original design goal. We may need to address this with future protocol modifications. Another observation is that none of these deployments involved a significant actuation component. While there have been

Deployment	Modality	Sensed Quantities
RACNet [67]	Datacenter	temperature
ACme [53]	Electricity	true power, apparent power
Lance [112]	Geophysical/Volcano	seismometer
MEDiSN [61]	Healthcare	ECG, blood oxygenation level, pulse rate, *etc*
GreenOrbs [75]	Environmental/Trees	illuminance
Flood Warning [10]	Water	river level
NAWMS [60]	Water	flow rate
PermaDAQ [13]	Alpine environmental	temperature, conductivity, crack motion, ice stress, water pressure
Heritage Buildings [17]	Structural health	temperature, deformation, acceleration
HydroWatch [103]	Water cycle	temperature, light, humidity

Table 4.6: Deployments from SenSys and IPSN in the past several years.

deployments involving actuation reported in the literature, this has not been a dominant research focus. It is, of course, common in building environmental conditioning system.

4.4.2 Scalable

sMAP must be scalable in both directions: "up" to millions of connected clients or subscribers, and "down" to tiny embedded devices.

Scaling Up

Scaling sMAP up is, if not simple, a well-understood problem given that it runs over HTTP. Using the HTTP caching model, sensors can and do define how long their readings will be valid for using the "Expires" header – presumably, until the next sample is taken. Intermediate caching proxies then offload queries onto multiple servers. The syndication design we have built is also scalable, since in the futur, a single sMAP source may be republished other places on the Internet. For instance, a sMAP instance running on an embedded device may communicate only with a caching proxy that also offloads most syndication duties.

Another element of making sMAP scale to large data volumes is ensuring that publishing larger volumes of data is efficient. This is of special concern since since we are encoding much of our state using JSON which is somewhat verbose. Fortunately, multiple different content encodings can be used with HTTP; Table 4.7 shows some of the tradeoffs. For this example, we use the data published by the Cory Hall Sub-metering project – its 130 Dent meters together export 3870 individual time series. The raw size of the data, including only timestamp, value, and UUID is around 108kB. When packed with JSON, it is 4.5x larger. However, sMAP objects definitions are all made using an Apache Avro [6] schema. Using the Avro toolkit, much of the repetitive nature of JSON can be elided – integers and floats are packed as binary data, and common strings are moved to a strings table. By applying

Compression Scheme	Resulting Size	Percentage of Raw
Raw	108kB	100%
JSON	487kB	450%
JSON + gzip	134kB	124%
Avro	296kB	273%
Avro + gzip	117kB	109%
Avro + bzip2	96kB	88.5%
Raw + gzip	83kB	77%
Raw + bzip2	79kB	73%

Table 4.7: Comparison of data size from a meter producing 3870 time series. "Raw" consists of packed data containing a 16-octet UUID, 4-octet time stamp, and 8-octet value but no other framing or metadata. Avro + gzip and Avro + bzip2 are both available methods of publishing data from sMAP sources.

this to a large sMAP object, and applying a standard compression algorithm like `gzip` or `bzip2`, we can recover all of the overhead added by using JSON, and in fact be competitive with the compressed raw data as well.

Scaling Down

A more challenging test is whether sMAP can scale down to fit on embedded devices. While it is possible to expose embedded devices' functionality via web services by using an application-layer gateway, this is a limiting architecture because it requires the gateway to be aware of the functionality of each connected device. In the sMAP architecture, a gateway or proxy may still be present for caching, but it is a transparent component between the client and the server.

As a proof of concept, we implemented a prototype embedded version of sMAP, using an architecture shown in Figure 4.18. For our prototype implementation of sMAP on con-

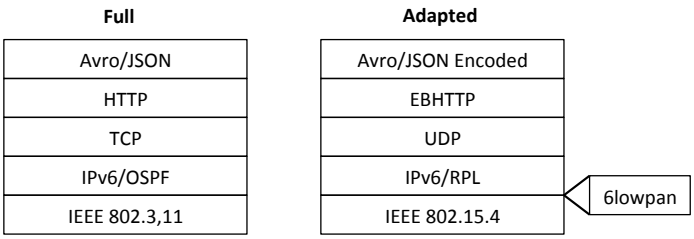


Figure 4.18: Layers in the protocol stack, and the protocols in use in the full version of sMAP, next to the version adapted for embedded devices.

strained devices, we used draft versions of the `6lowpan` standards, and a predecessor to RPL known as HYDRO [77, 105]. These are used in the `blip` package which provides UDP and TCP interfaces on devices running TinyOS [66]. Within constrained networks, we can

also run sMAP over Embedded Binary HTTP (EBHTTP). EBHTTP is a binary-formatted, space-efficient, stateless encoding of the standard HTTP protocol, intended for transport of small named data items (such as sensor readings) between resource-constrained nodes [107]. By using EBHTTP, we reduce the number of message bytes needed to transport sMAP data, while maintaining the URL structure and HTTP method semantics of sMAP. EBHTTP is very similar to later efforts at an HTTP targeted at embedded devices in the IETF, notably the Constrained Application Protocol (CoAP) [14]. We have implemented all the components of the stack presented in Table 4.18; it is used to make sMAP data from AC plug load meters available.

Component	Size
Application Code	936
HTTP	542
TCP	3534
Routing (HYDRO)	2890
IP + ICMP	4382
6lowpan	2262
Link	4926
Total	19472

Table 4.8: Code size of key elements of a sMAP implementation using Avro/JSON, HTTP, and TCP.

In addition to the message size reduction, we reduce the size of the embedded code needed to implement this service by replacing the TCP stack with a UDP stack, replacing the text-based HTTP parser with a small EBHTTP parser, and replacing the text-based JSON encoder with a simpler packed JSON encoder. This compression is not without cost. By switching to UDP, we lose the segmentation and flow control provided by TCP, and switch to a simpler stop-and-wait reliability model. But, this is often an acceptable tradeoff when using sMAP to communicate with constrained devices.

4.5 Takeaways

We cast a wide net with sMAP, attempting to sweep a great variety of existing instrumentation into a common framework. We directly incorporated many of the needs identified in the design motivation (Section 4.1) into the design of the system. This was the result of an iterative design process that unfolded across several years and the deployment experience gained at the Berkeley and LBNL campuses. For instance, the work with ACme nodes in homes and commercial buildings, which were generally highly reliable, led us to conclud that no matter how reliable wireless protocols become, there will always be a need for local buffering and retransmission when possible, since not all failures are network failures. This directly informed the design of our **reporting** subsystem. Our work with the relatively high data rates observed with building management systems led us to ensure that the underlying protocol was **efficient** both for a trickle and a flood of data, both of which benefit from the

types of compression needed to scale down to embedded devices. The sheer number of device types represented in Table 4.4 – more than 30 individual data sources – led us to appreciate the good tooling and robust runtime support provided by the library.

The various data sources noted here provide the basis of the next several chapters – first, addressing the issues surrounding capturing and querying this volume of data, and then completing the loop and finally actuating the underlying hardware.

Chapter 5

Time Series Data Storage

Most modern buildings contain thousands or tens of thousands of sensors and actuators; these large sensor and actuator networks are widespread. Collecting and making use of data from these devices is currently hard. Unlike other types of time series data such as financial, logistics, or clickstream data, making sense of building data requires relatively detailed understanding of the context the data was generated in. Buildings are also large cyber-physical systems, since the result of computing on the data is often used as the input to actuators, forming control loops. It is also primarily generated by machines rather than people, yielding the potential for significantly higher data volumes than other primarily human-driven systems. The raw data consists of time series, not high-level events, suggesting that an initial level of processing is needed before it is ready for further analysis. This suggests using a special purpose storage engine to maintain the historical record. A typical control system such as those described in Chapter 2 has the capacity to produce a significant amount of data; thousands of points sampled as much as once a second, if the system has a fast enough instrumentation bus. A challenge is that it is that this data stream is essentially unending – as long as the control system is operating the data will be produced and must be either stored for later analysis, processed in near-real-time, or discarded. The first step of our monitor – model – mitigate loops is to simply collect all of that data centrally. In this chapter, we discuss some of the properties of time series data which can be leveraged for an extremely efficient storage engine.

5.1 Challenges and Opportunities

Because the underlying data are predominately time series of scalar valued readings, the access patterns are not at all random; read access is nearly always structured around the time dimension of the data. When data are inserted, they are typically either part of a bulk load of data from an existing data base, or appended as a single (or small number) of readings which were just produced by a sensing system. In aggregate, this "trickle load" pattern is common, where data from a large number of sensors trickles in, with the overall

```
CREATE TABLE time_series {
    stream_id INT;
    timestamp DATETIME;
    value DOUBLE PRECISION;
};
CREATE INDEX stream_time_index
    ON time_series (stream_id, timestamp);
```

Figure 5.1: A typical representation of time series data in a relational schema. The

rate becoming large only in aggregate. Data retrieval almost always also is ordered around time: retrieving the latest data from a time series for real-time uses, or else retrieving all data from a few streams, within a time range for block analysis.

These access patterns challenge many relational databases, while providing many opportunities for time series-specific optimization. Using a conventional relational schema with a product like MySQL or PostgreSQL leads to significant inefficiencies; consider the schema in Figure 5.1. This schema represents the essential pattern for time series – the first field, `stream_id` identifies what time series the data is part of, and the other two fields contain actual data values. A typical implementation would represent each of these fields using 8 octets, and so the bare record size is 24 octets.

In a monitoring system, this table will grow very large; tens of billions of rows are to be expected even from modest installations. This leads to poor performance in conventional RDBMS's for several reasons. Firstly, per-row overheads are often large relative to the record size, leading to significantly more IO than is truly required; table compression can help alleviate this problem. A more significant issues is the index necessary to avoid sequentially scanning billions of rows; since essentially every datum is being inserted twice, once as a raw data row, and secondly into a B+-tree leaf, appending data becomes incrementally slower as the database size increases, due to rebalancing the extremely deep B+-trees.

The ordering of access along the time dimension also suggests processing optimizations which can significantly improve performance through better batching. In a classical database system, data tuples are read off disk or from cache and processed by either pushing or pulling each record through an operator graph. This model introduces a certain amount of per-tuple dispatching overhead per tuple; by batching operations along the time access, we can achieve significant performance gains by processing time chunks together.

5.2 A Time Series Storage Engine

To take advantage of the unique properties of time series data, we have designed and implemented the `readingdb` storage engine[1] – a key-value store for time series data, efficiently representing the map:

$$(\texttt{stream\_id}, \texttt{timestamp}) \Rightarrow \texttt{value}$$

`readingdb` identifies each datum by 32-bit integer stream ID and 32-bit timestamp with second-level resolution; all other semantics are left to other layers. Internally, individual values are placed into dynamically-sized buckets containing up to a day's worth of data from a single stream, reducing the size and frequency of index updates while preserving the ability to quickly seek to an arbitrary timestamp in a stream. Each bucket is compressed by first packing deltas between neighboring timestamps and readings into a Google protocol buffer [38], and then applying a Huffman-tree code to the result; this process is illustrated in Figure 5.2. Since many feeds report only slowly-changing or discrete values – for instance, light switch position – this algorithm gives excellent compression with little or no penalty for highly entropic data. It is built on top of Berkeley DB [82], an embedded key-value store, which provides on-disk page management, transactional updates and simple (B+-tree) indexing. Each bucket is stored in the BDB volume using a key which is the concatenation of the stream ID and the starting timestamp of the bucket. Because BDB orders data on disk by primary key, this provides excellent locality when performing range queries in time from a single stream – the request will often correspond to a sequential read off disk.

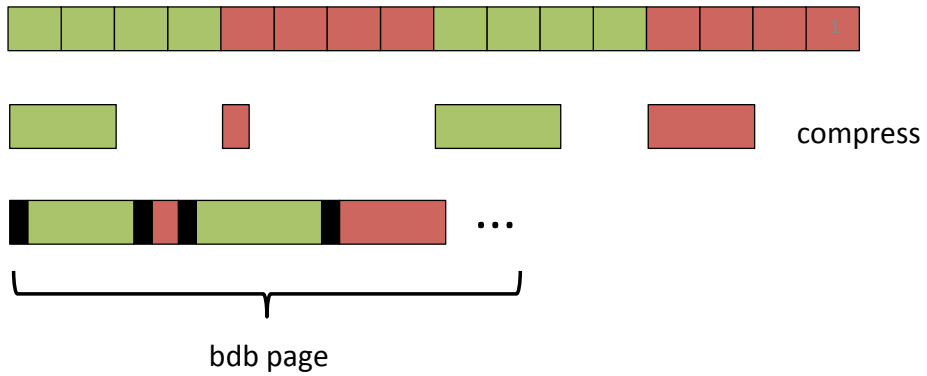


Figure 5.2: The `readingdb` historian first buckets data along the time axis to reduce the index size, storing values near other values with neighboring timestamps from the same stream. It then applies a run-length code followed by a Huffman tree to each bucket, resulting in excellent compression for for many commonly-seen patterns.

[1] `https://github.com/stevedh/readingdb`

5.2.1 Bucket sizes

Choosing proper bucket sizes is a design decision with several considerations. Within a bucket, the query engine must perform a linear scan to locate a particular reading; therefore, buckets should not be so large so that this scan becomes a bottleneck. Furthermore, the entire bucket must be decompressed to access any readings in it, putting more pressure on the designer to keep buckets relatively small. The main reason to increase bucket size is to limit the number of entries in the B+ tree index of all keys. This is an important optimization, since we often store tens of billions of readings within a time series archive. If each individual datum is placed into the index, the resulting B+ tree is very large; therefore, we should ensure that we reduce the number of tree entries significantly. A final consideration specific to BDB is that keys and values are stored, concatenated, within pages on disk; these pages are the unit the BDB I/O scheduler submits to the operating system. If a key or value is larger than a page, it is stored within a separate "overflow" segment on disk; the documentation notes that this tends to be inefficient because it destroys the I/O locality achieved by ordering keys on disk. The BDB page size is configurable up to 64K, and the documentation notes that the ideal loading is to have two key/value pairs per disk; additionally, since keys and values are not split across pages, it is also possible for there to be significant external fragmentation under certain usage patterns.

Based on these considerations, we chose a 16K page size, and buckets containing not more than 300 individual readings. 300 readings means that our smallest buckets store five minutes of data. A full bucket with 300 entries which cannot be compressed corresponds to around 5K of data, smaller than the page size and ensuring we achieve good utilization of most pages. This assumes that the finest level of time resolution is one second; for higher-frequency data, more levels of buckets are necessary.

5.2.2 Client Interface

Clients access stored readings using a client library; we have developed Python (implemented in `C`) and Java versions, which make parallel requests on the data store in order to maximize throughput when accessing a large number of streams. The client interface provides methods optimized for common workloads: retrieving data from a time range, or data before or after an "anchor" timestamp. Each of these operations can be performed using the index, since in a B+ tree locating neighboring keys is efficient. The Python bindings return data using a NumPy matrix data structure, in which both timestamps and values are represented using floating point values. Despite the fact that using a floating point number for timestamps is somewhat undesirable, keeping the data in a matrix is very desirable – each datum is simply a bit vector similar to a `c` array in memory instead of a native python object. This is important because an alternative strategy where each datum is represented using a Python object creates significant pressure on the garbage collector, as millions of small objects are quickly allocated and freed. Furthermore, this implementation technique makes all of NumPy's efficient vectorized operators applicable to our data.

```
# add/delete interface -- insert data into a stream
db_add(int stream_id, list data)
db_del(int stream_id, int starttime, int endtime)

# query interface
db_query(list stream_ids, int starttime, int endtime, int limit=None)
db_prev(list stream_ids, int reftime, int limit=None)
db_next(list stream_ids, int retime, int limit=None)
```

Figure 5.3: The complete `readingdb` interface.

`readingdb` provides a simple interface to client applications; as a result this piece is easily decoupled and used independently. It is designed with the common use cases of range and anchor queries in mind, shown in Figure 5.3. The interface avoids several common pitfalls of other time series systems which make them inappropriate for sensor data – all timestamps are preserved exactly, and data are not required to be sampled at a pre-defined periodic rate. Data can be added with any time stamp; either appended to the "end" of a series, inserted before any current data, or overlapping existing data; in the case of overlapping timestamps, the last write will win. We have found that violating any of these principles makes a system much less attractive to practitioners, since it forces them to make decisions about the sampling rate of sensors and import order of existing databases which then cannot easily be revisited. We have found that implementing a system with good performance which respects these constraints to be possible.

5.2.3 Compression

Many algorithms have been published for compressing time series data; it has frequently been thought necessary to discard the actual readings and keep only approximations of old data. Implementors or system administrators may apply policies which either down-sample or construct approximations with some known error bound of older data. We feel that these lossy compression techniques make increasingly less sense, as storage costs continue to plummet and the cost of maintaining even large volumes of old data becomes negligible. However, compression is still desirable for other reasons; the increasing cost of IO relative to processing means that it is often faster to read a compressed block off disk and then uncompress it than reading the raw and avoiding the subsequent processing.

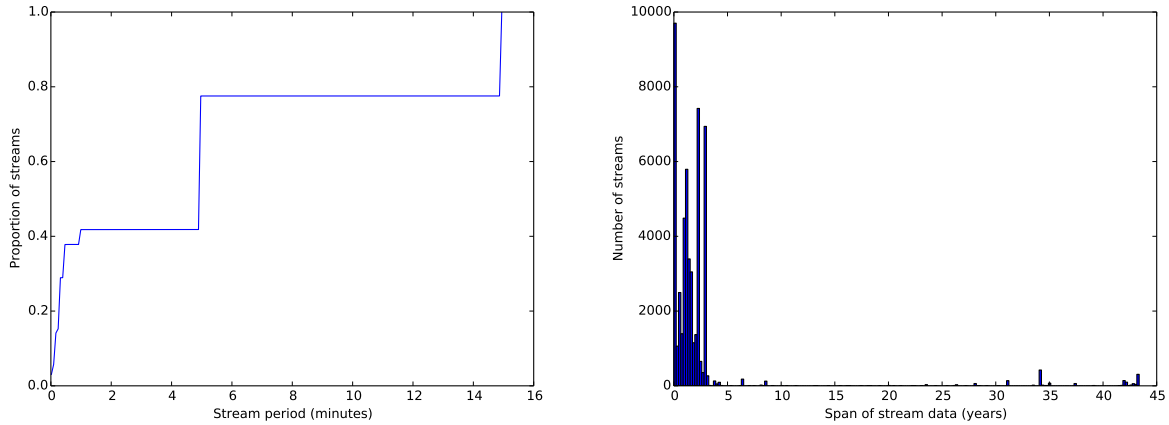
Therefore, `readingdb` implements lossless bucket-level compression of sequential readings; this allows the compressor to take advantage of locality in both time and the reading values. Entropy is extracted from stored data in two phases: first, deltas between successive readings and timestamps are encoded with a variable-length "zig-zag" integer code, with repeated values elided; this is placed within a Google protocol buffer, which performs this

compression efficiently. Secondly, we apply a fast Huffman-tree based code to the result using `libz`, which further compresses common byte-sequences.

5.3 Evaluation

We evaluate our solution on a few metrics, including how well the our compression performance on real-world data (from the sources listed in Table 4.4); how our solution performs on the common use case of retrieving the latest data point (*e.g.*, queries needed to populate a dashboard); how we perform relative to SQL database products on a standardized workload consisting of trickle inserts and batch reads; and how well our store scales to a reasonably large volume of data. The basis of our evaluation is an installation of `readingdb` at Berkeley running over the last four years, continuously collecting real-world sensor data from a heterogeneous selection of sensors. The input data are trickled in from sMAP sources, and queried by a variety of dashboards, analysis jobs, and researchers. Since we capture metrics about the system's performance within the system, we are easily able to extract a variety of performance metrics.

5.3.1 Scale



(a) Data period CDF from our installation of `readingdb`

(b) Histogram of time spanned by streams

Figure 5.4: Statistics about the sampling rate and time spanned by 51,000 streams stored in a large `readingdb` installation.

Figure 5.4 has an overview of the characteristics of the data stored in our Berkeley `readingdb` installation. In Figure 5.4a, we show a CDF of median sampling periods of all streams in our data set. We see that around 45% of streams are sampled more than once

a minute, with the remainder sampled at 5 and 15 minute intervals; these are generally imported from other data sources. Figure 5.4b shows a histogram of the amount of time spanned by different streams – the interval between the first and last data point. Many of the streams have been sampled for around the lifetime of the research project (several years), with a small number extending for as many as thirty years; for instance, historical records of campus energy billings.

Overall, the installation in question was storing 25,812,054,414 distinct readings, with an on-disk of around 90GB when we extracted these statistics, for an average per-record cost of approximately 3.5B – note that this figure includes some unallocated space due to external fragmentation, all B+-tree indexes, and other internal BDB data structures. Since the time of this snapshot in August 2013, the on-disk size has increased to 164GB, based on an average insert rate of 365 readings/second for April 2014.

5.3.2 Compression

We next examine how effective our compression technique is on our real-world data streams. Shown in Figure 5.5, we extract all of the buckets from our `readingdb` instance, and examine the compression ratio achieved by comparing the raw data structure size with the size actually written to disk; this figure displays a histogram of these values. We can clearly see a significant number of streams which compress to nearly zero bytes – these are primarily discrete or constant-valued streams such as switch position, or control-system set points that rarely change. We also see a significant number of streams with around a 25% compression ratio – these are primarily from the Cory Hall monitoring project, which consists of a large number of similar streams.

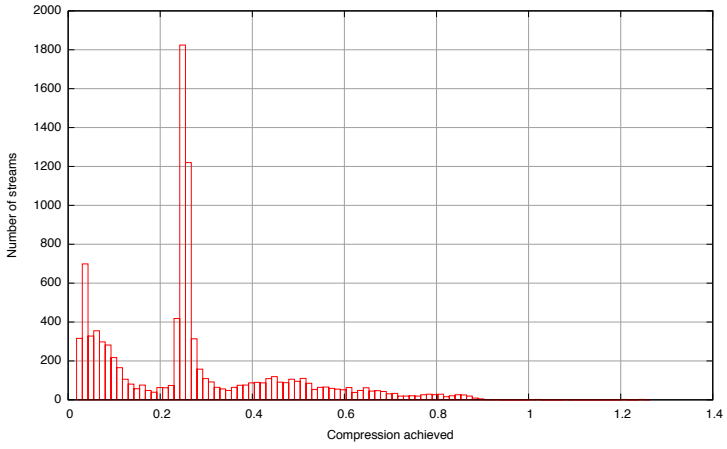


Figure 5.5: Compression achieved across all streams in the time series store at a point in time. A significant number of streams are either constant or slowly changing, which result in 95% or greater compression ratios. The spike in streams which are compressed to about 25% are plug-load meters which noise making them difficult to compress.

5.3.3 Latency

An important use of `readingdb` within BOSS is driving dashboards and system optimizations, while running both monitoring and controlling applications. These applications are typically interested in recent data from the set of time series they are interested in; for instance, retrieving all of the "current" temperatures in a space. Because of the index on time, retrieving data relative to a particular time stamp is very efficient; furthermore, constant trickled inserts ensure that the database pages with the latest data are kept in the buffer cache. Figure 5.6 shows the latency histogram for sequentially retrieving the latest point from a random set of streams on our archive, concurrently with the normal production read and write load. The 95th percentile latency is under 3ms, and the median is around 1.4ms. This clearly shows the latest data being in memory, since that is less than the disk seek time; furthermore, the store exhibits very predictable latency under this access pattern. Finally, it mitigates some concerns around the store not being optimized for fast access for queries which span the time series dimensions; for instance, this implies retrieving the latest data from 1000 streams would take around 1.4 second; this leaves open room for additional optimizations.

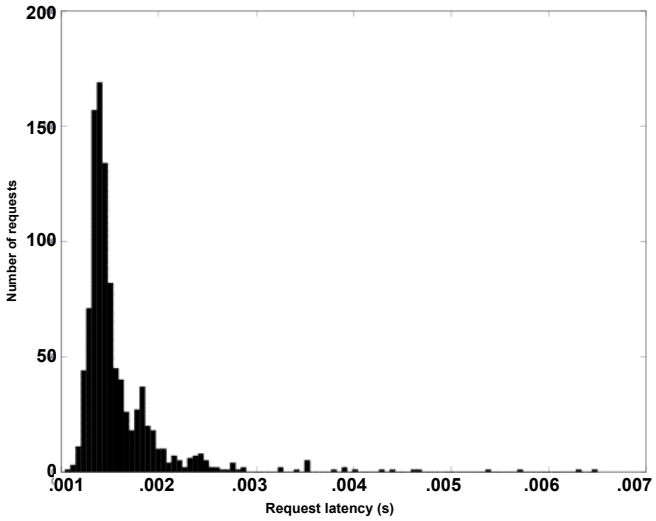


Figure 5.6: Latency histogram in milliseconds querying the latest piece of data from a random set of streams in the store. Because the cache is kept warm by inserts, the latest data is always in memory in this snapshot.

5.3.4 Relational Systems

Finally, we compare `readingdb`'s performance with several popular open-source relational databases. Our micro benchmark compares `readingdb` to two popular relational database products – MySQL (using both InnoDB and MyISAM storage engines) and PostgreSQL. For

this experiment, we start each storage engine and trickle load an empty database with synthetic data, simulating writing a new data points incrementally to 10,000 different streams. While loading this data, we periodically stop and examine the size of the on-disk files, the insertion rate achieved, and the query rate achievable when loading a fixed-size set of data. We attempt to normalize settings across the databases as much as possible, giving them equal cache sizes and matching the default `readingdb` durability setting, that data are committed to disk before returning to the client.

In the first result, Figure 5.7a, we look at the size on disk of the four different solutions. For the relational databases, there are essentially two sources of space overhead: per-row overhead added for internal database bookkeeping structures, and overhead implied by the index (normally a B+ tree), since the database engine must maintain additional references to each row within the B+-tree structure. When used to store time series data, these overheads become significant since it is not uncommon to have billions of readings in a data repository. We estimate the per-row overheads for MySQL to be 18 octets and PostgreSQL 23 octets. The value of time series specific compression is clearly visible here – uncompressed, Postgres and InnoDB are over 700MB, compared to only 50MB for `readingdb`. This has salutary effects for query-time performance, since `readingdb` must read significantly fewer bytes off disk; it also can make more effective use of cache, since data are compressed in memory until accessed. One potential improvement we did not explore was running the relational databases on a compressed filesystem; we expect this would narrow the distance some, but not all of the way since there are still high per-row overheads, and the compression will not take advantage of the time series' unique structure.

We next compare insert performance between the four systems, in Figure 5.7b. Here, the MySQL with MyISAM is the clear winner. To explain, we must understand the ways Postgres and InnoDB differer from MyISAM. The first two managers store data essentially sorted on disk by primary key, which in this case is the `(stream_id, timestamp)` tuple. Therefore, inserts may require reordering existing data; `readingdb` has a fundamentally similar approach. We see these three engines clustered together at the bottom, with `readingdb` outperforming the other two by a constant factor of 1.5-3x – this is due to the bucketing approach limiting the amount of index rebalancing required. In contrast, the MyISAM storage manager simply appends new data at the end of a log file on disk, without attempting to order it in any way; this makes inserts extremely inexpensive, at the expense of query-time performance as we can see in Figure 5.7c. In query performance, `readingdb` is the clear winner, helped by its ability to minimize IO through compression, and the high degree of locality achieved through bucketing. InnoDB pays for its fast inserts here, coming in at less than 25% of `readingdb`'s performance; retrieving a range of data requires scanning much of the log, because data from different streams has been interleaved as it was appended.

Overall, the 1.5-4x performance improvement, along with the significant disk savings make `readingdb` a performant system for storing time series. Furthermore, we believe that the results achieved in a relatively general time series storage framework validate an approach of developing special-case solutions for systems dealing with streams of scalar data; we have many ideas of how to further improve performance for this workload, and believe several

other performance multiples are possible with the proper engineering approach.

5.4 Related Work

There is a rich history of related work in storing and accessing time series, originating from a variety of communities. Control systems have long needed to maintain records of the evolution of various process variables; sometimes called "trend logs." Known as the **historian**, there are the most similar in spirit to our efforts here. Developed by industrial systems producers such as GE, WonderWare, or OSISoft, these systems use a variety of techniques to efficiently store and query time series data; for instance wavelet compression [18], dimensionality reduction [59], and approximation [36]. These are often used to store a large amount of data, decreasing resolution of data as it moves further into the past. Although an important technique in certain domains, they are not necessary in the building controls space where data rates are relatively low given the declining cost of storage. Based on our results, we believe an inflection point has been reached, where it is not longer necessary to degrade old data from monitoring systems, and the data can be saved essentially "forever." These systems are often accessed using a protocol like OPC-HDA – OLE for Process Control/Historical Data Access. HDA is a distributed API providing integration support for a variety of applications making use of historical data at approximately the same level of abstraction as the `readingdb` interface. Given no open alternative to these products, we developed `readingdb` to show the performance left on the table when using conventional systems, and to form the foundation for what comes next in our ultimate contribution – a significantly richer and more useful interface to time series.

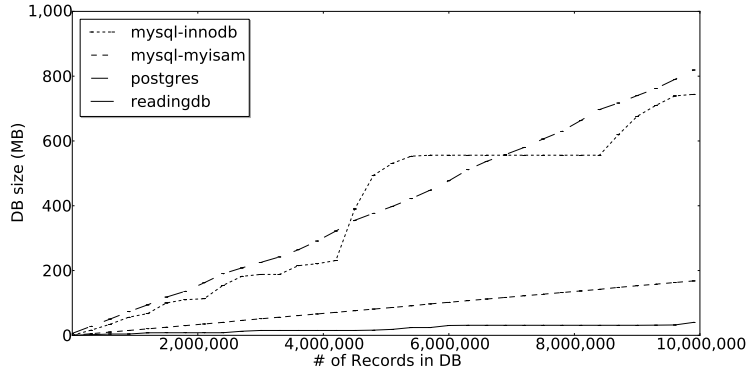
Another family of tools originates from the computer **systems monitoring** work. Early solutions such as Ganglia [71] and more recent tools such as Graphite, RRDTool, and Nagios [40, 95] all contain tools for monitoring system performance and other application metrics over time. Several of these tools make decisions which make them poor fits for monitoring applications; for instance RRDTool requires users to pre-allocate space for time series, and then stores data in a circular buffer with a fixed sampling rate without recording exact timestamps. Given that real-world data is lossy and intermittent, these design decisions are limiting for monitoring data.

Recent tools have also been built on top of large-scale data storage platforms like HBase and HDFS [96]; in particular OpenTSDB, as well as Bolt [43], which is built on a Microsoft stack. OpenTSDB uses similar bucketing techniques as `readingdb` in order to store time series data within HBase, a popular distributed column store. These tools enable massive scale – metrics can span multiple computers, with few limits since the workload is embarrassingly parallel. We developed `readingdb` before these tools existed, or at least were mature and believe it still has several advantages; although a single-node solution, it can be installed even on relatively underpowered hardware; because it is very efficient, it will still perform well.

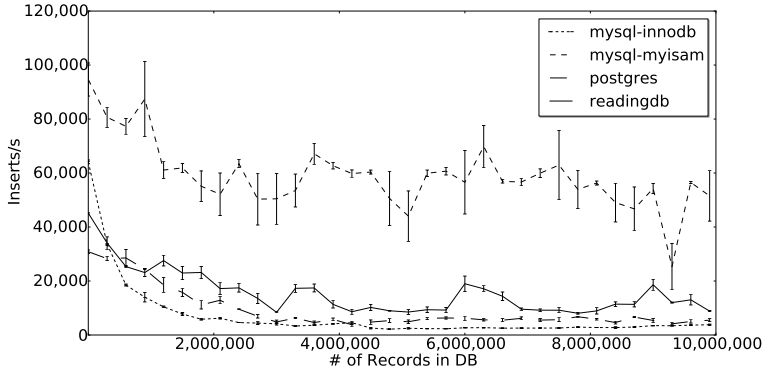
Gray *et al.* review the requirements of scientific data management and processing [41],

expanding on the NASA taxonomy of data into *Level 0* data, which is raw, *Level 1* data sets, which have been calibrated and rectified, and *Level 2*, where the data has been combined with other information. They identify the metadata problem of tracking the provenance, manipulation, and anything else necessary to understand the data. Many domain-specific systems, schema, and ontologies have been built up to meet this need, such as NetCDF [52] for array-oriented scientific data, OpenGIS for for geospatial data [11], and UCD for astronomy. Unsurprisingly, the authors are proponents of using relational database technology in order to meet the challenge of growing data volumes, arguing for the importance of data independence and views in maintaining software compatibility and enabling parallelism. `readingdb` is a good fit for storing Level 0 and Level 1 datasets, and as we have discussed has several performance advantages over relational systems. However, these other pieces of work do make evident that it is not a general solution but a specifically tailored one; although some data such as a GPS trace could be encoded into time series within `readingdb`, other forms of data such as GIS or imagery data benefit from their own specialized solutions.

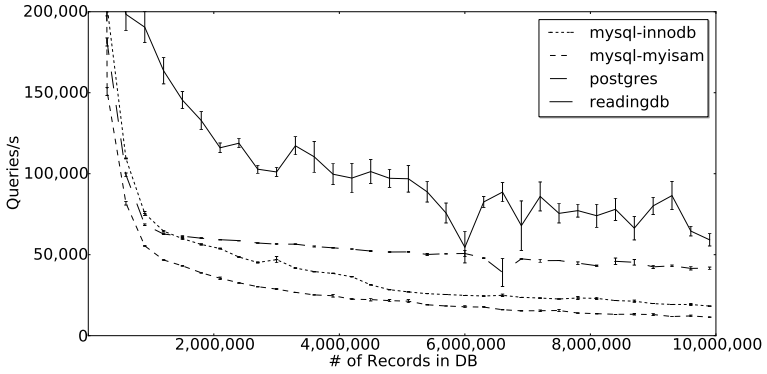
The Calder system [68] examines the same style of computing as Stonebraker [100] in the context of a dynamic weather forecasting application. They construct a list of "unique" requirements for data-driven scientific workloads: heterogeneous data formats, asynchronous streams, variable event size, relative timeliness, compatibility with a large system, changing computations over time, and domain specific processing. The event rates they consider are as fast as 1Hz, but are predominantly slower. Queries use the Globus OSGI framework, and a pub-sub layer for event routing. Their system supports monotonic time-sequenced SQL statements, with a subset of full SQL supported (no group-by); there is also support for custom functions. Many of the workload requirements analyzed overlap with the design goals of `readingdb`, in that we explicitly build for low-latency access, support different resolutions between streams since multi-level bucketing will perform well at a range of data frequencies. In the next chapters we build up a framework for domain-specific processing to address nearly their entire use case.



(a) Size of on-disk files



(b) Insert performance



(c) Query performance

Figure 5.7: `readingdb` time series performance compared to two relational databases. Compression keeps disk I/O to a minimum, while bucketing prevents updating the B+-tree indexes on the time dimension from becoming a bottleneck. Keeping data sorted by stream ID and timestamp preserves locality for range queries.

Chapter 6

Time Series Data Processing Model

In this chapter, we develop an abstracted processing model for time series data. This model helps users of BOSS develop the modeling and mitigation stages of their system analysis, both of which make extensive use of data. We build on several pieces of previous work, including array databases [99] and processing models [1, 117]. We take a top-down approach to this section, first identifying what it is to clean time series data, and then developing a simple type model for expressing operators which compute derivative streams.

6.1 Data Cleaning

Rather than attempting to build a completely general processing framework for time series data, we focus here on designing an approach for specifying transformations which are applied to produce "clean" data. Using a declarative approach, the implementation is free to materialize cleaned versions of the underlying data, or generate them on the fly using the functional specification; whatever is determined to be most efficient. In this section, we provide an overview of the types of operations commonly applied to clean time series data and develop a computational model allowing these operations to be concisely expressed.

6.1.1 Dealing with Time: Subsampling and Interpolation

A ubiquitous characteristic of physical data is that the timestamps and associated data require some degree of processing before the data can be combined with other data sources. The primary cause for this need is that sensors often sample asynchronously at ostensibly (but not truly) fixed rates, generating time series where time stamps contain jitter and skew, and therefore do not line up with other sources of data. For the purposes of analysis, the time stamps from multiple streams must be rectified so that it appears that the instruments were sampling synchronously (even if this is not the case). Additionally, it's common to wish to combine data from multiple sources, sampled at multiple frequencies.

The ultimate goal is to place all data within a common time base for correlating events between different streams – for instance, creating the appearance that all data were sampled synchronously every minute. To adjust the timestamps, the data must be manipulated appropriately; this necessitates a model of the data. Two standard techniques for creating this appearance are *windowing* and *interpolation*.

Data windowing involves compressing all readings within a window down to a single point using a compressive aggregation function. For instance, a temperature sensor might be sampled every 30 seconds; to down-sample to five-minute resolution, an analyst might wish to use the average (mean) value in each five-minute period. On the other hand, if the underlying process is discrete, such as a light-switch position, it would be more reasonable to perform aggregation by counting the number of transitions or the final state inside of the window. If the underlying sensor is accumulative, for instance a water meter reporting the cumulative number of gallons, simply taking the first or last value in the window might be most appropriate.

Data interpolation instead revolves around constructing a local functional model of the underlying process using measured data or *a priori* models, and then evaluating it at a new set of points. For instance, simple linear interpolation assumes that the process moves in a straight line between measured values; higher-dimensional interpolation methods use polynomials or splines in order to attempt to estimate the value of the function between measurements. Whether and how a particular time series should be interpolated is a decision which depends on knowledge underlying process. This model is *local*; for instance, a linear model only requires two neighboring data points to make a prediction; higher order models may require additional (but still fundamentally local) state, making these tasks amenable to "once through" processing with bounded local state.

6.1.2 Data Filling

A related problem to time adjustment is determining values during periods when the underlying data source was unavailable – interpolation and windowing are in fact versions of this, simply operating on short time scales. Over longer periods, analysts use a variety of techniques to provide predictions of values during periods of missing data. For instance, if producing monthly consumption totals, analysts might wish to use a comparable month's value to approximate a period of missing data. Alternatively, if consumption is known to be a function of some other variable (*i.e.*, temperature), for which data is available, it might be more appropriate to build a functional model of that relationship using available data, and use that model to predict values for periods during which data is not available, perhaps fitting to the endpoints.

Data filling has a somewhat more complicated computation framework than time adjustment, since it may require access to non-local data, or perform significant computation on the data in order to optimize the model; for instance, an ARIMA operator may perform parameter optimization on the entire time series in order to discover an appropriate parametric model. Whenever significant computational effort is expended in deriving a functional

representation of the data, it may be desirable to save the output of the model; either as a materialized set of computations on input data, or as a functional representation which can be queried in the future to generate new predicted values using the existing model.

6.1.3 Calibration and Normalization

Raw data sampled by instruments typically cannot be used directly, requiring various transformations for conversion into accurate engineering units for use in subsequent analysis. The type of calibration and normalization necessary is instrument specific, but typically follows one of several patterns. In a simple example, a data stream may represent the output of an analog sensor; if the sensor is sampled with 12 bits of resolution, the output value will be in $[0, 4096)$. It is common to apply constant factors or linear models applied to this reading to convert the raw ADC value into an engineering units.

There are several ways in which this simple procedure may become more complicated. For one, the calibration coefficients may themselves become time series; for instance, if the instrument is periodically re-calibrated with new parameters. Second, the conversion to engineering units may depend upon other time series values present, as in a temperature calibration applied to the raw data. In this case, converting raw values to engineering units requires applying a function to multiple inputs; for instance, applying a quadratic temperature correction to raw count data; of course, further complicating matters is that the coefficients within the polynomial may themselves change over time.

6.1.4 Outlier Removal

Another area of interest when processing raw data is the removal or interpretation of spurious or otherwise exceptional values. Outliers come from many sources: instruments may become disconnected and continue produce readings of zero or some other constant value; transient effects such as a glint of sun or other physical phenomena may lead to transient readings which should be discarded; devices may have internal glitches; or many other sources. Propagating these errors further into the analysis pipeline is problematic, since once rolled up within an aggregate or otherwise condensed, it becomes impossible to determine that the actual result was influenced by such a transient.

Outlier detection and removal strategies may vary from the very simple to very complex, but involve applying additional domain knowledge about what is a reasonable value within the system to the problem. For instance, only certain values or transitions may be physically meaningful.

6.1.5 Analysis Operations and Rollups

Not all analyses can be easily expressed within our data cleaning model; in particular, algorithms which require large amounts of working memory or perform iterative computation on the entire data set are better served by a fully general analysis system designed for

high level computation. We use an optimized data cleaning service to perform perform the initial rectification steps on data, using its simpler operations to efficiently transform raw sensors to usable data. None the less, it is useful to provide a certain amount of simple analysis functionality which can be efficiently located within the access system to provide simplified, quick roll-ups of the data present in the system. Furthermore, simple analyses often provide the basis for implementing other important data cleaning functionality such as outlier detection; consider rejecting 99^{th} percentile data.

6.1.6 Processing Issues in Data Cleaning

From these varied use cases, we can identify a few issues which any solution performing data cleaning must address. First, we note that many, although not all algorithms which are used for data cleaning are *incremental*; that is, they can be evaluated without examining the entire data series. For instance, windowing and interpolation, unit conversion, calibration and normalization, and computing certain derived products can all be computed using streaming algorithms with limited buffers. Not all cleaning tasks fit this model; outlier rejection and data filling often require multiple passes over the data; *e.g.* to first build the model and then to compute over it; although there are sometimes streaming analogs, some tasks like computing a percentile cannot be achieved without first examining the entire data set. Secondly, much of the functionality requires queries over the metadata as well as the data – for instance, applying a temperature correction to raw data will require joining the raw stream with an asynchronously-sampled set of data from a temperature monitor.

In the next section, we design a framework in which we can place our time series data, allowing for the implementation of these operations within two popular models of streaming computation.

6.2 A Model of Data Cleaning

In order to inform the design of a computer system which will clean data, we first briefly review an abstracted version of the problem. What do the data look like, and how are they modified as processing advances? In traditional databases, each row is modeled as a tuple, who's elements are determined by the relation. However, the ordering of tuples within a relation is normally unspecified (although of course, an ordering may be imposed). In these databases at query time, these tuples are pushed or pulled through an operator graph using an iterator interface.

The key difference when dealing with time series is that data are nearly always generated, indexed, and accessed by the time dimension. Furthermore, individual tuples are small and many time series operations can be implemented much more efficiently when data are batched, with a block of data passed through an operator. These simple realities suggest that the time dimension of these relations should be treated specially – that we can leverage these additional semantics to develop both a simpler and more efficient processing framework.

6.2.1 Data model

We can think of a time series generated by an instrument as a special kind of matrix – one which is of infinite height, and where one column represents time; for we'll call this dimension (the height of the matrix) T. Because time is infinite, we can only ever examine a portion of the matrix between two reference points; for instance data within "the last hour." The other dimension of the matrix represents the dimension of the measurement being taken. A scalar sensor such as a thermometer would only produce a single value; more complicated measurements are vector valued (*e.g.*, GPS position, 3-axis acceleration, an image). In fact, the shared relationship between the columns of this sort of time series relationship is that they were sampled *synchronously* (or at least, it is made to appear that they were); all measurements within a row of this time series matrix have the same time stamp. We will call this dimension C, for the number of "channels" present.

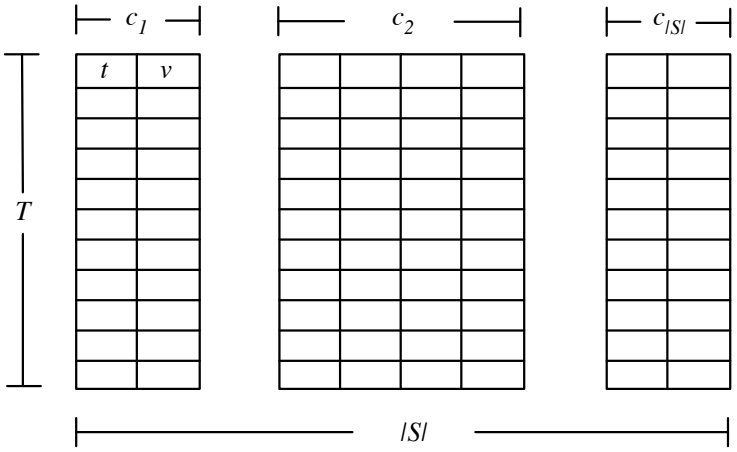


Figure 6.1: Each time series being processed is a matrix with a notionally-infinite height (each row shares a time stamp), and a width of c_i, representing the number of instrument channels present in that series. Sets of these may be processed together, forming the set S of streams.

Most sensors in the world do not sample synchronously; dealing with this fact was the subject of Section 6.1.1. In order to allow for this use where individual series are not already aligned, we allow for a third dimension: S, or the series dimension. This dimension consists of an unordered set of separate time series which have not yet been aligned or coerced into a single synchronous matrix. The elements of this set may be bound to any time series within the system – for instance, we might query "all outside temperature series within Berkeley, CA." The result would be a set of series, each with its own independent sampling rate, engineering units, data source, and other distinct attributes.

These three dimensions are illustrated in Figure 6.1; in this example, $|S| = 3$, since there are three series present, and $C = [2, 4, 2]$, corresponding to the number of channels in each

of these three series. Nearly all physical data generated from sensors can be placed into this model, as well as the results of processing operations; particular operations of the style we are concerned with, that is, data cleaning operations. All of the cleaning operations we are concerned with can be expressed in **operators** which operate along these three dimensions in different ways. Because the output of processing data in this model is more data in this model, it is simple to create a pipeline of processing elements which examine a set of input data and produce a mutated set of output data. More generally, this structure could take the form of a directed acyclic graph (DAG) of processing elements; this is essentially the same principle in which many data processing systems ranging from relational databases to stream and complex event-processing systems operate with.

Within this model, each time series is also referenced by a unique identifier, and a set of metadata linked to that identifier. This allows us to identify a particular data product globally, and to look them up by querying the metadata relations. Metadata is generally much smaller than the data, and changes infrequently relative to the data. In this processing model, we cary forward our simplifying assumption from the hardware presentation section, treating metadata for a particular time series as constant over time – that is

6.2.2 Operators

Functional Operators

Many operations can be expressed in a functional form – it is common to want to apply a windowing operator, filter, or other operation to all or some of the input set S. A subset of the operators which we have found to be important is present in Table 6.1

Operator	Description
`window(`*op*`)`	apply inner operator to windows in time
`sum(`*axis=0,1*`)`	compute the sum across time or streams
`units()`	convert input data to a canonical set of engineering units
`workday()`	filter data to return only data within a workday
`ewma(`*alpha=1.0*`)`	smooth data using an EWMA
`meter(`*period="day"*`)`	compute usage during a time period from a sequence of meter readings
`missing()`	filter timestamps and values where data are missing, indicated by the missing data marker

Table 6.1: A subset of operators well expressed using functional form. They allow for resampling onto a common time base, smoothing, units normalization, and many other common data cleaning operations.

There are several common patters for functional operators governing how they modify the data they are processing; generally, we can thinking about how the operators mutate the three dimensions – $|S|$, c_i, and T. Some operators, the *universal functions* do not change any

dimension, and operate element-wise on all input data; examples of these operators are functions like `multiply`, `add`, `log`, `floor`, and may others. Those are common transformations which are basic mathematical primitives for constructing more complex processing.

Another common class of operators are *vector operators* which operate on one dimension of each input series, but do not modify the overall set of streams S. For instance `sum` can be considered a vector operator, which either produces sums across rows, in which case it produces an aggregate value for each timestamp, or across columns, in which case it produces a rollup across time for each column. In the first case of aggregation across time, the operator would have the following dimensionality transformation:

$$\texttt{sum}(\texttt{axis} = \texttt{0}) : (|S|, C, T) \Rightarrow (|S|, C, [1])$$

In other words, the operator compresses the time dimension of all streams to a single value, but doesn't change either the number of streams or channels. Alternatively, if compressing data across channels, the operator has the transformation type

$$\texttt{sum}(\texttt{axis} = \texttt{1}) : (|S|, C, T) \Rightarrow (|S|, [1], T)$$

Whenever an operator is applied which reduces the time dimension, there is an important question of how much data the operator must be presented with in order to produce a meaningful output. For instance, a sum across time in general will depend on how much and which data are present in the input to that operator. In order to address this concern, operators are first-class – they can be provided to other operators as arguments. Therefore, we can control which time periods are provided to `sum` in this example by specifying an outer `window` operator, which guarantees that sufficient input data has been accumulated before evaluating the sum.

Finally, some operators also mutate the set of streams; for instance `paste`, which performs the cross-stream join on the time column. This operator, explained in more detail later on, has the transformation type type

$$\texttt{paste} : (|S|, C, T) \Rightarrow (1, \sum c_i - |S|, |\cup t_{ij}|)$$

We have not found there to be many operators beyond paste which mutate the set of streams; generally preprocessing is performed in parallel on the input streams, which are subsequently pasted together (potentially in groups) for further processing. However, nothing in the language restricts adding more of these operators as the need for them is found.

Information Metadata and Provenance

The process of cleaning raw data is often accompanied by an *ad-hoc* process for storing metadata, describing changes made along the processing pipeline. However, this is subject to error and makes it difficult to automatically make use of pre-computed distillates and cleaned versions of the raw data. Within our processing model, we attach metadata to all time series in the form of key-value pairs. Whereas the time series data has specific

usage patterns we can leverage as well as a large volume making it desirable to exploit these patterns for efficient, the metadata has much more in common with the use cases of traditional (relational) databases. Within BOSS, these data (known as tags) often come from the hardware presentation layer – for instance, if processing the time series from Figure 4.4, the operator would have access to all of the metadata tags; `Metadata/Location/Building`, etc.

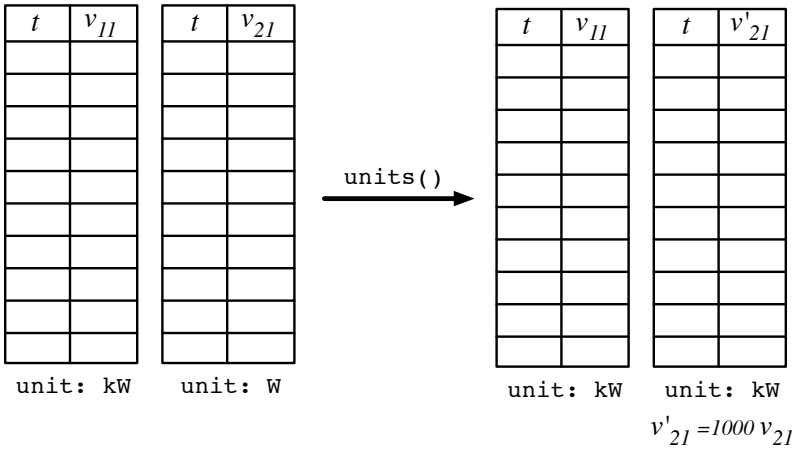


Figure 6.2: An example application of a unit conversion operator. The operator contains a database mapping input engineering units to a canonical set and uses the input metadata to determine which mapping to apply. The result is a new set of streams where the data and metadata have been mutated to reflect the conversions made. This is an example of a "universal function"-style operator which does not alter the dimensionality of the input.

In Figure 6.2 we show a unit conversion operator being applied to two input time series. Each input data stream has a tag named `unit`, who's value is `kW` (kilowatts) for the first stream, and `W` (watts) for the second. The operator being applied here, `unit` : $(|S|, C, T) \Rightarrow (|S|, C, T)$ is a universal function-style operator, in that it simply converts scalar values to a new set of scalar values without altering the overall dimensionality. However, this operator also examines the input metadata, looking up the `unit` tag in a directory to discover if the corresponding series requires conversion; in this case, the operator results in two output time series where the second stream has been mutated – it's values have been divided by 1000, to convert into kilowatts, and the `unit` tag has been updated to reflect the change.

6.3 Related work

There are a few key pieces of related work we have been influenced by in designing our processing model. The array model of SciDB [99] is particularly interesting, since time series

can easily be represented as two-dimensional arrays or matrices. SciDB also has the concepts of views which are created through the application of a functional operator, producing new matrices which can themselves be materialized and queried. In SciDB, these are implemented as `c++` functions; this concept is extremely important for producing cleaned distillates, the first step towards producing higher-level events from raw time series data. SciDB itself is currently a poor choice for streaming workloads, since it explicitly targets analysis of complete datasets and the documentation makes clear that it is difficult to append to an existing array.

Provenance tracking has been established as an important design goal for data processing systems, especially those for scientific domains [73], where practitioners deal with a large number of uneven-quality data sets. Provenance tracking is possible on a number of different of different levels of abstractions. For instance, the Provenance Aware Storage System [78] tracks provenance at a low level, by interposing on kernel-level filesystem calls to observe all scripts and transformations being made on data, and recording a log of analysis actions taken. Similar in many regards to taint tracking [32], it suffers from similar problems with a low level of semantic content of the operations, and state-space explosion. At a slightly higher level of abstraction, Spark [116] introduces the concept of Reliable Distributed Datasets (RDDs). These primitives are immutable sets of data which are generated either by loading raw data, or computed deterministically from other RDDs using a language closure. By tracking the dependency graph, Spark gains significant flexibility as to which products are materialized; however this dependency-tracking data also serves as a valuable record of which operations have been applied.

There are also a wide variety of techniques used for data cleaning, some very general and some domain specific. Generally, many features of SQL:99 are relevant – specifically user-defined functions (UDFs) either implemented in an external language or PL/SQL, and views. Analysts create specific UDFs which clean a particular attribute of a raw dataset, and use it to create a view on the underlying table which contains the cleaned version. In particular, Hellerstein advocates using SQL as a generic interchange language between different components [46], allowing each component the full power of a declarative approach. This technique has the prospect of preserving much more provenance than the fallback, would would typically involve writing a custom program which is used inside of an Extract-Transform-Load (ETL) paradigm. Our approach of having operators which track provenance is most similar to creation

Our time series operators provides a time series specific functionally for data cleaning, based on these general techniques for provenance capture and storage. They are able to generate concise, deterministic descriptions of common data cleaning operations which can then be used as in Spark for materialization, or as in an SQL view to provide a higher-level, cleaned data set for users to work with.

Chapter 7

Implementation of a Time Series System

To build on the ideas for time series storage and processing discussed in Chapters 5 and 6, we have implemented a small domain-specific language embodying these principles. The system, whose architecture is shown in Figure 7.1 uses separate storage engines for time series data and metadata, and consists of an application worker that implements a query processor for our Time Series Cleaning Language. The structure of the language is high-level enough to admit several different styles of compilation; in particular, many of the operators would be trivially parallelized and pushed down closer to the data. Our focus with this implementation has been to provide sufficient performance to build real applications and not close off our ability to apply many different database optimization in the future, without getting lost in the thicket of standard database optimizations which clearly are applicable.

In this section, we evaluate our the language for a few characteristics, with the main ones being *expressiveness* and *performance*. To examine the language's expressiveness, we look back at several applications developed in the course of various monitoring and control applications, and look at the how the application tasks can be expressed in the language's idiom. For performance, we examine some of the implementation techniques needed for good performance in our prototype implementation.

7.1 A Domain Language Approach to Data Cleaning

The language and its supporting runtime environment support efficiently acquiring data from the hardware presentation layer, storing this data, and implementing a processing pipeline of operators which operation on the data model described in Section 6.2.1. The language allows applications and users to *select* the data they are interested in on the basis of the metadata relations, and *transform* this data using a sequence of operators. Our language, called the Time Series Cleaning Language (TSCL), combines relational access to metadata with a declarative, pipelined sequence of operators used to perform cleaning operations and

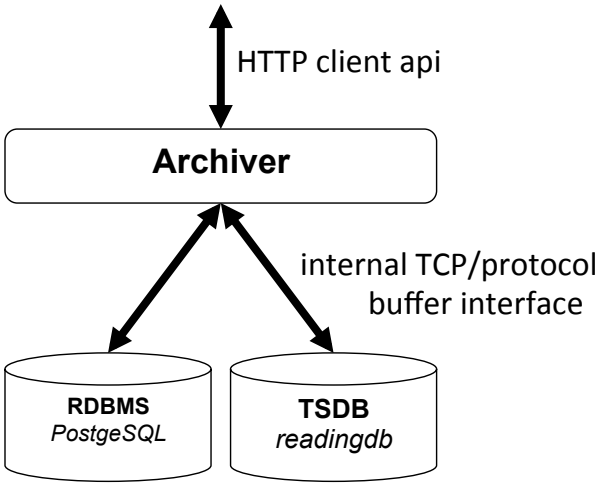


Figure 7.1: The archiver service implements the time series query language processor, as well as several other services on top of two storage engines; metadata is stored in PostgreSQL, while time series data uses `readingdb`.

allows users and programs to easily download filtered, cleaned versions of the data. Due to the language's structure, it allows many opportunities for optimizing common data access patterns.

We have implemented the language design described within the sMAP archiver; a software component which forms one of three components of BOSS evaluated in this thesis. The archiver uses the language in several places to present an API to applications over HTTP. Internally, the archiver relies on several components; a LALR parser generator uses a language grammar to compile user-generated queries to executable versions. Three compilation strategies are used: metadata queries on the entity-attribute-value tag schema are compiled into raw SQL queries which can be executed against a standard relational database. Queries referencing data are complied into closures which can be executed against the `readingdb` API, which is discussed below. Finally, operator expressions are compiled into an abstract syntax tree which can be evaluated against the data returned by the time series database.

7.1.1 Select

The first task applications face is simply locating the time series data in question using the time series metadata. Today, we have hundreds of thousands of streams in BOSS; this scale is relatively common-place in medium to large environments. In the future, the number of streams will only increase. To efficiently locate streams, one must make reference to an underlying schema or ontology which organizes the streams with reference to a model of the

world. The issue is that no one schema is appropriate for all time series; data being extracted from a building management system (BMS) may reference a vendor-specific schema, while other time series may be organized by machine, system, or many other schema. As a result, a system which requires a rigid schema is a currently a bad fit for a general building data system, because there are many different schema and it is difficult to automatically convert from one to another. Therefore, we choose to use a simple entity-attribute-value (EAV) data model, in which individual time series are the entities and are tagged with key-value pairs including metadata from sMAP.

TSCL queries specifying metadata queries follow a general grammar including a *where-clause*:

select *select-clause* **where** *where-clause*

The where-clause is a set of constraints matching the time series tags available in the metadata store; within our system. Therefore, it is common for the time series to be already tagged with basic information about their source; instrument model names, locations, units, and network location data such as BACnet point names and description fields. The where-clause supports basic boolean and filtering operations on tag values, enumerated in Table 7.1.

The use of this schema type allows users to pose queries such as

```
select * where Properties/UnitofMeasure = "kW"
```

or

```
select * Metadata/Instrument/Manufacturer ~ "^Dent.*"
```

In the first example, we select the metadata for all streams in units of kilowatts, while in the second we use a regular expression to search for all data where the instrument manufacturer begins with "Dent" – the maker of some of our three-phase electric meters.

Selection operator	Description
`=`	compare tag values; `tagname = tagval`
`like`	string matching with SQL LIKE; `tagname like pattern`
`~`	regular expression matching; `tagname ~ pattern`
`has`	assert the stream has a tag; `has tagname`
`and`	logical and of two queries
`or`	logical or of two queries
`not`	invert a match

Table 7.1: Selection operators supported by the query language. The selection syntax is essentially borrowed from SQL.

We have implemented this lookup system on top of the PostgreSQL `hstore` column type, which stores metadata for time series within a single column value representing a bag of key-value pairs. This approach builds on the metadata provided by the hardware presentation layer, and allows applications to locate any time series which is being stored by the service. In the future, it maybe be desirable to provide more relational-schema checking to this system once widely accepted information models are available for the specific problems we are solving. Internally, each metadata selection is compiled to a raw SQL statement, which is executed against this database. For instance, the first example query is into a more verbose query making use of the postgres-specific hstore query operators:

```
SELECT s.metadata || hstore('uuid', s.uuid)
FROM stream s
WHERE ((s.metadata ? 'Properties/UnitofMeasure')
    AND ((s.metadata -> 'Properties/UnitofMeasure') = 'kW')
   AND ((sub.public )) AND sub.id = s.subscription_id
```

At the same time, the query may reference underlying data through the use of a data selection clause. For instance, in the first example, the query could instead return the last reading from all matching streams using a modified query:

```
select data before now where Properties/UnitofMeasure = "kW"
```

Instead of returning the set of tags for the matching time series, this query instead returns the latest reading from each of them; it is implemented efficiently using `readingdb`'s db_prev method. When loading data, querying is a two-stage process – the worker must first evaluate the where-clause to identify the set of time series referenced, and then load the data using `readingdb`'s client API.

7.1.2 Transform

To support the use cases examined in Section 6.1, it is clear that users need robust support for interpolation, windowing, and filtering operators. However, some uses, such as temperature normalization and recalibration require more complicated support for algebraic operations; for instance, computing a polynomial to remove a quadratic temperature dependency or evaluating a weighted sum of vectors. The key to designing a language for this purpose is allowing users to concisely specify both types of operation, using familiar notation.

Applications may submit requests for transformations on data through the use of the `apply` statement:

apply *operator-clause* **to data** *data-clause* **where** *where-clause*

For instance, consider the `units` operator discussed in Figure 6.2; we could apply this operator to data with units of either kilowatts or watts with the query:

```
apply units to data in now -1d, now where units = 'kW' or units = 'W'
```

This query introduces several new features of the language:

- The `apply units` operator clause will instantiate an instance of the unit conversion operator, and pass data through it before returning it to the user;
- the `data in now -1d, now` data clause uses a range query instead of an anchor query to load data from the previous day; and
- the where clauses introduces a boolean operator (`or`) for metadata selection.

This example is also a use of *functional* operator notation, instantiating operators using a familiar functional notation – "`units()`" returns the operator which performs unit conversion. Functional operators may also take arguments, including other operators; consider instead

```
apply window(mean, field="minutes", width=10)
  to data in now -1d, now
    where units = 'kW' or units = 'W'
```

where we use the windowing operator to generates an operator which returns the mean of each 10-minute window of data.

TSCL also supports *algebraic* notation, which allows the use of infix notation and instantiates multiple operators from an algebraic formula. For instance, the "`* 10`" operator clause generates an operator which multiples all input data by 10. Algebraic notation introduces several complexities when evaluating the operators, which we discuss in detail in Section 7.1.2.

Multiple operators may also be piped together in order to process data sequentially. For instance, consider the combination of the window and units operator:

```
apply window(mean, field="minutes", width=10) < units
  to data in now -1d, now
    where units = 'kW' or units = 'W'
```

where we process right-to-left, first convert the input data to normalized units, before computing a 10-minute windowed mean over the processed data. The input to each stage of processing is an ordered set of time series, following the data model discussed in Section 6.2.1. Each operator can inspect both the operator data as well as the operator metadata; this is crucial for implementing operators such as unit transformations and other metadata-dependent computations common in data-cleaning. Additionally, each stage in the pipeline appends a description of the computation it performed – essentially, what arguments it was called with as well as any additional arguments which effect the output. This metadata provides the critical provenance chain of computation, connecting processed data to the original raw results. Finally, output time series from each stage are named within the same global UUID namespace as the inputs, by deterministically mapping the input stream identifiers

to new ones, based on the computation performed. This provides an efficient and scalable mechanism for locating shared computation within the processing graph.

Figure 7.2 illustrates this complete process for our example query. In the first stage, the worker extracts the where-clause of the query, and uses it to generate an SQL statement which returns the metadata of the relevant streams – in this case, two of them. This metadata is used to synthesize a call to the `readingdb` API, which returns the corresponding time series data according to the data clause. The data is placed into a set of arrays corresponding to the time series, which are then pushed through the operator graph. At the first stage, the `units` operator notes that stream 2 requires a unit conversion, and so computes the converted data. It also updates the uuid of this stream to reflect that the data has change, and changes the units tag associated with the stream. In the second stage, the windowing operator computes the appropriate compressive window of the data, reducing the time dimension. It alters both uuid's to reflect the new derivate stream's provenance.

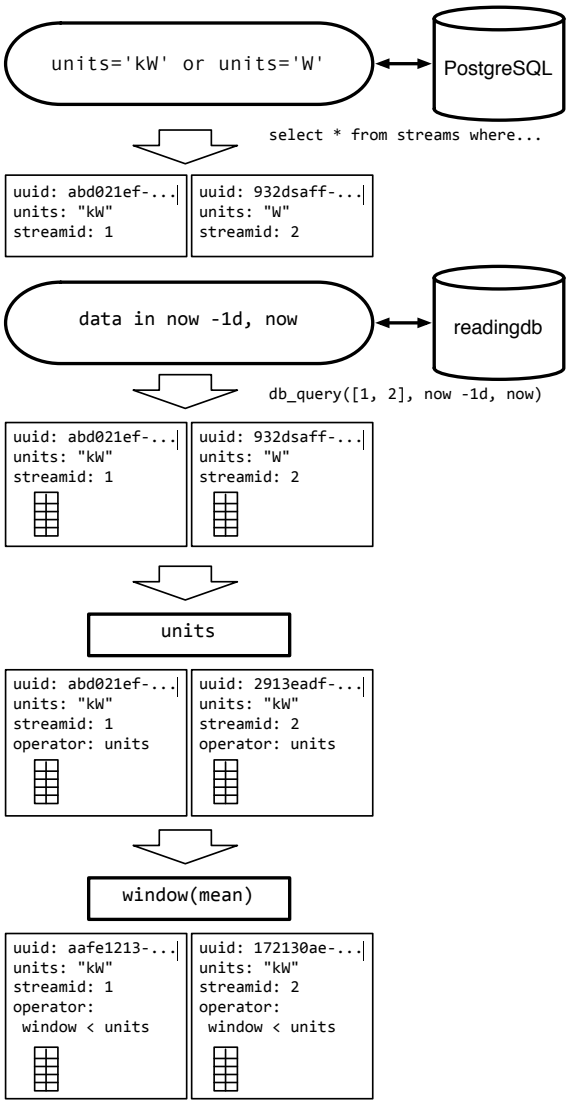


Figure 7.2: The complete pipeline needed to execute our example query. The language runtime first looks up the relevant time series using the where-clause, and uses the result to load the underlying data from `readingdb`, using the data-clause. It then instantiates the operator pipeline, which first converts units to a consistent base and then applies a windowing operator.

Pasting data

One key transformation of the data model explained in Figure 6.1 is known as *pasting* data. Pasting is essential a merge-join on the time dimension; to explain the need for it, consider the problem of adding two different time series to produce an aggregated value. Shown in Figure 7.3, initially these time series are separate vectors; perhaps originating from different electrical meters or generated by some other process. Since the time series are generated by different processes, there is no reason to believe that they will sample at the same rate, or in the same phase. Therefore, some position must be taken on how

to combine readings into before the addition is performed.The result of pasting these streams is a single matrix containing one row for each time value in the input data, with the columns corresponding to the input series. Missing values are represented using a missing data marker within each row of the merged matrix.

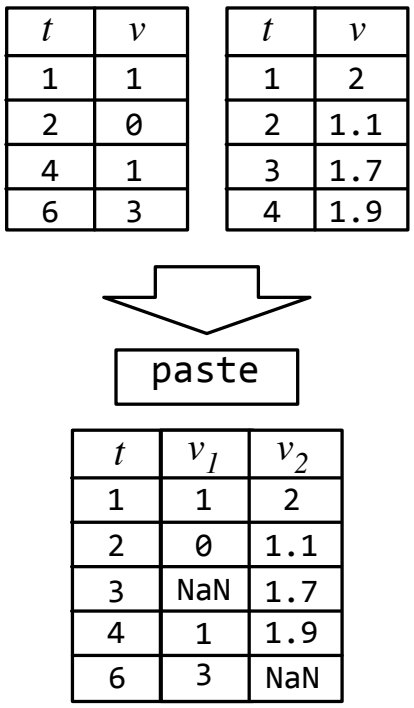


Figure 7.3: The paste operator. Paste performs a join of input time series of the input streams, merging on the time column. The result is a single series which contains all of the input timestamps. The transformation dimension in this example is $\text{paste} : (2, [2, 2], T) \Rightarrow (1, [3], T)$.

Within the processing system `paste` is defined as the operation which merges a set S of streams, each with width c_i into a single stream with a width of $\sum c_i - |S| + 1$. In the time dimension, a row is produced for each unique timestamp in any of the input streams; values from input streams with matching timestamps are placed in the same output row. As a result, the output in the time dimension has a size which is the size of the union of all input timestamps. Once data have been coerced into this format, various additional computations are relatively trivial – aggregations and rollups correspond to operations on the rows and columns of this matrix.

Algebraic Operators

Functional notation is quite cumbersome for some operators; in particular, operations which are naturally expressed as formulas. For instance, consider applying a quadratic temperature correction to a stream of raw sensor data. One might wish to compute the calibrated sensor output as a function, for instance,

$$c_i = r_i + k_1 * t_i + k_2 * t_i^2$$

In this case, r_i is the raw sensor reading at time step i, and t_i is the temperature at the same time step; k_1 and k_2 are polynomial coefficients. Therefore, our query language also supports the use of binary infix notation to concisely represent this formula.

Constructing formulas builds on the ability to address time series by their metadata; in this case, we will use it to construct the formula in this example. For instance, suppose the two streams in question have already been tagged with a tag name `Type`; the raw data stream has type `raw`, while the temperature stream has type `temp`. We can then construct a query referencing these elements, shown in Figure 7.4. In this case, we have included the constants (2.68 and -27.16) directly in the formula, although these could also reference other time series.

```
[Type = "raw"] + 2.68 * [Type = "temp"] + [-27.16 * [[Type = "temp"] ^ 2]
```

Figure 7.4: An algebraic operator expression which computes a quadratic temperature calibration from two time series: a raw sensor data feed, and a temperature data feed.

This special syntax allows for idiomatic construction of simple formulas, referencing data by their metadata. For instance, consider the inner part of this expression:

```
[Type = "temp"] ^ 2
```

The brackets operator acts as a selector on the set of input streams, applying the inner operator only to time series in the input which match the metadata query – in this case, having the `Type` tag equal to `"temp"`. In fact, this operator can also be expressed using functional notation, with the help of the `where` operator:

```
where("Type", "temp", pow(2))
```

The where operator applies an inner operator only to time series containing a particular key-value pair; it is clear how cumbersome this becomes for even modestly complicated formulas.

Provenance Transmission

Along with computing on the data as it is processed, the operators also have an opportunity to read and edit the metadata provided to the next operator in the system. As a result, the final result contains not only the modified data elements but also mutated metadata. The most important piece of information an operator attaches as part of the its operation is a concise, executable description of the operator. Typically, the operator will also mutate any other relevant metadata as part of its operator – for instance, the `units` operator which performs unit conversions would edit the output to reflect the conversions made as in Figure 7.2.

Because metadata is carried along with the data in the processing pipeline, it is relatively trivial to accomplish provide this functionality, yet very powerful. Each operator constructs a normalized, executable version of itself, and attaches itself to its output streams as a new or modified tag. In this way, the final computed product has the entire sequence of operations performed on it in a form which can be reproduced simply by executing the description.

The output of these programs can be made use of in many different ways. It may simply be streamed to consumers and never stored; alternatively, it may be materialized for future querying within the system. Choosing which subexpressions are materialized versus transient is a decision made either by the user or by the runtime; similar to the materialized-view problem within relational databases. Since many of the operations are functional and don't have side effects, the output may be materialized if space is available to accelerate future

queries, or dropped if the system is overloaded and recomputed at a later date. Furthermore, users often have good intuition about which subexpressions will be useful to precompute; for instance, many users are perfectly happy computing statistics on decimated or subsampled versions of high-frequency series to provide fast, approximate answers; it is nearly always a good idea to precompute these streams to avoid the need to load the raw data.

7.2 A DSL Processor

We have implemented the language processor described in Section 7.1 as a web service, within a software component known as the archiver. The archiver compiles user- and application-submitted queries, and executes them against stored and streaming data. The archiver evaluates queries by fetching metadata from the relational database and then loading time series data from `readingdb`. The returned time series are passed through an operator DAG generated by the query complier before the data are streamed out to the client. Because the operators are built to allow for incremental processing of input data, it is possible to process very large datasets incrementally by loading chunks of data while pushing them through the operator graph; this has the advantage of limiting the size of the working set which must be kept in memory. The archiver uses `twisted`, an asynchronous programming framework for Python to support respectable performance and concurrency; nonetheless, we believe significant performance improvements would be possible through the use of a systems programming language.

7.2.1 Compilation to SQL

The first stage in query execution is extracting a true SQL query from the submitted TSCL string which can be executed in the relational database. We have implemented two versions of this SQL compiler – the first compiles the where-clause (the metadata selection component) to a standard SQL99 query which can be executed on any relational database, while the second complies these where-clauses using PostgreSQL's `hstore` column type. Especially when `hstore` is used, the compilation is a relatively straightforward exercise; each type of operator in Table 7.1 maps directly onto an `hstore` operator or expression.

```
apply [Path="/versa_flame/inlet"] - [Path="/versa_flame/outlet"]
    to data in (now -10m, now)
        where Path like '/versa_%'
```

Figure 7.5: Example query with an algebraic expression, which benefits from extracting additional where-clause constraints from an operator expression.

The main complication involved in compiling to SQL actually results from algebraic expressions. Although it might appear from an examination of Figure 7.5 that only examining

the where-clause would be necessary to generate the SQL, support for algebraic operators is made significantly more efficient when the operator expression is also considered. Consider Figure 7.5; in this simple example, we are subtracting the inlet temperature from the outlet temperature of a water tank, producing the temperature delta as a new stream. In this example, only two time series from the device will be used in the operator expression (`/versa_flame/inlet` and `/versa_flame/outlet`.) However, there could be a large number of time series produced by the device – the where-clause restricts to only streams who's `Path` tag begins with `/versa`. If we execute the query sequentially by first evaluating the where-clause, then loading the data based on the data selector, and finally pushing that data through the operator graph, we will in this case load far too much data.

The solution, at least in this example, is to allow the operator expression to push additional constraints upwards and conjoin them with the constraints from the where-clause. In this case, for instance, the query ultimately evaluated in the metadata store should actually be something like `Path like '/versa_%' AND (Path = "/versa_flame/inlet" OR Path = "/versa_flame/outlet")`. Making this work for all operators requires additional bookkeeping due to the fact that the metadata may be altered within the operator pipeline. Therefore, we take a conservative approach to adding these additional restrictions to the SQL given that we could produce incorrect results if we apply incorrect restrictions. In our current implementation, we only apply restrictions which refer to tags which have been unmodified throughout the entire query pipeline, or have been renamed using the `rename` builtin operator.

```
SELECT s.metadata || hstore('uuid', s.uuid)
FROM stream s, subscription sub
WHERE ((s.metadata ? 'Path') AND ((s.metadata -> 'Path') LIKE '/versa_%')
    AND (((s.metadata -> 'Path') ~ '/versa_flame/outlet')
        OR ((s.metadata -> 'Path') ~ '/versa_flame/inlet')))
    AND ((sub.public )) AND sub.id = s.subscription_id
```

Figure 7.6: Compiled version of the query in Figure 7.5, using the `hstore` backend. We can clearly see the additional constraints imposed by the operator expression, as well as the security check being imposed. The query selects rows from the `stream` table, which can be used to load the raw time series from `readingdb`, as well as initialize the operator graph.

A final step in query execution is the application of security checks. Although the archiver has8 a relatively simple security model in which time series may be marked "private" so that only their owner may query data from those series, the query parser provides a natural point at which to impose additional access restrictions. Because the complier is a trusted component and has access to the request context, many different authentication systems can provide for identifying the principal, with authorization compiled directly into the query. In

```
from smap.operators import Operator

class UnitsOperator(Operator):
    """Make some unit conversions"""
    operator_name = 'units'
    operator_constructors = [(),
                             (lambda x: x, str, str)]
    required_tags = set(['uuid', 'Properties/UnitofMeasure'])
    def __init__(self, inputs):
       # inputs is a list of stream metadata
       # return a list of mutated metadata

   def process(self, data):
      # convert units as appropriate based on the input metadata
```

Figure 7.7: A part of the operator which standardizes units of

this case, we perform a join with the `subscriptions` table so as to check if the time series is associated with a public subscription and thus available to unauthenticated users.

7.2.2 Operator Evaluation

The compilation to SQL allows the query engine to resolve which time series are mentioned within a particular query, retrieving the set of stream `UUID`s for which we must load data or subscribe to streaming updates. The next step is constructing an implementation of the operator pipeline specification which can be used in evaluate the query against the data. This proceeds in two phases, and results in an executable version of the operator specification.

When evaluating the operators, the archiver first looks up the name of each operator, and binds that operator to its implementation in Python. Operators are represented as classes in Python, and the query parser is able to automatically discover available operators. Operator evaluation proceeds in two stages – in the first, the operator is instantiated and bound to the set of input streams, by creating an instance of the class implementing that operator. Figure 7.7 has a shortened implementation of our units operator. At the bind state, the `__init__` method receives the metadata of the input streams, inspects it and mutates it as necessary, and produces the output set of metadata. At this point, the dimensionality type of the operator is fixed, allowing a degree of type checking.

To allow for streaming, basic operators implement a `process` method, which is called with an array of data arguments during the second processing phase of operators. Each array element contains a matrix data structure with data from a single time series; essentially, directly implementing the model from Figure 6.1. When processing, data are pushed through the operator graph by placing them within the data structure and then executing

absolute†	add†	around†	catcol
ceil†	clip†	copy	count
datetime	dayofweek	diff	equal
ewma	exp	first	floor
greater†	greater_equal†	hist	hstack
index	interpolate	isnan†	less†
less_equal†	log†	log10†	max†
mean†	median†	meter	min†
missing	movingavg	multiply†	nanmean†
nansum†	nl	nonzero†	not_equal†
null	paste	power†	print
prod†	product	reflow	rename
rint†	set_key	snap	sqrt†
std†	strip_metadata	subsample	sum†
swindow	tgroup	trunc†	units
var†	vectorize	w	window

Table 7.2: Operators implemented in the application interface. † operators automatically imported from NumPy.

the operators. The runtime guarantees operators that processing is monotonic – data are pushed through an operator graph from oldest to newest. This significantly simplifies the implementation of operators, and with limited consequence; if data are later backloaded or revised, we must recompute any results from the beginning of time, unless we have more information about the operators.

Within the archiver, we make good use of existing libraries, in particular `numpy` and `scipy`, popular numerical and scientific computing resources for Python. These libraries provide a wide variety of functions which are efficiently implemented (often with compiled versions) and includes an excellent array implementation, arithmetic operations, windowing and resampling operators, interfaces to the BLAS and LAPACK solvers, and much more. Therefor the key challenge in implementing operator is not functionality, so much as adapting what exists to operate in a streaming context. To that end, we provide a framework for operator implementation which takes advantage of what these libraries have to offer.

7.3 Evaluation

To demonstrate how we can use operator pipelines to extract cleaned summaries of data, we explain the execution of two queries in detail, and also present a set of additional queries from real applications making use of the processing system. Within the query language, we construct an operator pipeline using the `apply` keyword.

7.3.1 Weather Data

Our first query, shown in Figure 7.8, loads data from all streams labeled as "outside air temperature" and windows it using a 15-minute period. The first argument to `window`, "`first`" is the windowing operator to be applied; this operator takes the first reading inside of the window. We could specify any compressive operator here to obtain the mean, median, or other statistic of the window. The two large integers in the data clause are the the UTC milliseconds specifying the time range of the query.

```
apply window(first, field='minute', width=15)
    to data in (1379478283262, 1379564683262) limit -1
        where Type = 'outside air temperature'
```

Figure 7.8: Our first attempt at obtaining 15-minute resampled outside air temperature data.

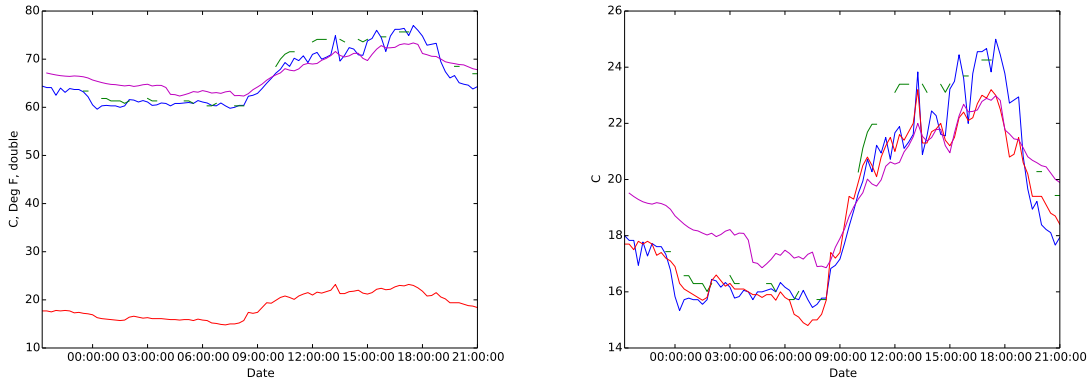
Visualizing this data is trivial using, *e.g.*, `matplotlib` as shown in Figure 7.9; here we elide some additional commands for setting up the axis labels and legend using the series metadata coming from the processing engine. Examining the output, shown in Figure 7.11a, we notice a common problem with weather data – most of the time series are in units of Farenheit, and additionally one is apparently in units of `double`. That series is actually in Fahrenheit. Figure 7.10 shows that we can quickly apply a correction to that stream, transforming the data to Celsius and correcting the metadata. The `units` operator contains a default Farenheit to Celsius conversion, but we also create a custom conversion from `double` to `C`; the resulting correct plot is shown in Figure 7.11b

```
c = TSClient()
data = c.query("""apply window(first, field='minute', width=15)
    to data in (1379478283262, 1379564683262) limit -1
        where Type = 'oat')""")

for d in data:
    pyplot.plot_date(dates.epoch2num(dat[:, 0] / 1000), dat[:, 1], '-',
                     tz=d['Properties']['Timezone'])

pyplot.show()
```

Figure 7.9: Interactively plotting time series data re-windowed using our cleaning language. Because metadata is passed through the processing pipeline, all of the streams will be plotted in the correct timezone, even though the underlying sensors were in different locations.

```
apply units( [ - 32 ] * .5555 , 'double', 'C')
        < window(first, field='minute', width=15)
    to data in (1379478283262, 1379564683262) limit -1
        where Type = 'oat'
```

Figure 7.10: Dealing with mislabeled data in inconsistent units is trivial; we can quickly convert the streams into Celsius with a correct units label using a custom units conversion expression (the first argument to units)



(a) Initial outside air data with inconsistent units

(b) Final normalized, resampled data

Figure 7.11: The application interface lets us quickly resample and normalize raw data series.

7.3.2 Building Performance Analysis

Much more complicated queries can be expressed using the application interface, expressing significant amounts of data parallelism and running over large volumes of data very concisely. As an example of this, consider Figure 7.12. In this example, we compute a few statistics about how well all of the thermal zones on the fourth floor of a building are meeting their set point. This requires several steps – for each location (represented by unique values of the "Location" tag) we first down-sample to once-a minute data; following that, we compute the room set point error in each of those minutes (the difference between the temperature and set point). Following that, we exclude any missing data, removing any times at which we were unable to compute the delta due to lack of a reading for either sensor. The `vectorize` operator runs each of the internal operators on all input data, producing an expansion in the stream dimension; in this case, it produces three output streams for each input stream corresponding to `min`, `max`, and `percentile`. Finally, we join these three streams together using `paste`; the result of this operator is a single vector for each location with the maximum, minimum, and 90^{th} percentile set point error over the day requested.

The operator graph for this query, shown in Figure 7.13 makes more of the potential

```
apply paste
    < vectorize(max, min, percentile(90))
        < missing
            < [Type = 'room temperature'] - [Type = 'room setpoint']
                < window(mean, field='minute')
to data in "4/8/2013", "4/9/2013" limit -1
where Location like 'S4-%'
group by Location
```

Figure 7.12: A complicated, data-parallel query. This query computes the minimum, maximum, and 90^{th} percentile temperature deviation across a floor over the time window (one day).

parallelism clear; the entire group-by can be executed independently for each location, and within the pipeline the vectorized operations can also occur concurrently. Furthermore the query executes efficiently even on very large datasets because all of the operators in use have efficient streaming versions – although `min`, `max`, and `percentile` need to see all of the data in order to output a result, efficient streaming versions exist in the literature.

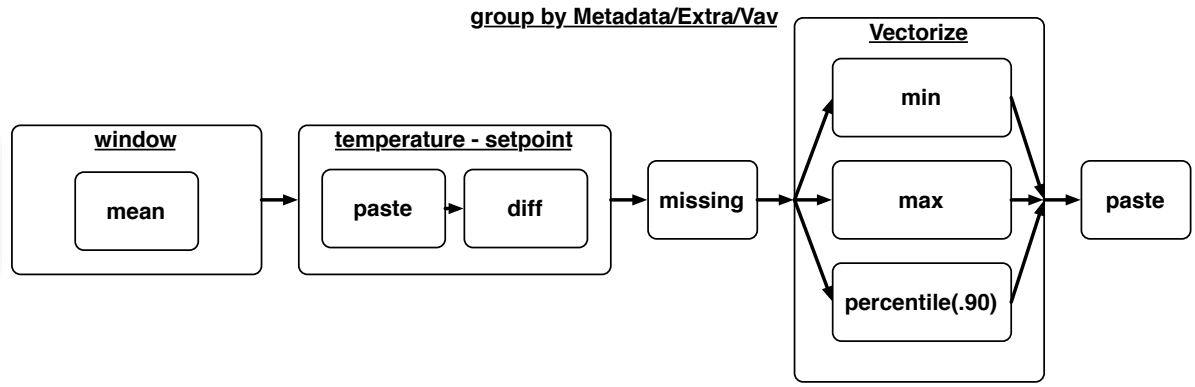


Figure 7.13: The execution pipeline for the query shown in Figure 7.12. The group-by clause exposes the fundamental parallelism of this query since the pipeline is executed once per distinct location.

7.4 Related work

Stream processing is not a new field; Stonebraker *et al.*provide a useful overview of the requirements of such as system [100]. They include:

1. "In-stream" processing with no required storage.
2. Support for a high-level "StreamSQL" language.
3. Resilience agains stream imperfections like missing and out-of-order data.
4. Guarantee of predictable and repeatable outcomes.
5. Provide uniform access to real-time and stored data.
6. Guarantee of availability and integrity despite failures.
7. Capable of distributing processing across machines and cores.
8. An optimized mechanism for high-volume, real-time applications.

The authors identify three technologies which are positioned to provide these capabilities. These are databases (DBMSs), rule engines, and stream processing engines (SPEs). According to their analysis, SPEs come closest to meeting all requirements, since the other systems (DBMSs, and rule engines *e.g.*prolog) must shoehorn the processing requirements into their processing models. When comparing these requirements with what is important for time series and monitoring data, we note that many of these are important, but note that a provenance and metadata tracking, as well as a specific processing support for the types of time series operations are two key design aspects of TSCL not mentioned at all. This oversight is actually fundamental to the distinction between event streams and time series. Stream processing engines are mostly designed with event streams in mind – a flow of discrete events which are themselves meaningful. For instance, "the stock price is now $20.47," or "the user clicked the button." These are meaningful on their own. However, time series data is often meaningful only relative to other values in the stream – is the trend up or down? We have drawn out the ways in which this distinction informs several different aspects of system design.

7.4.1 Stream Processing Engines

TelegraphCQ [19] reflects on challenges involved with building a scalable stream-processing engine over high-volume and bursty data streams. Built on top of PostgreSQL, they identify a need for graceful load shedding and shared processing between common subexpressions. The system consists of a dataflow graph where processing nodes share data using either push or pull. Queries are executed by routing tuples through non-blocking versions of standard relational operators. The system dynamically routes incoming queries through modules to respond to failure and dynamically repartition computation. They also include an ugly mechanism for defining the windows over which queries operate in a query body using an imperative syntax; the data in a window can be flushed to disk if necessary due to memory pressure. Data is pulled using special "input" modules which interface with different data sources. This builds on previous work like NiagaraCQ [21] which was a database for streaming

XML documents which explored issues with evaluating a large number of constantly changing queries.

The other streaming system in CIDR '03 published the results of the Aurora* ("Aurora-star") and Medusa systems [22]. Their vision is of multiple sites cooperatively processing streamed data to support load balancing/sharing, high-availability, and the shared protocols necessary to federate. The processing model is also that of a dataflow graph, where tuples flow between boxes, and are schedule by a schedule and query planner; they can access stored data. Aurora* is the system for distributed processing within an administrative domain, while Medusa is used for distribution across administrative boundaries. Medusa uses an economic system to exchange virtual currency in exchange for stream data in the wide area; they also construct a global namespace and registry for stream products using a DHT. They investigate sharing and load shedding at the transport and node levels. Aurora* uses pairwise load-shifting to redistributed load between nodes at the same site. The work on which Aurora* is based, Aurora [1] presents the motivation for this style of stream processing as essentially, approximate answers based on incomplete or inconsistent data based on triggers which execute on data arrival.

On the subject of windowed processing, [37] presents two forms of rollups typically computed one, landmark-based windows are computed from the previous landmark to the present point in time; for instance, daily averages. Sliding windows continually advance the start marker and are of a fixed with; for instance, hourly averages. These two forms of rollups have been the study of considerable work and are the basis of the Aurora* intuition that it is the windowing of streaming data that makes it challenging to compute queries without large buffers.

The IBM System S is a large-scale effort to develop a distributed, multi-site stream-processing engine which processes many times of time series data [29]. Example queries include geographical ones like "find all bottled water in the disaster area", and they mention as goals the integration of video data. Like other stream processing frameworks, they envision extensive cross-site collaboration; in their system, it is mediated through Common Information Policies (CIPs) which define the types of data a particular site will share with other partners, and what actions they will take on behalf of other sites. Processing may encompass historical data, derivatives and to operate under continuous overload, shedding lower-priority queries.

7.4.2 Bulk Processing

There has been significant work at enabling massively scalable processing of distributed data sets, particularly using the map-reduce paradigm exemplified by its popular open-source implementation Hadoop [27]. These large systems consist of a storage system providing a file-like abstraction and a job execution service. Jobs consist of *mappers*, which read input key-value pairs and maps them to new pairs, and a *reducers* which combine output keys with the same value. Jobs are frequently pipelined into multiple stages to compute complicated

functions of the input data. The model has been widely adopted, perhaps because it allows massive parallelism without need for explicit locking or synchronization.

Several attempts have been made to layer higher-level languages on top of the map-reduce paradigm to improve analyst productivity. Pig [84] compiles an imperative, data-flow style language into a series of map-reduce jobs which run on Apache Hadoop [96]. Later work optimizes pig programs through the sharing both of concurrent executions of overlapping computation, as well as caching of popular intermediate results [83]. Sazwall is a similar effort [91]; Spark reuses the storage infrastructure underlying Hadoop, but keeps data in memory to reduce execution times by as much as an order of magnitude [116]. Tenzing [20] is a recent effort to develop a compliant SQL92 query engine on top of the Google MapReduce implementation. They overcome real and perceived problems such as slow execution time and a lack of applicability of traditional database techniques and report that the base query execution time is about 10s. Tenzing is in some sense a database-of-databases, since it can compile a query to run against files, traditional relational databases, stored protocol buffers, and several other internal stores.

More generalized data-flow processing systems such as Dryad build a data processing framework with more expressive potential than map-reduce [50]. Programmers have control over the structure of the graph, which is then scheduled onto physical hardware by the Dryad runtime. They apply various optimizations to the data-flow graph, and implement pipelining between operators. DryadLINQ is their high-level query language which compiles LINQ (Language INtegrated Query) programs into a Dryad call graph [114].

The TSCL is very much in the tradition of domain-specific languages for specific processing tasks. Mapping the operators as defined here onto several of these processing frameworks is an interesting exercise, since the high-level, declarative nature of TSCL would provide significant ability to parallelize the underlying computation which we currently do not exploit. However, many operations are "embarrassingly parallel" and would be greatly accelerated using standard techniques.

Adding incremental processing

A problem with bulk processing systems is that processing incremental updates can require the rebuilding of the entire computation, a potentially very expensive operation. The Percolator system [86] allows users to incrementally update the results of a previous batch (map-reduce) computation through the use of *notifications* and *transactions*. First, the system allows chains triggers to be installed which cause custom handlers to run when a piece of data in the underlying repository is changed. These handlers can atomically update the state of the repository through the use of a distributed locking service which can provide ACID semantics to provide snapshot isolation.

Another attempt to avoid needing to wait for batch jobs to complete to retrieve current results is the MapReduce Online system [23]. MRO preserves the structure of a map-reduce job within Apache Hadoop, but pipelines data from map tasks to reduce tasks; it is customarily committed to disk before running the reduce stage for fault tolerance; as a result the

system can produce partial results as jobs approach completion. Using this ability, application writers can build systems which allow analysts to iteratively drill down, following the CONTROL methodology of allow iterative, interactive data exploration [45]. The prototype also allows for continuous queries, where the mappers and reducers run forever. The reducers are periodically run on data accumulated from the mappers – the period can be determined from wall clock time, or features of the input data.

The TSCL runtime currently only supports running queries on-demand, which requires loading the source data. However, much of the computations are predictable – for instance, querying windowed subsamples of data which could be maintained incrementally as new data arrives.

7.5 Takeaways for Physical Data Processing

Time series processing in monitoring and control applications is not the same as relational data processing, nor is it the same as streaming event processing although it has important overlaps with both of these models. It differs in how the data should be stored, access, and the kinds of operations users wish to perform on it. To date, it has often been treated as an additional application of these tools, or argued that it varies in only minor ways. In developing the workloads, use cases, and ultimately a domain specific language embodying these uses. In this design, we can clearly draw parallels to previous work and begin to see how one could implement a time series-specific framework while leveraging parallelization to enable additional scale. Our prototype implementation demonstrates how a concise language can significantly shorten application code by dealing with the first few processing stages before the application ever sees the data; essentially, it is the interface through which applications can access the data. This has the benefit of simplifying application code, and exposing significant potential down the road for optimization of the underlying computation.

Chapter 8

Controllers

Given a distributed set of actuators such as those presented by the sMAP drivers in the hardware presentation layer, the final stage of the monitor – model – mitigate paradigm requires us to put in place new controllers and sequences which alter the operation of existing physical resources to improve efficiency or alter operations. Because we assume that the control logic which is assuming control of these resources may be located in a different fault domain from the controllers themselves, subject to network partition or poor availability, we must redesign the interface between the control logic and controllers themselves to provide well defined semantics to the application. In designing this interface, we aim to provide an interface which is robust to non-malicious applications operating in an environment subject to a variety of common errors; in conclusion, we do discuss some initial approaches for extending this approach to untrusted applications.

Unlike the hardware presentation layer, which exposes a relatively simple, predominantly stateless interface upon which more complicated applications can be build without the pain of extensive system integration, the control tier aims to begin to provide a more full-featured runtime in which application may run. At the control tier, we which to enable simple, straight-line code which runs with well-defined failure semantics. We are particularly interested protecting building systems from misbehaving applications – applications which crash, become partitioned, or experience losses in resources or access to data which renders them unable to proceed. We are explicitly *n*ot interested in protecting the system from actively malicious applications; except in relatively simple examples, protecting from malicious applications will be very difficult.

8.1 Control Architecture

The overriding metaphor we use to inform the design of our controllers is the transaction. Although imperfect, it is a useful comparison; in databases, transactions provide a way of encapsulating a piece of logic within a structure which provides well-defined semantics around the behavior of the logic and its effect on data in the presence of failure and concurrency.

We are interested in a similar idea for control systems – small pieces of logic which run with a certain amount of isolation and fault tolerance from other code in the system.

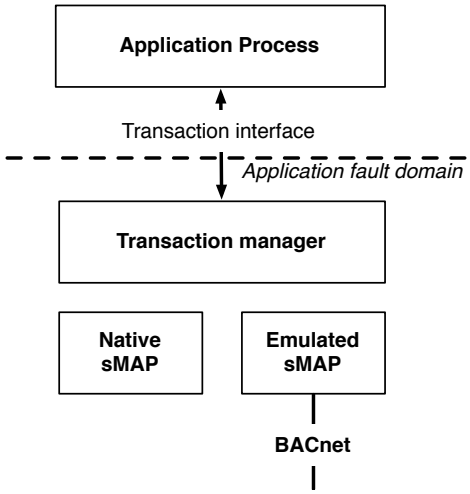


Figure 8.1: A high-level view of transactional control. Control processes, containing application-specific logic interact with sensors and actuators through a transaction manager, which is responsible for implementing their commands using the underlying actuators. The interface presented to control processes allows relatively fine-grained control of prioritization and scheduling.

Control processes containing the program logic connect to a centralized transaction manager as shown in Figure 8.1, as well as other services such as the time series service and web services, as required. The transaction manager is responsible for implementing the actions requested by the process, as well as undoing them or mediating between different demands from other processes. In the future, it may be possible to distribute this management logic or move it into a library within the control process. The transaction manager is a small piece of code which can reliably take actions as requested by control processes, and undo them upon various failures; because network partition may occur between the transaction manager and the control process. The major design goals of transaction manager are:

Network partition: control processes may become partitioned from the underlying sMAP servers; the transaction manager provides predictable semantics in the case of partition so that the control processes do not need to be physically collocated with the actuators, as is common practice today.

Coordinated control: a common pattern for application writers is to affect synchronized changes across a building. If any of them fail or would be masked by higher-priority writes, none of the other actions should occur; in some sense, all the writes should become visible at once.

Application prioritization: mediating conflicting application requests is challenging; the transaction manager provides primitives for ensuring relative priorities between different applications.

8.1.1 Failure Models

Introducing control external to this picture changes the failure modes the composite system may experience, because the shared bus and outbound connection now are now placed in the critical path of operations which heretofore they were only needed for supervisory purposes, and for which a lower level of availability could be appreciated. Once the control loops are extended, the system may now experience partition between the new controller and the low-level actuators being controlled. The main risk which this introduces is that the low-level controllers may be left in an inconstant state; most controllers which are elevated by the HPL into sMAP drivers have simple "last write wins" semantics, so a failure or partition will leave the output or set point holding the last value which was written by the application.

Because the applications we are introducing override the default control logic implemented by the original controllers, our overriding assumption is that if we release our control of the system, the lower-level controllers will revert to a sensible default; for instance, for a room temperature controller, if we stop providing control inputs with new set points, the room will revert to whatever set point was preprogramed.

8.1.2 Coordinated Control

In observing a large set of applications being built to control buildings, we observed two distinct classes of applications being built; we document these here and return to them after introducing the detailed transaction design. These three classes we introduce briefly are archetypes; in actuality, applications will employ more than one of these patterns.

Direct and Supervisory Control

Direct, and to some extent, supervisory control applications are in some sense the least radical modifications of existing systems, and are made up of applications which attempt to operate the system in a new and better way, but fundamentally within the same regime in which it currently operates. For instance, an improved VAV controller might attempt to emulate a newer, dual-maximum control sequence on top of an existing controller with only simpler single-max or other less-efficient control sequence; or trim-and-respond might be implemented as a chilled water reset strategy on an older building with simple setpoint-based water loop control strategy. Both of these are examples of bringing modern best practices to older systems which had less sophisticated control logic.

Although these simple examples are illustrative, more complicated applications are also possible; for instance, whole-building model-predictive control, which attempts to manipulate nearly all of the control loops in the building in a coordinated fashion so as to achieve a fully

optimal mode of operation, instead of the simpler, decoupled control loops which are typically present.

External Responsive

Another significant class of applications consists of logic which is often event-triggered, where the trigger is initiated by an external event; or at least by an external control signal. One example of this type of application is *demand response*, where a utility generated signal requires a building to take an action such as relaxing set-points, reducing lighting levels, or other load-shed action so as to reduce their electricity consumption during periods of peak load. Another, *occupant responsive*, results from interaction with building occupants – they make a request via smart phone, web page, or other Internet-connected device; in response to the request, the building responds in some way, perhaps adjusting the lights, HVAC settings or other functionality. There are myriad potential applications in this category, and building become part of much larger loops, and many of them require making multiple coordinated adjustments to the building's operation.

8.2 Control Transaction Design

The HPL provides the level of access typically available in supervisory control systems: the ability to read and write individual points. However, these actions are typically taken as part of a larger sequence of actions which occur concurrently with other actions, and require cleanup following a failure. Control transactions provide the primitive for constructing reliable applications. A transaction is composed of several parts:

1. A set of **actions**, which are the underlying changes to affect; for instance, to read or write a set of points and when to do so,
2. a **transaction id**, uniquely identifying the transaction;
3. a **callback**, triggered when the action is complete (or fails),
4. a **lifetime**, determining when the transaction will be reverted,
5. a **priority level**, which specifies how important the transaction is within the system and is used for scheduling between multiple concurrent transactions,
6. an **error policy**, determining what happens if the actions fails, and
7. a **reversion method** specifying how to roll back the action.

Transactions are submitted by control processes to a transaction manager running in the same fault domain as the HPL; the transaction manager is responsible for scheduling actions, sending callbacks, and reverting actions when necessary. Actions taken by a transaction are

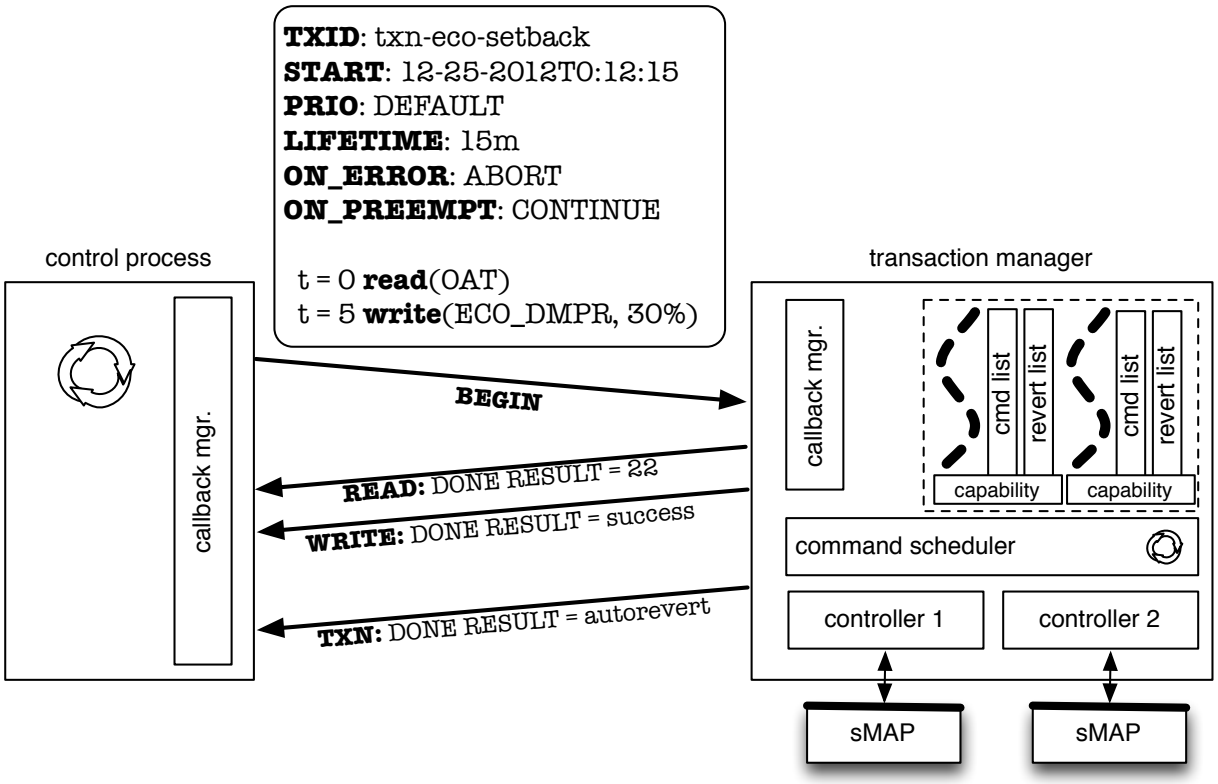


Figure 8.2: A transaction manager (TM) communicating with a remote Control Process. The TM prioritizes requests from multiple processes, alerting the appropriate control process when necessary. The TM also manages the concurrency of the underlying physical resources, and provides all-or-nothing semantics for distributed actions. It provides the ability to roll back a running transaction.

always transient; when the lifetime expires, all actions taken by the transaction will be reverted. Running transactions may be renewed to lengthen their lifetime. As a result, in the absence of control input from applications the state of the control system will revert to the hard-coded, least-common-denominator control strategy; *i.e.*, how the building would operate in the absence of a sophisticated application infrastructure. Transactions also provide a mechanism to ensure "all or nothing" semantics when applying actions affecting a number of points. The error policy specifies whether or not a transaction should continue if one or more of its component actions fails; failures can occur for a number of reasons.

A transaction proceeds through three states during is lifetime, as it is prepared, executed, and finally reverted.

8.2.1 Prepare

The *prepare* phase of the transaction involves only the client (the control process), and consists of constructing a static representation of actions to be taken when the transaction is executed. This may include querying other services such as the time series service to obtain information about the building and the names of control points to be manipulated; scheduling actions for when the transaction runs, for instance a periodic reads of a certain point, and attaching callbacks to the various schedule actions. The client will also configure other details of the transaction such as the default lifetime (after which all actions will be reverted), the default error policy (what to do if any actions fail), and the default reversion policy (any special steps that are needed when undoing actions).

As a result of this process, the client constructs a *prepared transaction* containing a schedule of actions which are to be taken. Because this part of the schedule is constructed without the involvement of the transaction manger, it is possible to execute a prepared transaction in a "fire and forget" mode – the client can submit the prepared transaction to the manager and then disconnect, leaving the manager to run and finally revert the actions without further client interaction. Once the transaction is prepared, it can be run by submitting it to the transaction manager.

8.2.2 Running

When run, the transaction is sent to the transaction manger, which adds it to the list of runnable transactions. All actions in the prepared transaction are added to the global transaction schedule, to be run at the appropriate time; the client receives a handle to the running transaction instance that allows them to return later to alter actions, add new ones, or abort the transaction before its lifetime expires. Once actions are runable, actual execution is passed off to controller components which perform the action by communicating with the appropriate sMAP devices and proxies. While running, the transaction manager is responsible for executing the actions and dispatching callbacks to the client with the results; for instance, if the action schedule contains reads of various points, the callback will include the value that was read. The manager allows clients to mutate the transaction schedule at this point by inserting or deleting actions; for instance, applications which have data dependencies can use this functionality to implement feedback control loops by periodically reading the loop inputs and then scheduling new actions with the output result.

8.2.3 Reversion

When a transaction's lifetime expires, it is canceled, or it encounters an error that requires aborting, the transaction enters the reversion stage, during which the transaction enqueues new commands to undo the previous control inputs. The goal of the reversion stage is to release whatever control the application was exerting over the system. This generally means "undoing" any writes to actuators, and stopping access to any sensors which were being

used for reading. Depending on the underlying control system, it may be possible to simply "clear" the write; for instance, when writing to a BACnet point with a priority array such as those shown in Figure 2.5. With such systems, a clear has the advantage of being idempotent and stateless. Simpler systems such as an actuator without a priority array may require a "writeback" strategy for reversion, where the transaction manager tracks the actuator value before a write, and the writes that value back when the transaction reverts.

Various problems may be encountered when reverting a transaction – the command to undo the previous write may fail, or the system may be in a new state which makes it undesirable to immediately jump back to previous values.

8.3 Implementation

We have implemented the transaction design as a service within a sMAP driver, available in Python. We made several implementation decisions when doing so that vary from approaches we have taken elsewhere – in particular, we do not provide an HTTP-based API for control processes, but instead use a more traditional asynchronous remote procedure-call style. The main reasoning for this particular choice is that the client-server model of HTTP does not naturally lend itself towards client callbacks, while the `twisted.spread` package allows us to easily marshal and sent references to callbacks to the transaction manager that appear as idiomatic Python, yet run remotely. The use of this package is not fundamental to our design, but does result in significantly cleaner code than would be otherwise possible.

8.3.1 Example Transaction

To illustrate how a transaction proceeds through the three states, we will utilize one of our example applications: the *user responsive* application. In this application, a user request on a web page generates an immediate response from the HVAC system which immediately heats or cools their space. In our test building, Sutardja Dai Hall, the response is generated by commanding the heating coil in the VAV to either open or close, and then opening the damper to allow more air than is normally provided to to the room.

The source for a simplified version of this transaction, shown in Figure 8.3, illustrates many salient features of transactions – the code is short, concise, and portable, since it relies on the metadata stored in the time series service to identify the points needed for control, rather than being hard-coded. The result is a prepared transaction, which includes a schedule of actions to be take, shown in Table 8.1.

Figure 8.4 shows a slightly modified version of this transaction is executed on a building. In this figure we observe both the airflow set point and the actual measured airflow, as the zone runs through the transaction schedule. The simple example demonstrates all three key aspects of transaction design. First, the control of the airflow and temperature are *coordinated*; because the transaction was initialized with `error_cancel=True`, if any of the write actions fail the entire transaction will be reverted. Secondly, the transaction will run at

```
class Blast(ControlApp):
    def success(self, result):
        print "Transaction started"

    # vav (str): the name of the vav unit
    def cool(self, vav):
        txn = RemoteTransaction(timeout=timedelta(minutes=10)),
                                error_cancel=True,
                                cmd_priority=BLAST_COMMAND_PRIORITY,
                                priority=BLAST_PRIORITY)
        # get airflow control point names
        airflow = self.client.query("select distinct uuid where "
                            "Metadata/Extra/Type = 'airflow setpoint' and "
                            "Metadata/Extra/Vav = '%s'" % vav)
        # and setpoints
        valve = self.client.query("select distinct uuid where "
                              "Metadata/Extra/Type = 'valve command' and "
                              "Metadata/Extra/Vav = '%s'" % vav)

        # open the airflow damper
        d = txn.add_write(0, airflow, 750)
        txn.add_write(120, airflow, 600)
        # close the heating valve
        txn.add_write(0, valve, 0)

        # note that we succeeded
        d.addCallback(self.success)

        # set the right txid
        txn.txid = 'blast-' + vav

        return txn
```

Figure 8.3: Example code setting up a "cool blast" of air in a building. The code first finds the needed control points using TSCL metadata query; it then prepares a transaction which consists of fully opening the damper, and closing the heating valve to deliver a cool stream. It can note when the transaction has started by attaching a callback to the first write.

a high priority level relative to other transactions, based on the `priority` and `cmd_priority` levels; we discuss some of the implications of these on the behavior of write actions in Section 8.3.3. Finally, the revision strategy (which runs at the expiration time, 10:38) ensures that the blast will complete successfully even if the control process crashes or is partitioned.

8.3.2 Reversion Strategies

Once actions are scheduled, actual execution is passed off to controller components that perform the action by communicating with the appropriate sMAP devices and proxies. As controllers take actions, they also append to a per-transaction **revert log**: a list of actions or pieces of logic that are needed to undo the control inputs which are being made. When the lifetime of a transaction expires, it is canceled, or it encounters an error, the revert method

time	point	value	reason
0	heating valve command	0	close valve to make sure we cool the room
0	airflow set point	750	fully open the damper to deliver cooled air
2 minutes	damper command	600	reduce the airflow somewhat
10 minutes	all points	*n/a*	transaction reverts due to timeout; writes cleared

Table 8.1: The resulting action schedule contained in the prepared blast transaction created in Figure 8.3.

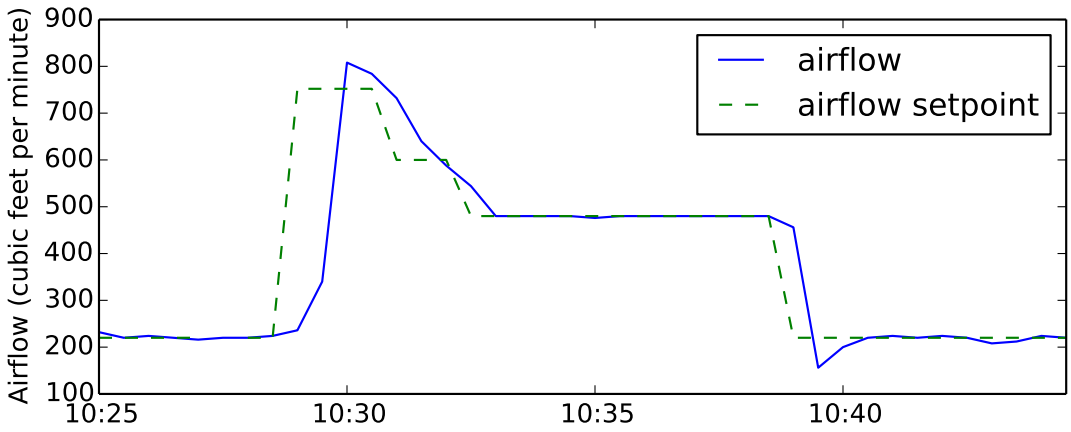
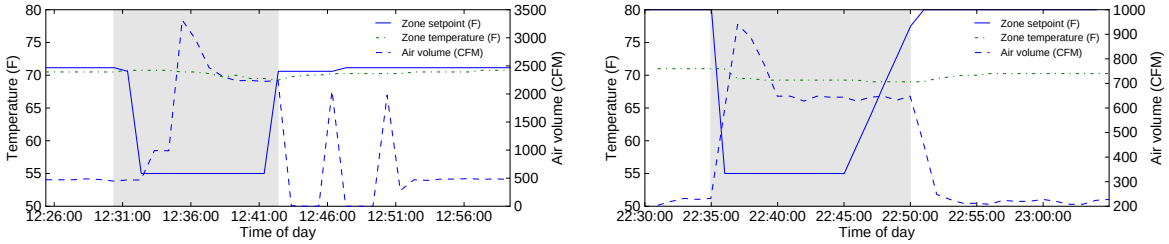


Figure 8.4: The result of running a real command sequence in Sutardja Dai Hall; the real sequence has more steps than the simpler sequence used as an example. We clearly see the airflow in the system in response to our control input.

is used to enqueue new commands to undo the previous control inputs.

The naïve reversion policy would simply clear any writes made; however, Figure 8.5a illustrates one problem with this method. Here, the setpoint is reduced at around 12:31, causing air volume to increase and room temperature to fall. However, when this change is reverted at 12:41, the default commercial controller which takes over becomes confused by the unexpected deviation from setpoint, causing the damper position (and thus air volume) to oscillate several times before finally stabilizing. Understanding and dealing with this issue is properly the concern of a higher-level component such as a VAV driver; to allow this, some drivers provide a custom revert action along with their inputs. These actions consist of restricted control sequences requiring no communication, replacing the default reversion policy. In Figure 8.5b, the VAV driver uses a custom revert sequence to gradually release control back towards a known steady-state position.



(a) Unstable behavior around a transition

(b) Specialized reversion sequences can deal with this problem

Figure 8.5: Drivers may implement specialized reversion sequences to preserve system stability when changing control regimes.

8.3.3 Multi-transaction Behavior

Because a design goal is to have well-defined behavior in the case of multiple applications, our transaction design contains several features to make it easier to reason about system behavior in the case of multiple writes. The basic strategy in use is simple prioritization, which appears in two ways: *execution prioritization* and *command prioritization*. We augment these simple priorities with notifications, so that applications can receive timely information about the state of the actions they have taken.

The first strategy, **execution prioritization**, determines the ordering between actions from different transactions when they are runnable at the same time when an underlying resource must be acquired in order to complete the action. For instance, suppose two transaction both contain actions writing a particular point on controller accessed using a shared serial bus. Because executing the two actions requires bus access, they cannot occur concurrently; the transaction priority level (in the example, determined by the `cmd_priority=` argument) will determine which action executes first. This priority level is analogous to the I/O prioritization present in many modern operating systems; for instance the Linux I/O scheduler contains extensive functionality for merging operations and enforcing quality of service. Due to our application domain, we implement strict prioritization between priority levels, with a FIFO policy within each level. This provides predictable behavior, with well-known drawbacks.

The second strategy, **command prioritization**, refers to how different transactions which access the same point interact in the presence of multiple writers. The building blocks of this strategy are priority levels, notifications, and locking. Each write action is associated with a *priority level*, relative to all other write actions to that point; in the example, the `priority=` argument applies to all writes within that transaction. The write with the highest priority level "wins," and the point takes on that value. However, writers with both high and low priorities may wish to receive guarantees about their writes when other transactions are present. For instance, a low priority writer might wish to be sure that their writes take effect and are not masked, whereas a high priority writer might want to prevent any writes

by lower priority. Therefore, write actions may specify a locking precedence using the `xabove` and `xbelow` flags.

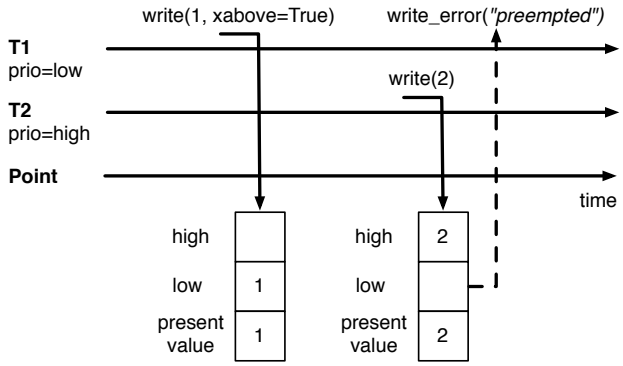


Figure 8.6: Illustration of preemption. In this example, two transactions, one with low priority and another with high priority submit write actions to the same point. The low-priority action occurs first, and is made with the `xabove` flag. When the second write by T2 occurs, the point takes on the new value because T2's write action has higher priority. Additionally, T1 receives a notification that its write action has been preempted and cleared. Depending upon T1's error policy, this may result in aborting that transaction, or executing a special handler.

When a write is made with `xabove=True`, the write is said to be "exclusive above" – that is, any higher-priority writes should result in an error. Because the transaction itself will be lower priority than the transaction submitting those higher priority writes, the runtime will abort the submitting transaction if a higher priority write is made to that point any time that the original write is submitted. This process is illustrated in the timeline in Figure 8.6. Analogously, making a write with `xbelow`, or exclusive below, indicates that no lower priority transactions should be permitted to make writes; although normally those writes would be masked by the higher-priority write, this functionally along with a null write is useful as a way of "locking" a point – preventing lower priority transactions from accessing it.

8.4 Related Work

Although we have not seen a similar transactional approach to changing control state before, there are a number of approaches which have been investigated either industrially or academically which address some of the same concerns as those we are interested in. These fall, to some extent, into categories of *controller programming*, *robust control* and *temporal logic*. Much work known as "control theory" is actually quite different, although complementary, to our goals in this section. Control theory, the science of building transfer functions from

inputs to outputs so as to maintain some overarching control goal is not as much concerned with how those transfer functions are actually implemented.

Controller programming frameworks are widely used for programming programmable logic controllers (PLC) in the manufacturing, process control, and building industries. PLC controllers are hardware devices with a range of digital and analog input and output points, and can be programmed in many different ways. Historically, these controllers were designed as digital replacement for relay panels programmed in ladder logic – a sort of boolean programming logic. More modern PLC's support programming in graphical logical-flow languages, or in versions of basic. Those programming paradigms have been standardized in IEC 61131 [49], although the major building controls systems vendors implement proprietary variants of these ideas. To illustrate how PLC's are programmed in practice, we have included a small portion of a PPCL program [97] (the native language of that control system) in Figure 8.7 from Sutardja Dai Hall. This particular system uses a BASIC-like structure, with control flow determined primarily by line number. The controllers interpret this code by sequentially evaluating lines from the program once programmed. Other controllers are programming using a graphical logical-flow language similar to LabView [79]; in such a framework, the programmer wires together input and output lines from pre-defined function blocks; for instance an airflow controller. The runtime system then synthesizes from this graphical program a firmware image for each controller in the system. These systems have the advantage of being (somewhat) intuitive, but fall short in several ways. Most of them do not encourage even use of basic software engineering principles like structured programming, much less the device abstraction that would encourage portability between systems. Their assumptions about the underlying control fabric also encourage the view that the network elements connecting different components is reliable which, while potentially reasonable within a building or factory, will cease to hold once control loops are extended to the wide area.

Within these systems, actual control logic is often encapsulated within blocks of precompiled code, which can be accessed by the interpreter or appear graphically as a visible block. The workhorse of control theorists for actually implementing these transfer functions on real hardware are tools like Matlab Simulink [106]. Once a transfer function has been mathematically constructed, these tools provided sophisticated code synthesis tools for generating a program implementing the transfer function, which can then be compiled and burned directly onto a microcontroller. These tools provide an important bridge between high-level control theoretic language and the details of implementing such logic in hardware.

There have been a number of recent academic efforts to improve the controls programing situation, which provide interesting design points and new capabilities. One key thread of work has been an effort to integrate model-based control into legacy systems, in an effort to increase building performance. The MLE+ toolkit [12] integrates building and environmental energy modeling from EnergyPlus [25] with a Matlab-based control framework which allows users to construct building control kernels using Matlab. Another system called BuildingDepot [2, 111] imposes well-defined patterns onto common building systems, so as to simplify writing code which mutates the buildings state, although does not seem to deal with the low-level semantics of control in the face of failure. A popular system for homes, HomeOS

```
00030       C            ######################################################
00040       C            ###                 UCB DAVIS HALL                 ###
00050       C            ###                    BERKELEY, CA                  ###
00060       C            ###                        MBC03                     ###
00070       C            ###          CHILLED WATER SYSTEM PPCL               ###
00080       C            ###                                                  ###
00090       C            ######################################################
00100       C
00110       C
00120       C        $LOC1  = BOTH CHILLERS RUNNING EVAPORATOR BUNDLE DELTA T LOAD CALCULATION
00130       C        $LOC2  = CONDENSER WATER PUMP 1/2 PROOF OF RUNNING TRIGGER
00380       C ********************************************************************************
00390       C
00400       C *** POWER FAILURE AND DEFINE STATEMENT CONTROL ***
00410       C
01000       ONPWRT(1020)
01010       GOTO 1040
01020       SET(0.0,SECNDS)
01030       LOCAL(LOC16)
01040       DEFINE(A,"SDH.CHW1.")
01045       $LOC16 = "SDH.CH1.CHW.FLOW"
01050       IF("SDH.CH1.CHW.FLOW" .OR. "SDH.CH2.CHW.FLOW")
                      THEN ON("SDH.CHX.CHW.FLOW") ELSE OFF("SDH.CHX.CHW.FLOW")
01052       "SDH.CHW_BYPASS_VLV_LOOPOUT" = $LOC13
01270       C *** CHILLER SEASONAL SEQUENCE CHANGE CONTROL ***
01280       IF(MONTH .GE. 4.0 .AND. MONTH .LE. 9.0)
                      THEN ON("%A%CH_SEASON") ELSE OFF("%A%CH_SEASON")
01290       IF(("%A%CH_SEASON" .EQ. ON .OR. "%A%CH2_FAIL" .EQ. ON) .AND. "%A%CH1_FAIL" .EQ. OFF)
                      THEN "%A%CH_SEQ" = 12.0
01300       IF(("%A%CH_SEASON" .EQ. OFF .OR. "%A%CH1_FAIL" .EQ. ON) .AND. "%A%CH2_FAIL" .EQ. OFF)
                      THEN "%A%CH_SEQ" = 21.0
```

Figure 8.7: A small portion of the programming of chiller sequences in Sutardja Dai Hall on UC Berkeley's campus. This portion of the control program is responsible for adjusting the chiller sequence in order to respond to seasonal changes.

[28] is an interesting, if monolithic approach to programming a distributed set of resources with in a home. It includes a language based on Datalog for controlling sharing (including a temporal clauses), which might become even more compelling in the large environment of a commercial building. The ubiquitous computing community has also proposed frameworks for overlay control on top of distributed physical resources such as ICrafter [92]; conceptually similar to our design point, this work represented control as a set of services, atop which the challenge was building a unified user interface.

8.5 Takeaways

The control transaction metaphor allows for creating applications which, while trusted to be non-malicious, may also not be perfectly behaved. By packaging all of the changes made on the building by an application into a single package, administrators can easily remove a

misbehaving application by manually terminating it. The abstraction also provides for good behavior in the case where the application crashes or becomes partitioned from the building it is controlling – the transaction manager will simply revert the actions it has taken, allowing the building to return to it's default operating regime. By keeping lease times short, we can ensure that the building does not move too far in the wrong direction in the case of failures.

Chapter 9

Conclusions

We began this thesis by posing the problem of building a system for building control and optimization that would admit a broader range of applications than is normally considered under the "control systems" banner. Roughly speaking, we divided the solution space into thirds – device access, data storage, and control semantics; our approach to building a system which solved our overarching problem was to investigate requirements in each of these thirds and then implement to gain real-world experience with each of these components.

9.1 Contributions

The major contribution of this thesis is to design and evaluate an architecture for implementation of overlay control onto existing systems – that is, systems which begin with traditional control systems as an underlay, and builds atop them new functionality which incorporates these systems into new, larger control systems. Because the types of applications one wishes to build in such a system are often integrative, in the sense that they coordinate control over a variety of underlying systems, we first addressed consistently exposing sensing and actuation from a heterogeneous underlying systems in a principled and efficient way. Although certainly not the first to address the problem of collecting telemetry, our solution, sMAP is based on significant, deep real-world experience in collecting data in a wide variety of settings, and provides a compact, simple solution which meets a diverse set of requirements not met by other protocols.

When we first began collecting large volumes of time series data and encountering the limitations of relational databases, we looked for the "MySQL for time series" – the easy-to-install package that would satisfy 90% of use cases without a fuss. It now seems that such a package does not exist, at least in the free software world; commercial database support is somewhat better. Beyond this rather glaring hole, however, it seemed difficult to express the common operations one wishes to perform on time series using SQL. Looking at what is needed led us to our initial solution to the problem, a system for naming, storing, and processing time series which overlaps with the relational model with special treatment for

time series, enabling concise, declarative specifications of data cleaning operations so as to represent distillates.

Finally, much experience in changing buildings' operations using overlay control on top of their existing control systems, especially using the BACnet protocol, let us to a better appreciation for those protocols and also their deficiencies. In particular, the challenge of reliably *undoing* changes which have been put in place by a higher-level control led us to the design of the control transaction as a package of logic and timing, with clearly defined behavior in the case of a variety of failure conditions.

We have found that the package of these three elements is more powerful than than the state of the art in control systems for a few reasons. For one, extensive integration effort is often needed just to expose relatively simplistic functionally to applications; this is a direct result of the monolithic, siloed architecture of many legacy systems. Breaking this apart and making the underlying functionally available as a service natural admits a more factored implementation and greater flexibility. Secondly, although many legacy systems contain a historian, it is often separate and used only for offline analysis. By building a time series system which supports real-time analysis and with sufficient performance to be place inline with certain control decisions, controllers are able to make use of historical data.

9.2 Broader Impacts

The work described here, while still young, has already had a certain amount of impact beyond our research group. Both sMAP and the time series service (which in our software distribution are packaged together) have seen adoption by several different communities – evidence, we believe, that our design has been on the right track. Building scientists and architects at the Center for the Built Environment have adapted sMAP as a platform for building data analysis, for both energy and comfort analyst. Scientists at the Lawrence Berkeley National Lab have built several interesting, new tools around it, and have used it for collecting data about residential water consumption and lighting usage. Researchers at the Pacific Northwest National Laboratory have adopted sMAP as the data integration and historian for their Voltron platform [64], a system for agent-based electric grid control. Danish researchers have used it as a platform for collecting telemetry from a variety of smart-grid connected appliances. Last but not least, a number of companies have used components of sMAP to build new products and services, including cloud-connected refrigerators, personal comfort systems, and energy analyses. We have also enabled a significant amount of research at Berkeley by exposing data and control which was heretofore difficult to access. Researchers have developed numerous applications on top of our platform, including demand controlled ventilation [104], model-predictive control for HVAC systems [7], demand response, as well as supporting a variety of undergraduate projects and hack-a-thons.

9.3 Future Work

There is of course much work to be done. Some of the work revolves around continuing extend the platform to meet address important considerations which we were not able to tackle in this initial work. Other work involves moving some of the key findings of our work from an academic setting into the wider world of industry and standards bodies. Finally, we believe some of our results are applicable to other neighboring domains, and that both would be well-served by closer collaboration.

Within the platform architecture, we presented three key pieces which are minimally sufficient for implementing interesting new services. However, we provided a relatively cursory treatment some important issues in this work: most importantly security, access control, safety, and abstraction. Specifically, we did not address the challenge of allowing *less-than-trustworthy applications* in this context; applications within BOSS are assumed to "know what they're doing." We believe that BOSS provides a good framework for a through exploration of issues; potentially using a declarative approach for expressing constraints, or a model-driven approach for identifying safe regimes of system operation and then taking action to prevent applications from entering unsafe regimes. Another area which requires elaboration is the issue of *driver composition* – our control framework allows single applications to make changes to building operations, but it is not entirely clear how to compose those applications into (for instance) a hierarchal structure in order to allow additional factoring. Finally, the time series service provides a useful amount of metadata about underlying control elements, but it is not appropriate for certain types of queries. For instance, it is difficult to encode information about relationships relating different domains – spatial, system, electrical, *etc.* The Building Application Stack [63] work provides a starting point for several of these issues; properly integrated into BOSS, it could be a compelling solution.

While designing and implementing our concept of transactions on top of existing controllers, we constantly encountered problems with reliably implementing reversion sequences and notifications, especially on top of BACnet. Although redesigning BACnet is probably too ambitious, we feel strongly that a few small modifications to the protocol would significantly simplify the task of implementing overlay control on top of it. First, allowing writes into the BACnet priority to array to be associated with a *write lease time* would allow stronger guarantees that actions taken will actually revert; at the present, a transaction manager crash or partition from the controllers may result in orphaned writes unless the implementor is extremely carefully and provides both write-ahead logging and log shipping of actions so that they can be reliably undone. Secondly, providing for *notifications to writers* when their writes are masked or unmasked by a higher-priority writer would eliminate the priority-array polling now required to properly implement notifications in case of being overridden.

Finally, there is significant overlap between the time series processing work we have accomplished and the Internet community's work on streaming event processing. Reading the literature (for instance Aurora, and more recently Spark Streaming [1, 117]) suggests much overlap with some differences; Internet systems tend to be much more focused on extracting meaning from discrete events, while time series rely more on batches of records in order to

look for trends. Nonetheless, taking some of our insights about time series and expressing them within a framework designed for massive parallelism and high data rates would be extremely interesting and could help bridge the gap between two separate communities; we believe we have much to offer regarding the semantics of time series data processing that would also be applicable to processing telemetry from server farms, clickstreams, and many other common Internet workloads.

9.4 Final Remarks

One underlying tension we feel constantly when designing overlay control systems is the tension between between centralized control and distributed decision making. This debate is felt broadly across many communities; generally centralize systems are simpler, more straightforward to reason about, and often preferable if feasible for the domain. Distributed systems are complicated, but with greater local autonomy can come greater tolerance to certain classes of faults and the ability to function in the face of partition or partial failure. The existing building control regime is probably too decentralized – there is too little coordination imposed on all of the various independent pieces of control going on.

An issue which unfortunately goes unexplored in this thesis are the practical economic, organization, and regulatory challenges of actually moving technology like that which we have developed into the marketplace, and ultimately into buildings. The building industry is by its nature conservative, given that it deals with expensive assets over long periods of time. We believe however that there is hope, coming from a number of directions. First, the overriding imperative of climate change and ideally, a concomitant increase in energy costs will increase the value placed on the advanced optimization capabilities we can achieve at relative low cost. Secondly, we believe that a platform like BOSS is capable of delivering not just the old building services better, but enabling new ones which will be desirable enough to drive the adoption of improved technologies.

Bibliography

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12:120–139, August 2003.

[2] Yuvraj Agarwal, Rajesh Gupta, Daisuke Komaki, and Thomas Weng. 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, pages 64–71, New York, NY, USA, 2012. ACM.

[3] J.D. Alvarez, J.L. Redondo, E. Camponogara, J. Normey-Rico, M. Berenguel, and P.M. Ortigosa. Optimizing building comfort temperature regulation via model predictive control. *Energy and Buildings*, 2013.

[4] American Society of Heating, Refrigerating and Air-Conditioning Engineers. *ASHRAE Standard 135-1995: BACnet*. ASHRAE, Inc., 1995.

[5] American Society of Heating, Refrigerating and Air-Conditioning Engineers. *ASHRAE Standard 55-2010: Thermal Environmental Conditions for Human Occupancy*. ASHRAE, Inc., 2010.

[6] Apache Software Foundation. Avro: A Data Serialization System. `http://avro.apache.org/`, 2009–2014.

[7] Anil Aswani, Neal Master, Jay Taneja, Andrew Krioukov, David Culler, and Claire Tomlin. Energy-efficient Building HVAC Control Using Hybrid System LBMPC. In *Proceedings of the IFAC Conference on Nonlinear Model Predictive Control*, 2012.

[8] Mesut Avci, Murat Erkoc, Amir Rahmani, and Shihab Asfour. Model predictive HVAC load control in buildings using real-time electricity pricing. *Energy and Buildings*, 2013.

[9] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 261–272, New York, NY, USA, 2000. ACM.

[10] Elizabeth A. Basha, Sai Ravela, and Daniela Rus. Model-based monitoring for early warning flood detection. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 295–308, New York, NY, USA, 2008. ACM.

[11] D. Beddoe, P. Cotton, R. Uleman, S. Johnson, and J. R. Herring. OpenGIS: Simple features specification for SQL. Technical report, OGC, May 1999.

[12] Willy Bernal, Madhur Behl, Truong Nghiem, and Rahul Mangharam. MLE+: Design and deployment integration for energy-efficient building controls. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '12, pages 215–216, New York, NY, USA, 2012. ACM.

[13] Jan Beutel, Stephan Gruber, Andreas Hasler, Roman Lim, Andreas Meier, Christian Plessl, Igor Talzi, Lothar Thiele, Christian Tschudin, Matthias Woehrle, and Mustafa Yuecel. Permadaq: A scientific instrument for precision sensing and data recovery in environmental extremes. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 265–276, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Carsten Bormann, Angelo Paolo Castellani, and Zach Shelby. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *IEEE Internet Computing*, 16(2):62–67, 2012.

[15] Randy Burch. Monitoring and optimizing pid loop performance. 2004.

[16] California Solar Initiative. California Solar Statistics. `http://www.californiasolarstatistics.ca.gov`.

[17] Matteo Ceriotti, Luca Mottola, Gian Pietro Picco, Amy L. Murphy, Stefan Guna, Michele Corra, Matteo Pozzi, Daniele Zonta, and Paolo Zanon. Monitoring heritage buildings with wireless sensor networks: The torre aquila deployment. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 277–288, Washington, DC, USA, 2009. IEEE Computer Society.

[18] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 111–122, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[19] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR – First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[20] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. Tenzing: A SQL implementation on the mapreduce framework. *PVLDB*, 4(12):1318–1327, 2011.

[21] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 379–390, New York, NY, USA, 2000. ACM.

[22] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *CIDR – First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[23] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[24] Automated Logic Corporation. ALC system architecture. `http://www.automatedlogic.com/files/documents/products/csconnrev7.pdf/`, 2007.

[25] Drury B. Crawley, Frederick C. Winkelmann, Linda K. Lawrie, and Curtis O Pedersen. EnergyPlus: New capabilities in a whole-building energy simulation program. In *Seventh International Conference of the International Building Performance Simulation Associateion*, pages 51–58, 2001.

[26] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of scidb: a science-oriented dbms. *Proc. VLDB Endow.*, 2(2):1534–1537, August 2009.

[27] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[28] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 25–25, Berkeley, CA, USA, 2012. USENIX Association.

[29] Fred Douglis, Michael Branson, Kirsten Hildrum, Bin Rong, and Fan Ye. Multi-site cooperative data stream analysis. *SIGOPS Oper. Syst. Rev.*, 40:31–37, July 2006.

[30] *Energy Outlook 2010*. Energy Information Administration, `http://www.eia.doe.gov/oiaf/ieo/index.html`, 2010.

[31] Varick Erickson, Miguel A. Carreira-Perpinan, and Alberto E. Cerpa. OBSERVE: Occupancy-Based System for Efficient Reduction of HVAC Energy. In *The 10th ACM/IEEE Int'l Conference on Information Processing in Sensor Networks (IPSN/SPOTS)*, 2011.

[32] Andrey Ermolinskiy, Sachin Katti, Scott Shenker, Lisa L Fowler, and Murphy McCauley. Towards practical taint tracking. Technical Report UCB/EECS-2010-92, EECS Department, University of California, Berkeley, Jun 2010.

[33] Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.

[34] International Alliance for Interoperability. End user guide to industry foundation classes, enabling interoperability. Technical report, 1996.

[35] Marc Fountain, Gail Brager, Edward Arens, Fred Bauman, and Charles Benton. Comport control for short-term occupancy. *Energy and Buildings*, 21(1):1 – 13, 1994.

[36] Sorabh Gandhi, Luca Foschini, and Subhash Suri. Space-efficient online approximation of time series data: Streams, amnesia, and out-of-order. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 0:924–935, 2010.

[37] Johannes Gehrke, Flip Korn, and Divesh Srivastava. On computing correlated aggregates over continual data streams. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD '01, pages 13–24, New York, NY, USA, 2001. ACM.

[38] Google, Inc. Protocol Buffers. `https://developers.google.com/protocol-buffers/`, 2010.

[39] Google powermeter. `http://www.google.org/powermeter/`.

[40] Graphite – Scalable Realtime Graphing. `http://graphite.wikidot.com/`, 2012.

[41] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Rec.*, 34:34–41, December 2005.

[42] Object Management Group. Corba component model 4.0 specification. Specification Version 4.0, Object Management Group, April 2006.

[43] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. Bolt: Data management for connected homes. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 243–256, Seattle, WA, April 2014. USENIX Association.

[44] Thomas J. Harris. Assessment of control loop performance. *The Canadian Journal of Chemical Engineering*, 67(5):856–861, 1989.

[45] Joseph M. Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J. Haas. Interactive data analysis: The control project. *Computer*, 32:51–59, August 1999.

[46] Joseph M. Hellerstein, Michael Stonebraker, and Rick Caccia. Independent, open enterprise data integration. *IEEE Data Eng. Bull.*, 22(1):43–49, 1999.

[47] C Huizenga, S Abbaszadeh, L Zagreus, and E Arens. Air quality and thermal comfort in office buildings. In *Healthy Buildings*, 2006.

[48] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. Cartel: a distributed mobile sensor computing system. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006.

[49] International Electrotechnical Commission. IEC61131: Programmable Controllers, 1992–2013.

[50] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[51] Information technology – Control network protocol – Part 1: Protocol stack, 2012.

[52] Harry L. Jenter and Richard P. Signell. Netcdf: A public-domain-software solution to data-access problems for numerical modelers, 1992.

[53] Xiaofan Jiang, Stephen Dawson-Haggerty, Prabal Dutta, and David Culler. Design and implementation of a high-fidelity ac metering network. In *IPSN'09*, 2009.

[54] Xiaofan Jiang, Minh Van Ly, Jay Taneja, Prabal Dutta, and David Culler. Experiences with a high-fidelity wireless building energy auditing network. In *SenSys '09: Proceedings of the 7th ACM conference on Embedded network sensor systems*, 2009.

[55] David Josephsen. *Building a Monitoring Infrastructure with Nagios*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[56] Alan Watton Jr. and Russell K. Marcks. Tuning control loops nonlinearities and anomalies. *ASHRAE Journal*, 21(1):46 – 52, 2009.

[57] Aman Kansal, Suman Nath, Jie Liu, and Feng Zhao. Senseweb: An infrastructure for shared sensing. *IEEE MultiMedia*, 14(4):8–13, October 2007.

[58] KEMA, Inc. Research Evaluation of Wind Generation, Solar Generation, and Storage Impact on the California Grid, 2010.

[59] EamonnJ. Keogh and MichaelJ. Pazzani. A simple dimensionality reduction technique for fast similarity search in large time series databases. In Takao Terano, Huan Liu, and ArbeeL.P. Chen, editors, *Knowledge Discovery and Data Mining. Current Issues and New Applications*, volume 1805 of *Lecture Notes in Computer Science*, pages 122–133. Springer Berlin Heidelberg, 2000.

[60] Younghun Kim, Thomas Schmid, Zainul M. Charbiwala, Jonathan Friedman, and Mani B. Srivastava. Nawms: nonintrusive autonomous water monitoring system. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 309–322, New York, NY, USA, 2008. ACM.

[61] JeongGil Ko, Răzvan Musăloiu-Elefteri, Jong Hyun Lim, Yin Chen, Andreas Terzis, Tia Gao, Walt Destler, and Leo Selavo. Medisn: medical emergency detection in sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 361–362, New York, NY, USA, 2008. ACM.

[62] Andrew Krioukov, Stephen Dawson-Haggerty, Linda Lee, Omar Rehmane, and David Culler. A living laboratory study in personalized automated lighting controls. In *BuildSys'11*, 2011.

[63] Andrew Krioukov, Gabe Fierro, Nikita Kitaev, and David Culler. Building application stack (BAS). In *Proceedings of the 4th ACM Workshop On Embedded Sensing Systems For Energy-Efficiency In Buildings*, 2012.

[64] Pacific Northwest National Laboratory. VOLTTRON: An Intelligent Agent Platform for the Smart Grid. `http://gridoptics.pnnl.gov/docs/VOLTTRON_An_Intelligent_Agent_Platform_Flier.pdf`, 2013.

[65] Chris Leung. Extended environments markup language. `http://www.eeml.org/`.

[66] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.

[67] Chieh-Jan Mike Liang, Jie Liu, Liqian Luo, Andreas Terzis, and Feng Zhao. Racnet: a high-fidelity data center sensing network. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 15–28, New York, NY, USA, 2009. ACM.

[68] Ying Liu, Nithya Vijayakumar, and Beth Plale. Stream processing in data-driven computational science. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, GRID '06, pages 160–167, Washington, DC, USA, 2006. IEEE Computer Society.

[69] A. Lovins, M. Odum, and J.W. Rowe. *Reinventing Fire: Bold Business Solutions for the New Energy Era*. Chelsea Green Publishing, 2011.

[70] Inc Lutron Electronics Co. Lutron light control strategies. `http://www.lutron.com/TechnicalDocumentLibrary/Energy_Codes_and_Standards_g.pdf`, 2013.

[71] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7), July 2004.

[72] McKinsey & Company. Unlocking energy efficiency in the U.S. economy, July 2009.

[73] Simon Miles, Paul Groth, Miguel Branco, and Luc Moreau. The requirements of recording and using provenance in e-science experiments. Technical report, University of Southampton, 2005.

[74] Evan Mills. Building commissioning: A golden opportunity for reducing energy costs and greenhouse-gas emissions. Technical report, California Energy Commission Public Interest Energy Research, 2009.

[75] Lufeng Mo, Yuan He, Yunhao Liu, Jizhong Zhao, Shao J. Tang, Xiang Y. Li, and Guojun Dai. Canopy closure estimates with greenorbs: sustainable sensing in the forest. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 99–112, New York, NY, USA, 2009. ACM.

[76] MODICON, Inc., Industrial Automation Systems. Modicon MODBUS Protocol Reference Guide, 1996.

[77] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan Hui, , and David Culler. Rfc 4944 – transmission of ipv6 packets over ieee 802.15.4 networks. Technical report.

[78] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 4–4, Berkeley, CA, USA, 2006. USENIX Association.

[79] National Instruments, Inc. LabVIEW System Design Software. `http://www.ni.com/labview/`.

[80] Open Building Information Exchange (oBIX) v1.0, 2006.

[81] America's Energy Future Energy Efficiency Technologies Subcommittee; National Academy of Sciences; National Academy of Engineering; National Research Council. *Real Prospects for Energy Efficiency in the United States*. The National Academies Press, 2010.

[82] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley db. In *FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–191, 1999. http://www.odysci.com/article/1010112988807638.

[83] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 267–273, Berkeley, CA, USA, 2008. USENIX Association.

[84] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[85] OPC Task Force. OPC common definitions and interfaces, 1998.

[86] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[87] M. A. Piette, S. Kiliccote, and G. Ghatikar. Design of an energy and maintenance system user interface for building occupants. In *ASHRAE Transactions*, volume 109, pages 665–676, 2003.

[88] M. A. Piette, S. Kiliccote, and G. Ghatikar. Design and implementation of an open, interoperable automated demand response infrastructure. In *Grid Interop Forum*, 2007.

[89] Mary Ann Piette, Girish Ghatikar, Sila Kiliccote, Edward Koch, Dan Hennage, Peter Palensky, and Charles McParland. Open automated demand response communications specification (version 1.0). Technical report, Lawrence Berkeley National Laboratory, 2009.

[90] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 49–, Washington, DC, USA, 2006. IEEE Computer Society.

[91] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[92] ShankarR. Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A service framework for ubiquitous computing environments. In GregoryD. Abowd, Barry Brumitt, and Steven Shafer, editors, *Ubicomp 2001: Ubiquitous Computing*, volume 2201 of *Lecture Notes in Computer Science*, pages 56–75. Springer Berlin Heidelberg, 2001.

[93] Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 253–266, New York, NY, USA, 2008. ACM.

[94] Jeffrey Schein, Steven T. Bushby, Natascha S. Castro, and John M. House. A rule-based fault detection method for air handling units. In *Energy and Buildings*, 2006.

[95] John Sellens. Rrdtool: Logging and graphing. In *USENIX Annual Technical Conference, General Track*. USENIX, 2006.

[96] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[97] Siemens. APOGEE Building Automation Software. `http://w3.usa.siemens.com/buildingtechnologies/us/en/building-automation-and-energy-management/apogee/pages/apogee.aspx`.

[98] Michael Stonebraker, Jacek Becla, David J. DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B. Zdonik. Requirements for science data bases and scidb. In *CIDR*, 2009.

[99] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of scidb. In Judith Bayard Cushing, James C. French, and Shawn Bowers, editors, *Scientific and Statistical Database Management - 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011. Proceedings*, volume 6809 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2011.

[100] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34:42–47, December 2005.

[101] W.Y. Svrcek, D.P. Mahoney, and B.R. Young. *A real-time approach to process control*. Wiley, 2006.

[102] Alexander S. Szalay, Peter Z. Kunszt, Ani Thakar, Jim Gray, Don Slutz, and Robert J. Brunner. Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 451–462, New York, NY, USA, 2000. ACM.

[103] Jay Taneja, Jaein Jeong, and David Culler. Design, modeling, and capacity planning for micro-solar power sensor networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 407–418, Washington, DC, USA, 2008. IEEE Computer Society.

[104] Jay Taneja, Andrew Krioukov, Stephen Dawson-Haggerty, and David E. Culler. Enabling advanced environmental conditioning with a building application stack. Technical Report UCB/EECS-2013-14, EECS Department, University of California, Berkeley, Feb 2013.

[105] Arsalan Tavakoli. *Exploring a Centralized/Distributed Hybrid Routing Protocol for Low Power Wireless Networks and Large Scale Datacenters*. PhD thesis, EECS Department, University of California, Berkeley, Nov 2009.

[106] The Mathworks, Inc. Simulink: Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink/`.

[107] Gilman Tolle. Embedded Binary HTTP (EBHTTP). IETF Internet-Draft draft-tolle-core-ebhttp-00, March 2010.

[108] Tridium, Inc. Tridium product guide. `http://www.tridium.com/galleries/brochures/2013_Tridium_Product_Guide.pdf`, 2013.

[109] U.S. Energy Information Administration. Commercial Building Energy Consumption Survey. `http://www.eia.gov/consumption/commercial/`, 2007.

[110] Gregory J. Ward. The radiance lighting simulation and rendering system. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 459–472, New York, NY, USA, 1994. ACM.

[111] Thomas Weng, Anthony Nwokafor, and Yuvraj Agarwal. BuildingDepot 2.0: An integrated management system for building analysis and control. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, BuildSys'13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[112] Geoffrey Werner-Allen, Stephen Dawson-Haggerty, and Matt Welsh. Lance: optimizing high-resolution signal collection in wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 169–182, New York, NY, USA, 2008. ACM.

[113] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, Lombard, IL, 2013. USENIX.

[114] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[115] Peter Zadrozny and Raghu Kodali. *Big Data Analytics Using Splunk*. Apress, Berkeley, CA, 2013.

[116] Matei Zaharia, N. M. Mosharaf Chowdhury, Michael Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010.

[117] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: A fault-tolerant model for scalable stream processing. Technical Report UCB/EECS-2012-259, EECS Department, University of California, Berkeley, Dec 2012.

[118] J. G. Ziegler and N. B. Nichols. Optimum Settings for Automatic Controllers. *Transactions of ASME*, 64:759–768, 1942.