Who's the BOSS? Getting Buildings under Control

May 17, 2012

1 Introduction

Technology trends towards greater instrumentation, data collection, and analysis are clear: low cost computation and communication make it feasible to collect data at much wider scale and finer granularity than was ever before practical. At societal scales, even modest data volumes "add up" and become a challenge to store, process, and ultimately extract value from. We are in the midst of a fundamental transition: data is changing from being a rare, high-quality, and clean product to a dirty yet ubiquitous commodity.

This transition is taking place in many sectors of the economy at different rates. Traditional computing environments have been among the first to move towards this new world, since the infrastructure for collecting and managing the data can largely be built on the computing infrastructure itself. Other areas like manufacturing plants and supply chains have also undergone significant automation and instrumentation as corporations seek additional efficiencies and cost-savings through strategies like just-in-time manufacturing and real-time inventory control. However, the built environment – the public, commercial, and residential buildings and streets where most people spend nearly all of their time – has undergone significantly less change despite these underlying trends.

When considering the built environment, it is important to consider the sources and types of data we are concerned with. In some sense, buildings are like plants, manufacturing a hospitable indoor environment; there are a large number of sources of data surrounding this process: data about temperature, humidity, set-points, flows, and light levels at many points of the plant. However, in important ways buildings are *not* like plants, because they are essentially "open systems" whose operation is subject to the behavior of occupants. Information about the occupants exists in many forms: building access logs, network traffic data, security camera footage, and work schedules yet is rarely incorporated into the system control plan. Furthermore, the built environment exists within a larger system, some elements of which are amenable to prediction and analysis; weather and traffic forecasts are routine, and many more phenomena like water levels, snow-pack conditions, solar and wind energy availability, and fuel prices are routinely observed and could be factored into planning and control loops.

Existing systems systems in the built environment are particularly ill-suited to the data-centric and networked future. Although some system components have been brought up-to-date, the architecture has not significantly evolved from the day where state-of-the-art control meant programing PID loops and ladder logic into mechanical computers. As a result, integrating these components into a modern networked systems architecture is challenging because the underpinnings are obscure or proprietary, and the legacy systems do not provide the isolation, security, safety, and liveness properties that would allow the deployment of new technologies while lowering the consequences of failure. Our contribution is to propose a forward-looking distributed operating system architecture that addresses these shortcomings; allowing management and control of distributed physical resources while enabling a broad range of applications for the built environment.

2 Background and Motivation

2.1 Existing Building Systems

A large modern commercial or industrial facility 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), a good air quality (ventilation), and sufficient lighting. Other systems provide for life safety with fire alarms and security, and the design of a particular building takes into account many other considerations. All of the active components of the system typically require management interfaces, just like components in a computer system. System managers need the ability to troubleshoot problems, adjust schedules, and change set points.

This control in existing building systems operates on two levels. Direct control is performed in open and closed control loops between sensors and actuators. In a direct control loop, a piece of logic examines a set of input values, and computes a control decision which directly commands an actuator. These direct control loops frequently also have configuration parameters which govern their operation; these are known as set points and are be 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 frequently reflected physically in the components and networking elements making up the system: direct control is performed by embedded devices called Programmable Logic Controllers (PLCs), which are directly connected to the sensors and actuators, while supervisory control is managed over a shared bus between the PLCs. This architecture is logical for implementing direct 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. One property of this system is that interaction with the outside world is achieved through the changing of set points (and less frequently, control logic), over the supervisory control bus.

Figure: control system physical/logical architecture

Figure: HVAC loop with VAVs, chiller,

2.2 Motivating Applications

The need for a new operating system for the building is motivated by several concrete applications we wished to deploy on our building. Applications developed for the built environment range from fine-grained energy accounting to localized climate control. Some of them require input from the occupants; data from sensors attached to personal items and control signals initiated by occupants. While others require no direct occupant participation. For example, control applications that override local control loops to maximize energy efficiency while maintaining occupant comfort. We also wish to support dashboard applications that gives occupants and the building manager a global view of building performance, or analysis applications that run outside the built environment to analyze the operational performance of the building, feed performance forecasting models or combine building sensor data with external signals, such as weather and energy pricing.

Although versions of these applications can be deployed directly using existing systems, developing them requires low-level knowledge about the construction of a building, restricting their generality. It also requires careful reasoning in each case about the effect of various types of networking and system failures. 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. Experience with ad-hoc development of this style of application led us to the conclusion that better abstractions and shared services would admit both faster and easier application development, as well as more consistent and robust fault tolerance.

The first motivating application is a *temperature float* application. Ordinarily, temperatures within a zone is controlled to within a small number of degrees Celcius. This drive for reaching an exact setpoint is actually quite inefficient, because it means that nearly every zone is heating or cooling at all times. A more relaxed strategy would be one of *floating*: not attempting to effect the temperature of the room when it is within a relative wide band; however this is not one of the control policies available in typical commercial systems.

A second application was developed as part of a program to improve efficiency and comfort by giving occupants direct control of their spaces. Using a smart-phone interface, occupants are able to directly control the lighting and HVAC systems in their spaces; a social component resolves conflicting preferences between neighboring occupants. The application requires the ability to command the lights and dampers in the space to, for instance, guarantee that the lights are on for a period of time and to deliver services like a “blast” of hot or cold air to address temperature or ventilation complaints.

A third application is an energy audit and live energy viewer of the building. Using a smart-phone interface, occupants input information about the structure of the building and the relationship between sensors and devices. This requires access to a uniform naming structure and streaming sensor data coming from physically-placed sensors. The former to capture the inter-relationships between building locations, sensors, and loads (energy consumers) and the latter to provide up-to-date information about physical measurements taken in locations throughout the building. For example, an occupant may wish to see the total energy consumed by all plug-loads on a particular floor.

Our final application is an offline analysis tool for characterizing data streams and finding if the semantic information, as captured by the naming structure mentioned above, is accurate. This application requires access to the uniform naming structure and assocaited historical sensor data.

2.2.1 Architectural implications

The *temperature float* application highlights the need for attaining feeds in real-time and locating the appropriate setpoint actuator for each temperature control unit. The architecture must treat streaming data natively, providing mechanisms for subscribing to incoming feeds. It must also provide a way to actuate devices explicitly. Furthermore, contextual metadata should be in place so that application can determine which devices it is receiving data from and which device it is actuating.

The *mobile climate control* application highlights the need for leasing control to the HVAC components and lighting fixtures. Access control is a concern here. The "lease" is a useful mechanism for providing *temporal isolation*. A system must include a mechanism that allows application to temporarily own resources and forcibly reclaim those resources if necessary.

The *audit* application highlights the need to couple semantic information with streaming sensor data in a uniform fashion and a way to meaningfully combine it with raw sensor data. Our system should provide a mechanism that allows users to leverage this structure when possible without constraining them if the structure needs to change over time. It also emphasizes the need for real-time data cleaning and aggregation. Sensor feeds are quite dirty, often containing errant or missing values. Security and privacy play a factor here as well. This application makes use of personal data quite explicitly this motivates the need for a way that applications and users can control who can access their personal feeds.

The *analysis* application highlights the need for generic interface for ease of integration with external tools. The filesystem presents the deployment as a distributed filesystem and allows users to seamlessly interact with and query their environment without having to write tools to integrate with their local application. In order to allow a broader range of applications, especially those that involve analysis rather than direct application building, our system must have multiple interfaces for access deployment information and sensor data.

3 Design

We propose a new architecture for building systems where the main goal is to **ease application development and support many simulatenously running applications**. More specifically we look to:

- improve programmabilty through useful *abstractions* for application developers.
- provide *shared services* and *controlled access* to underlying sensors, actuators.
- maintain *isolation* and *protection* while allowing *explicit sharing* between applications.
- maintain robustness and *fault tolerance* for running applications and resources.

Our architecture consists of three layers, the hardware-abstraction and control-transaction layer, the naming and data-access layer, and the application layer. A high-level view of each layer and its constituents is shown in Figure 1. In layer 1, the **hardware abstraction layer** consists of a hierarchical set of interfaces which provide interfaces to the underlying physical building hardware at increasingly abstract levels. The **transaction** interface is the runtime abstraction responsible for implementing command decisions made by applications. In layer 2, we provide a rich **application runtime** for creating reliable software on top of a building, which includes a global *namespace* provided by the directory service, and time-series service for accessing and cleaning historical data.

For applications that require a more structured, uniform namespace we provide a filesystem interface over the directory and time-series service. The filesystem allows users to interact with building sensors through read/write file operations. The fileystem also provides a level of *security* while performing file operations and sharing data with other applications; users are provided with a unique identifier and associated permissions that allow them to view and share files among and across applications.

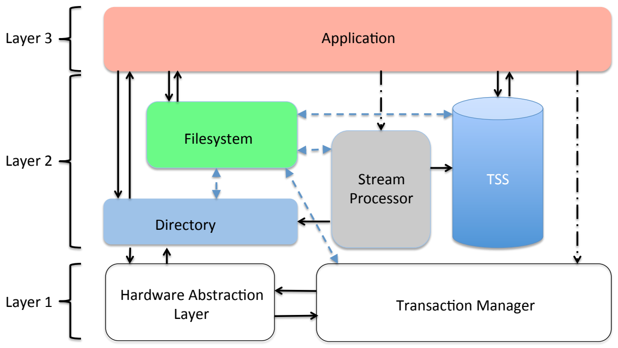

Figure 1: System architecture. Layer 1 consist of the hardware abstraction (HAL) and control-transaction components. Layer 2 consists of components for naming and data access. Layer 3 is the application layer. The application layer can access any of the underlying components directly, except the HAL.

Finally, layer 3 consists of applications for the built environment. The underlying architecture should support multiple applications running simultaneously on the underlying infrastructure. Some may consist of **control processes** that represent existing control strategies. Robust control processes require support for wrapping applications to allow them to be dynamically migrated by the system, and mediated access to other required functionality.

3.1 Hardware Abstraction Layer

One function of traditional computer operating systems is to provide a convenient abstractions of physical resources.

Architectural component	Function	Placement
Hardware presentation layer	Make the primitive low-level operations of the hardware available using a common set of interfaces.	As close to the physical sensors as possible; ideally, co-located with the transducer.
Hardware abstraction layer	Map the low-level functions of the physical hardware into multiple higher level abstractions.	Anywhere; should be persistent.
Directory	Place devices from the HPL and HAL into a namespace.	Replicated; may be offsite.
Historian	Maintain a history of readings from the sensors and actuators.	Replicated; may be offsite.
Filesystem	Provides explicit structure to the underlying namespace.	Replicated; may be offsite.
Transaction manager	Provide "all or nothing" semantics when applying control inputs; schedule conflicting and concurrent commands.	In the same failure domain as the HAL used to affect the changes.

Table 1: Description of architectural components

The interface to hardware devices is low-level and heterogeneous in both computer systems and in building systems. 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 communications protocols and presents hardware capabilities through a uniform, self-describing interface. The HPL 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. The HPL provides a consistent method of collecting data from the sensors and commanding individual actuators. The layer also provides the basis of naming in this system: each sense or actuation point is named with a single, globally unique identifier. This functionality is distributed across the computing resources closest to each sensor and actuator; ideally it is implemented natively by each device.

On top of HPL we form a Hardware Abstraction Layer. The task of this layer is to provide the basis for mapping underlying primitive hardware operations into useful higher-level relationships. As in computer systems, it is often desirable to access the same physical hardware using multiple abstractions. For instance, multiple network-layer protocols are run on top of the same physical network interface. The corollary for the built environment is that multiple views of the same physical sensor and actuator points are possible. For instance, answering the question of which sense points are in a particular room has no particular relationship to which systems: lighting, climate, security, that these sense points are part of. The HAL is responsible for representing the various types of metadata about the sense and actuation points.

In order to represent the relationships between sensors and actuators, the HAL uses key-value *tags* which can be applied to each sense and actuation point. By using tags, we avoid the need for imposing a schema on the metadata to be provided, allow representation of both hierarchical and subset relations, and support mapping multiple ontologies onto these points. These tags can be applied directly at the device itself, or within the global namespace formed by the directory. This is important, because some of this information such as the type of instrument present, and its sensing modalities are natural to apply at the point of instrumentation, while other information like the relationship of an actuator to other system components are only known later, once a global view of the system has been established.

A HAL from a physical system differs from that for a computer system in several ways. A major difference is that in a computer system, the "virtual" representation of the system (made up of the device drive state, *etc*) contains essentially everything relevant about the state of the underlying physical system. In the physical environment, the virtual representation is unlikely to be nearly as complete; the underlying physical system has complicated physical dynamics which is it extremely rare and difficult to fully capture. Therefore, we have much less observability and control over how actions made on the virtual representation will impact the physical system and the observable quantities. Furthermore, in a computer system the types of hardware are known ahead of time: the architecture of all computers falls essentially into one of several models of bus interconnections of the various components. In contrast, although the individual components of building systems are relatively standardized, the relationships between them depend on the design of a particular building. For instance, many buildings have similar systems for providing cooling: air is blown through ducts, where it is cooled and ultimately distributed into spaces through variable air-volume (VAV) boxes. However, the behavior of this system depends heavily on the routing of the network of ducts and placement of VAV boxes, which is customized for each individual building.

3.2 Directory

The HAL provides distributed real-time access to the system under control; however, implementing control strategies and applications requires a global view of the system and access to stored data. The directory provides a **global view** of this metadata. The sensor and actuation points identified by the HPL are objects in the namespace of the system; because this information is distributed across all of the sensors and actuators, it is necessary to have an operating system component which allows applications to query it in a straightforward way. The directory service allows applications to locate sense and actuation points by performing queries on the key-value metadata. It is often desirable to query across multiple ontologies at the same time; for instance, "locate the power reading (measured in kW) for the air conditioning component servicing a particular room." This requires accessing at least three separate mappings: the instrument mapping, recording information about the instrument, the spatial mapping locating that room, and the systems mapping that room to the system providing service.

The directory can be thought of as an analog of the standard unix `/dev` namespace which has been redesigned around database filesystem principles. The `/dev` tree is typically organized to match the underlying system architecture of the computer system; for instance, `/dev/bus/usb/001` contains devices on the system's first USB bus. Building systems have an analogous structure corresponding to the topology of the building network, which specifies which PLCs and head-end nodes the sensors is accessed through.

3.3 Filesystem

For a certain class of applications we wish to support, files are the least common denominator for accessing sensor information. Many services on sensor data can be provided through the filesystem interface. Allowing applications to read/write files to interact the building is an effective approach for supporting legacy client applications. For example, applications written in Matlab or R would not require any integration code to fetch sensor data; only local file reads/writes are necessary to attain the necessary data for analysis.

The filesystem interface provides a structured version of the directory service and provides a direct way to express structured semantic information and couple it with sensor/actuator information. For example, spatial relationships are inherently hierarchical. Buildings have floors, floors have rooms, room have sub-spaces. Sensors can be placed within the spatial namespace in order to encode the relationship between sensors and spaces explicitly and can be accessed via the associated spatial path (i.e. `/building/floor4/410/temp/reading`). Moreoever, the use of symbolic links allows users to create separated sub-trees that capture other hierarchical relationships, such as aggregation groups. The HVAC system has no clear hierarchical relationship, but the components of it do. The HVAC system consists of vents, ac units, heaters, pumps, valves, etc. Each of these components also have sensors attached. Therefore it can be constructed in a similar fashion as the spatial namespace. *Aggregation groups can guide namespace construction when there is no clear structural hierarchy in the physical system.* Moreover, symbolic links allow for sensor access to be referred to through mutliple names.

Security and access control can be taken directly from the Unix filesystem security model. Files that represent physical and logical resources are owned and shared by the user that created them. Each application sees a different view of the files in the system and can share access by giving read/write/execute rights to other users, as identified by their `username`.

Since we are fundamentally dealing with streaming data, the filesystem should provide access-to and management-of stream processing facilities. This can be achieved through overloaded Unix filesystem mechanisms. For example, the use of "pipes" can accomplish the goal of re-direction and data-flow declaration. Applications should be able to pipe streaming data through local processing scripts as well. Actuators can also be represented as a file, where reads fetch the current state and writes change the state of the actuator.

3.4 Historian

The historian provides access to historical data, frequently needed for analysis. The data model is relatively simple: there are a large number of time-series generated by the sensors and actuators; a time series is simply a set of related `(timestamp, value)` pairs which we call *readings*. Typical access patterns for historical data have characteristics somewhat different than those traditional relational databases are optimized for. Data are typically accessed by performing range-queries over time stamps and streams; for instance a typical query might extract all room light-level readings for a period of one month. Even a modest-sized installation will easily have tens of billions of readings stored and even simple queries have the potential to touch hundreds of points. New data is nearly always gradually appended to the end of the time series, because the data comes from individual measurements taken by the sensors and published by the HPL.

Finally, it is impractical to keep the highest-resolution data from all sensors forever since the value which can be extracted from this old data frequently can't justify the cost of maintaining the history; therefore the historian must have the facility for degrading or compressing older data.

3.5 Stream processor

Before stored data can be used, it must typically be **processed and cleaned** since raw sensor data is quite dirty; for instance, timestamps need to aligned and the data may be resampled and smoothed. Furthermore, since real-time data is important, especially for control applications, there is a clear need to process and clean the data in real time. A stream processing element is an important service that must be included in the archiecture. The engine should not only help manipulate individual data points, but should optimize for operating on vectors of data points. Most of the oeprations for data cleaning are performed on data vectors, the same should be provided here as well.

3.6 Control transactions (CTX)

It is desirable that control policies be extended or modified across multiple failure domains for a multitude of reasons. Internet-based services may have more processing and storage than is available in the building, or may wish to implement proprietary logic. It may also be simpler and more concise to express the desired outcome in terms of a set of changes to be made than to program the underlying controllers individually; ideally changes would be expressed in terms of desired outcome (make this room warmer) rather than the detailed implementation (increase the set point, re-set the chilled water temperature, and slow down the cooling loop pumps). However there are real problems with performing automatic direct or supervisory control over the Internet or building network. For direct control, latency may be an issue. There can be concurrency issues when changes are made by multiple parties. A failure of one domain can leave the system in an uncertain state.

In order to help resolve these issues, we propose using an extended *transaction* metaphor for affecting changes to control state. Transactions in database systems are a way of reasoning about the guarantees made when modifying the data. In a control system, a transaction is a similar way of reasoning about what happens when sequences of control inputs are made by different parties.

Control transactions operate conceptually at the supervisory level; but we expose a significantly richer interface than simple set-point adjustment. Most basically, a control transaction may write new values to a set of actuators, or update parameters being used by direct control loops; for instance, command the lights to turn on or change a term in a PID controller. Generally, a transaction may replace the direct control logic with a new piece of logic; for instance, replace a PID loop with a new loop. In addition to the action to be performed, a control transaction consists of several other concepts. One is a *lease time* during which the control action is to be valid, or the new loop active. When the lease expires, the transaction manager will execute a separate "END" block of the transaction and then restores control of the system to the next scheduled direct controller.

- **Concurrency Isolation**: Traditional database systems inform the user what view of the data he will see, when there are multiple writers are present. The typical choice is to provide strong *isolation* between multiple accessing processes. In a control system, isolation takes on a different meaning because control taken on one set of actuators could effect the outputs of other actuators and sensors. We will define several senses in which transactions may be "isolated" while commanding a system.
- **Scheduler**: Transaction scheduling goes hand-in-hand with concurrency; each transaction is, ultimately, a set of small changes which must be applied to the system. These must be scheduled in some way, respecting the concurrency constraints of an underlying hardware as well as the impact one set of actions will have on other running transactions.
- **Abort**: Running database transactions can be aborted when required due to a transient failure, a deadlock, or user command. In a control system, there may be other causes. A higher-priority transaction may require access to a resource currently used by a lower-priority transaction, or one of a set of changes may fail, requiring processing on the transaction to be aborted. Unlike database transaction, the control transaction manager may not have control of the underlying data; for instance, there is no way to "lock" the value output by a sensor; it crossing a value could necessitate aborting a control sequence.
- **Rollback**: Reverting a database transaction typically involves only throwing away changes which have yet to be committed; or not writing a commit record. Reverting a sequence of control decisions may be possible in some cases; for instance, going back to an

earlier set point. Generally, control outcomes are frequently "path dependent"; it's never possible to revert the physical state of the world but it is possible to shift to other control strategies.

3.6.1 Isolation

The notion of how two sets of control inputs can be isolated from each other in a control system is considerably more broad than that available in a database system, because transactions may interact through the physical environment. Transactions may take complex actions which interact in physical space and occur at different levels of abstraction.

1. **Actuator** isolation refers to isolation between the physical actuation points being accessed in the transaction. Different transactions which are concurrent and access any of the same control points are considered to be in conflict.

2. **System** isolation uses hierarchical locking to guarantee that when a control input is made, no other control input is made to the same system that would cause a conflicting result in physical space. For instance, consider a room with two lighting systems: it should not be possible for two concurrent transactions to actuate them simultaneously.

3. **Model-based** isolation is the most advanced type of isolation, at attempts to reason about isolation in terms of the effect of an action. For instance, a transaction which increased the light level in a section of a room might or might not conflict with a second concurrent transaction which increased the light in a separate part of the same room, depending on a model of light propagation in that room.

3.6.2 Security and Authorization

The transaction manager is also the logical point at which to enforce a security policy, controlling which principals can perform which control inputs…

The filesystem handle sceurity separately…

3.7 Control processes

The goal of this architecture is to enable the development of novel control strategies for CPS's in the built environment which use model-predictive control, occupant feedback, and many other innovative policies and then implement them in the real-world rather than leave them confined to either the research setting or proprietary, vertically integrated solutions. Collectively, we call the software implementing these new strategies "control processes," although this name belies the potential complexity in this layer. Essentially, control processes use the interfaces presented by the Transaction Manager and the Archiver to implement control logic; we make no particular requirements about the architecture of the CP's themselves. Equally possible CPs are a MATLAB script which subscribes to real-time data and submits control inputs based on a model, or a Ruby-on-Rails application which communicates with occupants and uses their input to make decisions.

Because of the careful design of transactions and the archiver, many of these new applications can be developed without the stringent reliability requirements traditionally required of control logic. Furthermore, they have fewer restraints on where they can be placed in the computing infrastructure, because 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.

4 Implementation

In order to evaluate our architecture for building software systems, we have developed a prototype implementation called BOSS , or Building OS System. The BOSS contains prototype implementations of the hardware abstraction layer, application runtime, transaction manager, and a number of control processes and is currently being used by researchers. After proposing a detailed design for each of these components, we evaluate each piece and show how traditional operating systems questions such as sharing, isolation, and abstraction are present. As a whole, we take a distributed systems view of this operating system and consider how well our architecture represents a useful decomposition of the problem at hand. In other words, we consider how well we can fully implement the desired functionality within our architecture; we make the case that this architecture admits clean, simple interfaces between the different components. We also analyze how well the architecture meets higher-level goals like providing reliability in the face of network partition and component failure while allowing applications to be built using appropriate tools.

Our evaluation considers the installation of BOSS in our test building: a large commercial building located on our department's campus. The building is approximately $90,000$ square feet and contains a mix of open collaborative spaces, faculty offices, and electronics fabrication laboratories. The facility is managed using a traditional Building Management System (BMS) with control over the climate plant, lighting, security system, and other building systems. In addition, we have added several additional sensing systems in order to enable various research goals; these included a wireless plug-load monitoring system and a prototype device-free localization system used for researching occupancy detection.

figures: building image, climate plant diagram.

4.1 Hardware Presentation and Abstraction

Existing building systems are made up of a huge number and of 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. Another challenge is making sense of the relationships between these components. The basis for these two tasks are provided by presentation and abstraction layers.

figure: multiple views of the hardware figure of the building

4.1.1 HPL

The presentation layer allows higher layers to retrieve data and command actuators in a uniform way. Our hardware presentation layer builds on previous work in this space and uses a simple RESTful protocol to represent and encapsulate the data sources and actuators present in the building. The profile presents transducers as embedded web servers which provide data from their sensors and actuators over HTTP. The design of this system must address several practical considerations: consumers of data need the ability to subscribe to the data points they are interested in and receive notifications of new or changed data values. This layer also provides the basis for naming in the operating system: each individual sense point or actuator is durably named with universally unique identifier (`uuid`).

Of key importance when interfacing with existing systems is ease of integration. In support of this, we have developed robust client library support for implementing the HPL. It takes care of much of mechanics of providing the interface and allows driver writers to focus on implementing only sensor or actuator-specific logic. It places sense and actuation points within a user-defined hierarchy of HTTP resources, while allowing the use of abstracted "drivers" which separate the device-specific logic needed for talking with a device (communications protocols, *etc*) from the site-specific configuration (network locations, sampling rates, and the like). The client library then manages the publication of new data from the device to any number of subscribers, supporting both push and pull modalities for data delivery. Clients not needing to receive every reading from the device may simply poll it using HTTP, while other clients not wishing to miss data may install a subscription which causes the HPL to begin sending data to them within an HTTP `POST` request. The HPL will perform buffering for these clients if it is able, so that no data will be lost if the client encounters a partition or other failure.

figure: example driver config file.

figure: software architecture for HPL/HAL drivers. container with drivers, publisher, config file, you got it baby.

In our implementation, most sensors and actuators do not implement the HPL directly because they are existing legacy systems. We elevate all of these existing sensors and actuators to this common level through the use of gateways and protocol translators; once completed, all of the inherent complexity in these legacy systems is hidden behind our simple RESTful interface. In the course of integrating our test buildings (and other buildings) we have produced about 15 different HPL drivers; they are typically small since each driver contains only code specific to accessing a particular device.

figures: example sMAP object, example actuation interaction, HPL library interface, table of drivers.

4.1.2 HAL

Retrieving data or commanding actuators without a semantic understanding of the relationships between the various components is not very useful. The hardware abstraction layer captures these complex relationships between system components. Our HAL builds on top of the objects and naming provided by the presentation layer, by allowing key-value tags representing the metadata to be applied to sensing and actuation points. In order to provide structure on an otherwise flat namespace of keys, we structure the names of the keys hierarchically. We have standardized key names dealing with describing different types of instruments, the physical location of sense points, and various types of building systems. Each of these different types of metadata receive their own namespace of keys; for example, all metadata regarding the location of an instrument is specified using keys starting with `Metadata/Location/`. This has the advantage of placing namespace clashes and also encoding which ontology a particular key name refers to.

figure: example time-series with tags

figure: example of existing ontologies: instrument, location, system, electric tree

Our implementation of the HPL allows tags to be applied to data at the same place that the data is generated. Although this might appear to be mixing layers, it is valuable for several reasons. Primarily, it is because some meta-data is best provided automatically at this layer. For instance, when collecting information from

Name	Description	Sensors Used	Actuators Used	Type of Control	HPL Adaptors
Demand Ventilation	Ventilation rates are modulated to follow occupancy, and room temps can float at night.	VAV temps & airflow	N/A	Supervisory	BACnet
Supply Air Temp Control	AHU supply air temp is modulated to minimize energy consumption while maximizing occupant comfort [**?**].	AHU SAT, VAV temps & airflow	AHU SAT	Supervisory	BACnet
VAV Control	Individual variable air-volume boxes are controlled to create better models for future experiments.	VAV temps & airflow	VAV damper positions	Direct	BACnet
Building Audit	Plug loads throughout the building are surveyed to enable DR and energy efficiency efforts.	Plug-load meters	N/A	N/A	ACme [**?**]
Personal Building Control	Users are presented with lighting and HVAC inputs for local control of building systems.	Light power, VAV temps	Light level, VAV airflow	Direct	BACnet

Table 2: Applications using BOSS that are currently deployed.

a wireless sensor network, it is natural to include the serial number of the device which produced the reading to make data searchable based on the instrument name. The client library sends changes in metadata to any subscribers who are interested so they can maintain a synchronized view of the metadata. In general, these tags are supplied in one of roughly three places: an HPL driver may apply tags which generally provide information about the instrument in use. A user may specify additional tags at installation time (generally in the sensor configuration file) which include information about the installation such as physical and network location. Finally, third-party users may apply tags at a later point once more is known about the instrument – how it relates to systems in the building, what it is used for, *etc*.

4.2 Directory service

The set of points created by the HPL and described by the HAL make up the namespace of the system; a classic distributed systems issue is specifying how this namespace is accessed. Our prototype allows applications to query this namespace in a centralized manner. Because users can specify any tag name, it is impossible to map this namespace cleanly into an SQL schema with tag names as columns. However, relational databases have an attractive query model, allowing users to pose complicated queries to find records. Therefore, we implemented a simple column-style store on top of a relational database in which tag names appear as virtual columns, and each row represents a single time-series. This choice lets us quickly prototype the application interface to the metadata while receiving acceptable performance, while receiving acceptable performance.

Figure: example of virtual table with rows as streams and columns as tag names

The directory subscribes to all streams present in the HPL, and listens for changes to metadata. Internally, it represents the data using a single row for each `(streamid, key, value)` tuple. The directory service contains a sql-to-sql compiler which rewrites queries over metadata treating them as columns into queries on this table. It also inserts access checks referencing a separate table which stores access control lists governing which streams are visible to which users. This language supports close-to-SQL syntax for **select**, **update**, and **delete**, allowing users to explore and update the stored namespace using familiar syntax while also expressing complicated relationships between the different ontologies which are stored. This addresses one of the key problems in storing metadata for building systems, which is that it impossible to pre-define both all of the types of metadata needed to describe the building system, and the ways of querying it.

4.3 Historian

Sensors and actuators in buildings have the potential to generate significant amounts of data. Currently, not much use is made of this data because it is difficult to access and the tools needed to do so are expensive. Furthermore, it is sometimes desirable to place the stored data close to the sensors and actuators generating the data, so it will still be accessible in case of network partition and so that local control algorithms can run on stored data without needing access to a wide area network. This suggests a solution which provides good single-node performance to avoid the need for excessive infrastructure located in each building.

We have built a prototype of a next-generation historian providing three key services:

1. Efficient storage of time-series data,
2. High-speed importing and scanning of this data, and
3. Optimized cleaning and resampling of data.

The historian uses Berkeley DB for on-disk page management and transactional updates, and stores data in dynamically-sized buckets containing up to a day's worth of data. We apply a compression algorithm to the data which first packs deltas between neighboring timestamps and readings into a variable-length code using Google Protobufs, and then applies a Huffman-tree compression to the result. This gives good compression in many common cases; for instance when the readings are integer valued and change by only a small amount between successive readings. Clients access the readings using a Python module (implemented in c) which makes parallel requests on the data store in order to maximize throughput when accessing a large number of streams.

Before the data is sent back to users, the historian allows the user to apply a set of transformations to the data. The processing model for these transformations is inspired by unix pipes: simple units of functionality which are combined into complicated processing pipelines. Unlike pipes, which pass character streams between programs, we pass a slightly more complicated data structure: sets of time-series. Like pipes, the data stream pushed through the operators isn't framed; the operators must be able to operate on both single readings and large vectors of data. This is important for efficiency because the data consists of a large number of very small records; many operators can be implemented quite efficiently if they are given a large chunk of data to work on.

At the start of query processing, the pipeline of operators is bound to the actual streams to be processed. Each operator in the pipeline inspects the output from the previous operator, and generates a set of output streams which are passed to the next operator. Importantly, these intermediate products are named in the same `uuid` namespace as the original streams; this allows us to store the output of operators as new streams in the historical database and in the directory.

Table **??** contains some of the operators currently implemented in the system. Using simple combinations of these, users are able to perform common tasks like interpolating time-stamps, removing sections with missing data, and adding together several streams. Extending the set of operators is simple since we provide support for wrapping arbitrary python functions which operate on vector data; in particular, we have imported most of the `numpy` numerical library automatically.

```
CREATE TABLE data (
  INT streamid ,
  TIMESTAMP WITH TIME ZONE timestamp ,
  FLOAT value
);
```

4.4 Transaction Manager

5 Demand Ventilation Control Process

One example of a control process running on our system is a demand ventilation application. Building codes require a rate of fresh air ventilation per room based on occupancy and room size [**?**, **?**]. Keeping ventilation rates at this required minimum is highly desirable for energy savings since it reduces fan power and the need for air conditioning. However, this is difficult to do in traditional building control systems because separate control loops are in charge of varying the fresh air intake into the building, controlling the per-room airflow and detecting occupants. Reliability and consistency is crucial - even if the network goes down and the control process is disconnected, room airflow should meet the required minimums for occupant safety and comfort.

We accomplish this with transactions and leases. Below is the pseudocode of our application built on top of our control architecture.

```
Every 10 min:
  outside_temp=`SELECT data BEFORE NOW LIMIT 1
    WHERE Metadata/Type = "Outside Air Temp"`
  avg_room_temp=`SELECT mean(data, axis="streams")
    BEFORE NOW LIMIT 1
    WHERE Metadata/Type = "Room Temp"`

  Solve for optimal fresh air intake
  Solve for optimal supply air temperature

  BEGIN XACT
  LEASE 10 min
    Set fresh air intake damper
    for all rooms:
      Set room airflow
  UNDO
    Set room airflow to conservative
      minimum
  END
```

5.1 Principles

- Communicate with protocols, not APIs
- Scale like Internet services

- Make trust relationships in shared computation explicit
- Multi-site cooperation without setup
- Name the computation and the data together
- Failure domains
- Scheduling: priorities, interruptible, unit of atomicity
- Security
- Isolation: access to building resources. points, system/spatial, model-predictive
- Fault tolerance: failure domains, interaction across interfaces
- Hardware abstraction: multi-level interfaces, building on sMAP, hardware probing to discover relations

5.2 Components of a Resilient Cyber-Physical System

- Control and actuator sources
  - Lots of diversity: hide in REST
  - Tag at source
  - Identify and name with a uuid
- Stream processing infrastructure
  - Main primitive is the time-series: progression of (time, value) tuples
  - Name all time-series; name derivatives by the series of computation steps you took.
  - Use database techniques and manual hints to decide what to
  - Some inputs are from the raw sensor plane
  - Secondary sources of republished data sent to control loops
  - Maintain distillates $\rightarrow$ by pushing data through operator graph $\rightarrow$ new streams (repeat)
  - Feed historian.
  - Real-time analyses, detectors, and alarms.
- Application runtime containers
  - Connect to inputs (from stream processing)
  - Provide scheduling (between loops)
  - Make implementation decisions / Compile to different implementations: the story of three PIDs. Level of semantic understanding?
  - Provide transactional access to control points.
- Transactional access
  - Transaction: set of instructions to implement on the CPS
    - example: replace this PID loop with that loop
    - example: change schedule
    - example: hold that vent wide open for ten minutes
    - example: adjust a setpoint
  - Revert blocks – app-specific rollback code
  - Based on timeouts or triggered in the transaction
  - Provided to controllers in all tiers
- Distributed execution
  - Execute over front end PLCs, computing devices in the building, the Internet (three domains).
  - Communicate with other tiers and services, connect with broker for location info
  - Allow dependency/failure analysis
  - Allow information flow analysis
  - Certain things are bound to physical location
  - Discover native/end-user logic; representation of the computation being performed?
  - Responsible for application container placement, migration, failover/faildown
- Core services
  - Historian
  - Metadata
  - Stream processing
  - Authentication
  - Transaction manager
  - "Service broker"
- The semantics of time when processing
  - wall clock time (for sensor data)
  - processing timeouts in boxes
  - real time (deadlines)