UNIVERSITY OF CALIFORNIA, SAN DIEGO

Toward a Sensor-Actuation Software Platform

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

in

Computer Science

by

Kaisen Lin

Committee in charge:

Rajesh K. Gupta, Chair
David C. Chu
William S. Hodgkiss
Ali U. Irturk
Ryan C. Kastner
Geoffrey M. Voelker

2012

The dissertation of Kaisen Lin is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

_____
Chair

University of California, San Diego

2012

DEDICATION

To YOU, the reader.

EPIGRAPH

Luck is what happens
when preparation meets opportunity.
—Seneca

Clear eyes. Full hearts.
Can't lose.
—Eric Taylor

You can kill a man but you can't kill what he stands for.
Not unless you first break his spirit. That's a beautiful thing to see.
—Cigarette Smoking Man

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Philip Levis and David Chu for research assistance and guidance throughout the early phases of my graduate school career. Without them, I would surely not be where I am today.

I would also like to thank Roy Liu, Sebastian Becerra-licha, Jim Hong, and Taurin Tan-atichat for their technical and moral support during the dark ages of my graduate school career and Ted Lai, Chris Lei, Maulin Patel, Rach Liu, and Brina Lee for expanding my horizons on life, liberty, and the pursuit of happiness during the enlightenment. I would also like to thank all the tutors who helped during my times as a TA. It sure made handling 100+ student classes much more manageable. Thanks, also, to members of my thesis committee and members of the Microelectronic Embeddeded Systems Laboratory (MESL).

Finally, I would like to thank Julie Conner, Nadyne Nawar, Yosen Lin, Rick Ord, and Joel Coburn for helping me push the right buttons when navigating lab, department, and university bureacracy.

Chapter 2, in part, has been submitted for publication and may appear as Arrowhead: Coordinating Actuation Roles Through a Sensor-Actuation Software Platform. Kaisen Lin, Ted Tsung-te Lai, David Chu, Hao-hua Chu, Rajesh Gupta. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in Data Discovery and Dissemination with DIP. Kaisen Lin, Philip Levis. In IPSN 2008: 7th International Conference on Information Processing in Sensor Networks, pages 433-444, Washington, DC, USA 2008, IEEE Computer Society. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in Gibraltar: Exposing Hardware Devices to Web Pages using AJAX. Kaisen Lin, David Chu, James Mickens, Li Zhuang, Feng Zhao, Jian Qiu. In WebApps 2012: 3rd USENIX Conference on Web Application Development, Boston, MA, USA, 2012. The dissertation author was the primary investigator and author of this paper.

VITA

2006	B. A. in Computer Science, University of California, Berkeley
2009	M. S. in Computer Science, University of California, San Diego
2012	Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS

Kaisen Lin, Ted Tsung-te Lai, David Chu, Hao-hua Chu, Rajesh Gupta, "Arrowhead: Coordinating Actuation Roles Through a Sensor-Actuation Software Platform", Under Submission.

Kaisen Lin, David Chu, James Mickens, Li Zhuang, Feng Zhao, Jian Qiu, "Gibraltar: Exposing Hardware Devices to Web Pages using AJAX", *Proceedings of 3rd USENIX Conference on Web Application Development (WebApps)*, 2012.

Rahul Balani, Kaisen Lin, Lucas Wanner, Jonathan Friedman, Mani Srivastava, Rajesh Gupta, "Programming Support for Distributed Optimization and Control in Cyber-Physical Systems", *Proceedings of 2nd International Conference on Cyber-Physical Systems (ICCPS)*, 2011.

Kaisen Lin, Aman Kansal, Dimitrios Lymberopoulous, Feng Zhao, "Energy-accuracy Trade-off for Continuous Mobile Device Location", *Proceedings of 8th International Conference on Mobile Systems, Applications, and Service (MobiSys)*, 2010.

Kaisen Lin, Rajesh Gupta, "Towards Automated Building Management through Cooperative Sensor-actuator Networks", *Proceedings of 2nd Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

Kaisen Lin, Philip Levis, "Data Discovery and Dissemination with DIP", *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN)*, 2008.

David Chu, Kaisen Lin, Alex Linares, Giang Ngyuen, Joseph M. Hellerstein, "sdlib: A Sensor Network Data and Communications Library for Rapid and Robust Application Development", *Proceedings of the 5th International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, 2006.

ABSTRACT OF THE DISSERTATION

Toward a Sensor-Actuation Software Platform

by

Kaisen Lin

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Rajesh K. Gupta, Chair

Control systems are critical in many aspects of human life and are used in both large infrastructure and every-day appliances. Examples include factory control for chemical processes, HVAC control in buildings, cruise control in automobiles, spin cycles in washing machines, and even smart cooking in microwaves. However, large scale control of physical environments has largely been done with SCADA systems, which are only practical for large enterprises and often consist of monolithic systems designed for specific functions HVAC or factory automation.

The past decade has seen the emergence of wireless sensor networks being deployed to monitor all kinds of environments including buildings, waterpipes, cities, and wilderness areas. These sensor networks offer cheap monitoring capabilities without the need for a large amount of fixed infrastructure. Recent work in

some of these areas has augmented the wireless sensor network environment with control, particularly in homes and buildings.

This dissertation focuses on combining control systems with wireless sensor networks to enable commodity sensor-actuator networks that can be incrementally deployed in any environment. These sensor-actuator networks can accommodate not just a single sensor-actuation application, but a set of them. An important challenge here is how such a group of sensors and actuators are integrated in a sensor network with commodity hardware and programmed by the end user to achieve embedded control objectives. This dissertation specifically addresses the latter – the programming – problem. We show methods and build tools that enable an end user to write sensor-actuator network control applications and address challenges specific to incorporation of actuation in modern sensor networks.

Chapter 1

Introduction

1.1 Control Origins

Automation has been an ambition of man since the antiquities. The Romans were able to successfully measure time by carefully regulating the flow of water[1], and were also the first to prototype a steam engine. However, it wasn't until the end of the Industrial Revolution that James Maxwell published a paper on "On Governors" and launched the field of modern control theory, which led to much of the automation technology we see today.

Although control theory was first widely used to optimize the amount of steam for regulating a steam engine, it is now used in a variety of applications including process control in factories, cruise control in automobiles and airplanes, and even water regulation in washing machines. A practical view of embedded control is that it seeks to steer the values of controllable input variables to a system by observing timing evolution of system output variables. For instance, consider the cruise control in a car where the input variable is the engine throttle and the output variable is the speed of the vehicle. Control theory falls into two main categories: closed-loop and open-loop control. Closed-loop control involves continuously sensing an output variable as input variables are being adjusted. An example of this is a home thermostat which turns on a heater if the sensed

[1]Known as a water clock.

temperature is below a certain threshold. Open-loop control, in contrast, does not have such feedback and makes decisions solely based on a mathematical model of how input/output variables relate. In practice, a combination of both closed-loop and open-loop is used.

While control theory focuses on the decision of which input variables to manipulate in order to have desired effects on output variables, a control system is required to handle the physical-world devices that represent the variables. In the cruise control example, a control system manages the fuel injection for an engine for controlling throttle, and uses the spedometer for measuring speed. Control systems are typically implemented with computers, embedded systems, and other electronic devices. An example of a large industrial control system is a Supervisory Control and Data Acquisition (SCADA) system. SCADA systems include a human-machine interface to give human operators visibility and control into the system, while programmable logic controllers (PLC) are used to control individual devices. A PLC is a specialized rugged computer with many input/output ports designed for control applications. We examine these further for insights into how embedded control systems can be devised to leverage advances in commodity sensing, networking devices.

1.2 Prior Work

Sensor-actuation can be divided into three different levels. At the application domain level, these focus on sensor-actuation solutions specific to deployment scenarios such as camera control for tracking and active structural monitoring. At a slightly more generic level are infrastructure-level systems. These differ from application domain solutions by creating a system to address a class of control problems such as building HVAC, pipeline control, or robotics. Finally, the solutions in the most general level address any kind of sensor-actuation system and encompass both control theory and control systems. These solutions have been the focus of the sensor networking and cyber-physical systems research community.

1.2.1 Application-specific Solutions

Specific application domains devise their own methods for dealing with sensor-actuator systems. As an example, pan-tilt-zoom (PTZ) applications are designed for people tracking [CDBF04] and/or surveillance [KKP+06]. For people tracking, a set of PTZ cameras are used to track multiple people. Unlike tracking a single person, more complex algorithms are necessary to identify and prioritize the people being tracked because adjusting a PTZ camera has a physical delay from determination of its setting to its effect on the camera. One example policy is to prioritize detecting people that have not yet been identified by the camera before people that have already been detected. Surveillance applications with multiple PTZ cameras have similar requirements. In PTZ actuation applications, control of the actual camera is usually done through a device specific interface, such as through a web browser [Axi]. The logic of the application is written in the developers preferred language (e.g. MATLAB) and then special code interfaces with the device. in these applications, there are no actuation externalities. When a PTZ camera adjusts, the only important changes are the sensed image and the objects detected in the image.

Another application example is structural health monitoring via active sensing piezoelectric patches. In this application, an ultrasonic wave is sent through a structure and then sensed to determine if the integrity of the structural element has been compromised. Based on the sensor data, more ultrasonic waves are used to localize the damage [WSA01, FT10]. This technique is even done to diagnose the integrity of the sensor-actuator infrastructure itself [OPFF09]. Wave emission is done via a transducer on that is controlled from a microcontroller, and the bulk of the signal processing is done offline in MATLAB after data has been collected. Similar to the camera tracking applications, there are no significant externalities. The piezoelectric devices only affect the structure for a limited amount of time as the wave is propagating. However, many deployments for other applications have to control multiple actuation devices that affect multiple sensors. These situations require more sophisticated modeling and programming methods. Application-specific solutions are also not suitable for adapting for radically other

deployments.

1.2.2 Infrastructure-level Solutions

Infrastructure solutions are designed to address many actuators and sensors simultaneously. An HVAC system is one major example of an infrastructure-level system. One of the goals of HVAC systems is to provide occupant comfort while minimizing energy usage [Sal05]. Programming is done using proprietary software (e.g. from Johnson Controls), which allows building administrators to create setpoints on certain variables. Achieving these setpoints is done via a proportional-integrated-derivative (PID) controller, that is directly related to a variable in that zone. For example, a temperature setpoint can be set for a specific room in a building, which will indirectly control dampers and fans that affect that room. HVAC is usually done in a hierarchical nature with a central plant working in concert with buildings and individual thermal zones. Because of the hierarchical nature, an actuator change in the central plant affects all thermal zones in all buildings, thus complicating actuation schemes and requiring SCADA system sophistication (e.g. multi-level control with dependencies).

Another example of an infrastructure system is gas pipelines. Gas pipelines must travel long distances to carry natural gas from the source to household consumers. The goal of gas pipeline control is to satisfy demand to all customers even during peak times, while minimizing the costs of processing, transporting, and storing natural gas. Gas pipeline infrastructure is composed of wide interstate lines that connect to narrower intrastate lines with compressor stations and storage sites along the way [Ene07]. Compressor stations propel the natural gas through the pipelines while storage sites are used to supply gas to consumers during times of peak and bursty demand. Valves are installed to redirect flow when necessary. Various pressure sensors are also installed along the pipeline to monitor pipeline flows. For example, a sudden pressure drop in a pipe can indicate a rupture meaning the compressor stations should stop propelling gas through the pipeline network and relevant valves should be closed. However, shutting down one section of the pipeline necessarily affects all sections downstream from it, which

may lead to service delivery problems. There are also many other building and infrastructure applications such as control of factories [JM86], nuclear power plants [MRVB00], water pipelines [LCL+12], and sewage pipelines [MLE08]. However, they all share similar properties in that a single change in part of the system has effects elsewhere. A SCADA system seeks to capture important dependencies as described next.

SCADA Systems

A Supervisory-Control and Data Acquisition System or SCADA system, designed primarily for industrial control systems, is a computer control system that is typically composed of a human interface component and a low-level actuator control component [US 06]. The human interface component provides alert visualization, data analysis, and control semantics to a human operator. These interfaces are particularly important for off-normal conditions. For example, if a power generator fails, the computer system may automatically spread the load to multiple backup generators, but a human is ultimately responsible for bringing the failed generator back online. The control component of a SCADA system is responsible for low-level control and is often composed of a set of programmable logic controllers (PLC). They are used because of their low latency and high reliability. SCADA systems are used for HVAC because they allow simultaneous control of devices such as dampers, fans, and water boilers.

Robotics

Research in robotics deal with similar problems related to sensing and control as well. Instead of controlling infrastructure systems, they must control an autonomous vehicle to perform multiple tasks simultaneously. The actuation and sensing is mobile instead of fixed. A common approach to control a robot is also hierarchical [Sar83]. At a high level, a centralized component has summary information of the whole robot and can create tasks for specific modules. The modules can include aspects such as movement (e.g. control of wheels or legs) or motion (e.g. control of arms). The lowest level directly controls actuation and has knowl-

edge of the specific interfaces and/or physical models. An example of this is a self-driving vehicle [TMD+07]. At the highest level is a path planner, which analyzes all the sensory terrain data available to the vehicle, and sends information to the steering and throttle/brake modules. These modules directly interface with the vehicle control to steer and throttle the car. SCADA and robotics systems are both well suited for their specific application domains, but they are both designed for expert users. For example, home automation tasks can clearly be done with a full SCADA deployment, but its high cost and technical know-how requirement make it impractical for individual consumers.

1.2.3 Cyber-Physical Solutions

The most general solution is one that assumes nothing about what kinds of sensors and actuation devices are available and the relations of what the sensors and actuators are. The research community has devised numerous methods, abstractions, and languages for programming sensor networks. Mottola and Picco give an exhaustive survey of such techniques [MP11]. Techniques range from abstracting networks, mobility, space, or time, but unfortunately, very few programming techniques deal with actuation in a general way. Deshpande et al. describe a system for HVAC systems, and in theory can be used for any deployment, but no results have yet been published [DGM05]. The bulk of the work in cyber-physical systems has focused on programmability and modeling. Automatic modeling and simple programmability are important in order to build a solution that is usable by more than domain experts.

Programming

Programming abstractions for cyber-physical system involves giving abstractions for dealing with the physical environment. This can either be for a specific operation, or for the application solution as a whole. An example of an abstraction for a specific operation is Hotline. Hotline [BLW+11] gives a distributed shared memory abstraction for addressing physical variables. Applications can acquire a distributed lock to a shared resource (i.e. camera) and have mutually

exclusive access to it for a certain duration. Work on computation platforms has also recently emerged, particularly for smart grids [TGW11]. This area of work is still actively developing.

The other approach is to use a macroprogramming solution [KGMG07, HSH+08, HAAIKR11, VHXS10]. Macroprogramming solutions abstract away the networking, distributed, and hardware access of a deployment to allow developers to focus on the applications themselves. An example macroprogramming solution is Macrolab, which provides a MATLAB-like vector abstraction for applications that is converted into deployment-specific code. Accessing any actuators is done with special native code. sMAP [DHJT+10], in contrast, gives a web programming abstraction where all sensor data and actuation is done via a RESTful interface. TinyOS [t2] and IPv6 enable the abstractions on the embedded nodes themselves. Although these techniques provide simplifying abstractions for developers, their abstractions for actuator control are minimal.

Modeling for Embedded Control

Control systems as increasingly modeled using hybrid modeling methods such as hybrid or timed automata spanning continuous and discrete system variables. The physics underlying the variables is an important part of such models and in cases which such physical models are overly complex or even non-existent, system architects resort to system identification techniques [Lju08]. Here the designers seek to construct models of dynamical systems through empirical measurements. These models can be black box or white box. A white box model constructs a building model using first principles. A blackbox model, on the other hand, uses only empirically measured data to construct the model. Domain experts use both methods for creating the model. They perform actuation experiments using first principles, but record empirical data, which can be used in a real deployment. This allows more complex models to be created, such as model predictive controllers [Nik01].

1.3 Emergence of Wireless Sensor Devices and Networks

Control systems have predominantly been used with large infrastructure deployments such as building, bridges, and pipes because proper monitoring and control is critical for safety and operational efficiency. The importance of these systems makes the investments well worth the costs of having to instrument data sensing, collection, and control devices. However, with the emergence of inexpensive monitoring, communications and networking devices, it is now possible to construct at scale sensor networks for deployment by individual users in their homes and at work.

For example, mobile phones are now capable of sensing location, sound, light, motion, and orientation. The emergence of wireless sensor networks (WSN) and the Internet of Things [Ash09] have further increased the amount of sensory information available. WSN applications have included localization [SLV11], water-pipe monitoring [LCL+12], search-and-rescue [HAM05], and earthquake detection [SSKM07]. These WSNs can be deployed cheaply without existing infrastructure and still collect data in real-time, enabling control systems to make timely decisions based on the data. The embedded computing platforms that have performed the sensing and communication have now also been used for control. Companies with products in this area include X10, Cisco Systems, and Crestron. However, innovation in control systems is far from over as important control scenarios become more pervasive and more personalized.

Given the wireless control and sensing are becoming cheaper, and able to be incrementally deployed with existing infrastructure, the natural question to ask is: How should all these disparate components interact with each other and take advantage of existing research and infrastructure?

1.4 Dissertation

In order to build such a commoditized control system through use of wireless sensors and actuators, there are several challenges that must be addressed. First, we need a way to bridge applications with a deployment. This involves converting sensor and actuator devices into variables that can be programmed against. Then we need to provide an application programming interface (API) that can access those variables in a safe way. The first third of this dissertation, Arrowhead, focuses on addressing this problem. It sets up the framework and focuses on how control policies can be written as software applications, how actuation devices are integrated, and how environmental models that apply ideas from control theory can be used in a deployment. Application goals are ultimately converted to actuation configurations that can perturb the environment in a desired way.

Once actuation configurations have been determined, relevant actuation devices need to receive the configuration information. We assume actuation devices will be networked in an ad-hoc wireless manner because this is the most convenient method for incrementally deploying sensor and actuation devices. Furthermore, these devices may be battery-powered and have noisy network connections. Thus we need a networking protocol to disseminate actuation commands quickly with minimal messaging costs. The second part of this dissertation, DIP, describes the design and evaluation of such a protocol. It broadcasts advertisement messages in a controlled and efficient way to determine to quickly detect and propagate new actuation commands across the network.

Lastly, mobile phones today have numerous sensing capabilities, as well as continuous wireless connectivity. This combined with sophisticated mobile web browsers creates a rich opportunity for the mobile phone to integrate with our sensor-actuation system as both a sensor data provider as well as application platform. The final part of this dissertation, Gibraltar, focuses on enabling control applications to be web applications while simultaneously giving them access to phone sensors that are typically not accessable to web applications.

Chapter 2

Coordinating Actuation Roles

2.1 Introduction

The purpose of sensing often includes its effect on actuation which can be user-assisted or automatic. Control automation has existed for centuries and has been used in applications such as regulating temperatures in furnaces and controlling water levels for keeping time. Example applications include commercial HVAC systems, pipelines, automobiles, airplanes, and even washing machines. Despite these advancements, control automation has always been a monolithic affair. Take for example, a home "thermostat" where a resident is responsible for installing and maintaining heating devices, setting the temperature goals of the environment, and determining how to use the devices to achieve that goal. While this is acceptable for a single person in a single home, it is much more complex if a fire code must also be followed, new windows are installed, more residents move into the home, etc. As a sensor-actuation deployment becomes more complex with more devices and environmental goals, it becomes more challenging to satisfy all the various policy requirements.

Despite recent advances in modeling, design and optimization of sensor networks in recent years, actuation remains a challenging problem. This is because unlike sensing, actuation is an intrusive action that seeks to change the underlying physical processes. As a consequence, while sensing tasks can be shared among multiple applications, actuation actions present coordination, conflict resolution

challenges among competing actions and/or their goals. When multiple applications are in play, sensors can be sampled at a rate which satisfies the requirements of applications with the highest sensing fidelity. Actuators, on the other hand, cannot always be virtualized nor finely time-shared because of the effect they have on the environment. While multiple sensing actions are independent, actuator actions can combine in non-trivial ways. Applications in sensor-actuator deployments not only sense data, but specify policy goals for the environment based on the data. For example, consider two straight-forward thermal control applications: one that wants to ensure temperature levels are sufficiently high for human comfort when a space is occupied, and one that wishes to minimize the amount of energy. Naive time sharing of these two applications may be ineffective if they continuously toggle the heater on/off thus cancelling their effects. Because of these complexities, a building application like this would likely be implemented monolithically to account for both human presence and time. It is true that monolithic applications are advantageous because it allows a whole environment to be modeled and optimized, but it comes at a cost. These applications require redesign when deployment environments change because they are so finely tuned and optimized for the current environment. Thus an ideal solution will separate the deployment into different roles such that performing the duties of one role does not require detailed knowledge of the others. The three roles are operator, developer, and environmental modeler. To illustrate the roles, consider a home thermostat scenario.

1. **Operator:** A resident who buys a new fan for their home should not need to know the precise temperature effects of the fan nor precisely know the temperature goals of the home. The operator is responsible for ensuring that the actuation devices are operational.

2. **Developer:** A guest who enters a home with a particular thermostat preference should not need to know what devices are in the home and how they affect the environment. It is the developer's responsibility to specify environmental policies as applications.

3. **Modeler:** A thermodynamic expert who understands how air flow affects

ventilation and temperature should not need to know what temperature range preferences are being used. The modeler's responsibility is to create an environmental model that relates how actuators affect sensor values, and thus help applications decide which actuators to use to in order to achieve a certain policy.

Each role must then eventually be merged together into a single deployment without sacrificing reliability or efficiency. For example, applications that specify temperatures must be done without causing a fire hazard. These issues also apply to other deployment scenarios. Oil, gas, water, and sewage networks alike all have multiple requirements that must be met simultaneously. Pressure in pipes must be high enough to satisfy service agreements, but not exceed certain safety limits, which might cause ruptures and ensuing explosions, especially in gas pipes. For water pipes, pressures must also be high enough to prevent backflow and water contamination in potable water sources. HVAC situations are similar in that occupants have certain thermostat preferences, regulators have various safety requirements, and operators have efficiency goals. All of these issues require developers to write applications carefully in order to not interfere with each other and deal with any sensor faults.

Existing solutions in sensor-actuation focus on individual roles. Examples include building a better model relating actuators and sensors [Nik01], providing a unified sensing API [DHJT+10], or providing coordination primitives among applications [BLW+11]. However, none of them addresses integrating these three roles into a complete deployment system. We propose a unified actuation abstraction and runtime that is implemented in a system called Arrowhead. Arrowhead has knowledge of the underlying device relations, and is intuitive specifying deployment scenarios. In Arrowhead, sensors and actuators in the environment are shared variables, and multiple independent applications can use them. Applications represent policies of how the actuators should be used, and can be executed in its own execution environment as long as it has network access to Arrowhead's API. These applications can operate cooperatively, and collectively emit a set of actuation configurations that can be sent directly to a control system or to a human

operator. We make three key contributions in Arrowhead.

- **Sensor-actuation Runtime Environment:** We design and implement a sensor-actuation runtime that separates the responsibility of the three actuation roles, yet allows them to work cooperatively in a deployment scenario. Part of supporting a multitude of actuation applications involves an API that supports a variety of programming languages and is compatible with existing infrastructure.

- **Formalizing Deployments:** We describe how to take two existing deployments from the literature, formalize them as Arrowhead deployments, and deconstruct their policy goals into multiple applications written in various languages and execution environments.

- **Model-based Applications**: We show that even without domain expertise, we can develop environmental models from device relations in home automation deployment scenarios with enough fidelity to enable thermostat applications that simply specify the environmental goals rather than specifying actuator usage.

In the remainder of this chapter, we first describe the terminology and design of Arrowhead, and how deployments, applications, environmental models all integrate together. Second, we demonstrate how to apply Arrowhead to real scenarios from the literature, and write applications against them. Third, we describe how we use our empirical model in home HVAC scenarios, and simulation results of a home thermostat application.

2.2 Architecture of Arrowhead

Arrowhead is based on the following assumptions. First, *environmental models are important* for the semantic information that is considered to be the eventual test of "correctness" and "intent". In other words, Arrowhead does not seek to validate these models, instead these are considered as a golden reference

model against which individual actions are devised. Environmental models allow applications to specify what they want at a high-level rather than directly control actuation devices at a low level. This maximizes the number of actuation options that can be used. For example, instead of describing individual actuation actions, a thermostat application only needs to describe temperature objectives. Second, *it is unlikely for all devices to be computer controlled.* Actuation deployments may often have legacy infrastructure which are difficult to retrofit for automation. In these cases, human action may be necessary. For example, applications that require heavy signal processing may want to use MATLAB because of the math libraries available. Third, *applications can be written and launched from any execution environment.* There are numerous programming environments that are preferred by developers, and these programs may have there own execution environments as well. Thus, Arrowhead and its applications should not be required to run on the same machine or any specific machine. Lastly, *an existing sensor network infrastructure is in place.* Unlike installing actuators that are automated, it is becoming straightforward to add off-the-shelf sensors to existing environments. Hnat et al. give one such example for homes [HSL+11].

In the remainder of this section, we first describe the terminology used in Arrowhead. Then we describe the system architecture of Arrowhead and how various applications interact with it before finally going into detail about each of the components.

2.2.1 Formulation and Terminology

Sensors and actuators are two "types" of variables that the operators must specify in a deployment. A sensor in Arrowhead is a variable that can represent a detectable physical-world variable, such as temperature and humidity. Sensor variables can also be virtual. That is, a sensor variable can represent values that are "inferred" from measured sensor data. An example of this situation is when a classifier is used to convert a set of raw accelerometer data into a human action [RDML05]. The programing model should provide appropriate type system for the virtual sensors.

Actuator variables in Arrowhead represent a device configuration that can affect the environment. An example of this is a setpoint for a thermostat, in which the range of values would be the setpoint range and the units would be celsius. It is possible for a single actuator to have multiple device configurables, such as a space heater with an attached variable-speed fan. In this case, each configurable would be its own actuator variable. Because actuators may also have discrete setpoints (e.g. ON/OFF), actuator units can also be represented as enumerations. Both sensor and actuator variables are read/write. Writing to a variable means an application is creating a setpoint on what it wants the value to be.

A *relation* between an actuator variable and a sensor variable means that changing the actuator configuration will affect the sensor value. It is a relation since a configuration can lead to multiple sensor values. These relations come from physical dependencies. For example, an increase in fan speed results in a lower temperature in a certain thermal zone. All the relations are used to help build the model. A *model* is used to help determine what actuator variables should be manipulated to achieve target sensor values. Conversely, it can also be used to predict the sensor variable outcome for a particular set of actuator setting values. Arrowhead does not require that the model follow any specific type of physical model. For example, an HVAC model can be a statistical lookup of measurements from the environment, or it can be a system of thermodynamic equations.

A *deployment* is composed of all the variables in the environment, the relations among the variables, and a model. The more sophisticated a Arrowhead model is, the more flexibility an application has in achieving certain setpoints.

2.2.2 System Architecture

Arrowhead has a distributed architecture and is composed of the components illustrated in Figure 2.1. The Arrowhead *API* is designed to support querying variables and creating setpoints. Applications written using the runtime do not require any deployment-specific programming language or language execution runtime, so developers can write applications in their preferred environment. The Arrowhead *server runtime* provides the API, stores the relevant sensor-actuator

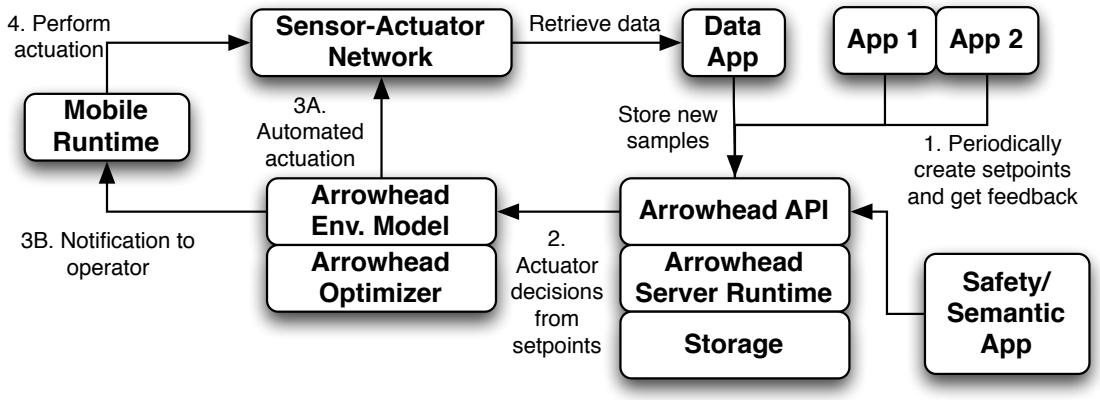


Figure 2.1: Arrowhead system architecture

data, and tracks which applications are accessing which sensor-actuator variables. It communicates with an environmental model to determine the best actuators to use. The *mobile runtime* is a background phone application which receives notifications from the server runtime, and forwards to a human operator. The runtime is designed with a human in the loop and bridge the actuation gap with non-automated infrastructure. The *environmental model* is important for allowing applications to focus on what the environment should be, rather than which specific devices to use. After it determines what device combinations will satisfy the environmental goals, an optimizer will pick the best one.

Applications periodically enter setpoints into the system through the API. The server runtime collects each setpoint and records the application associated with each setpoint. Then, at a fixed interval specified by the runtime, each setpoint is checked to make sure there are no conflicts. These setpoints are then passed into the environmental model to produce a set of actuator device configurations. Any actuation devices that cannot be automated, are sent to the mobile runtime which sends a notification to a human. The human performs the action and confirms with the mobile runtime, which notifies the server runtime that a device setting has changed. A data application periodically queries the sensor-actuator network and gives new sensor values to the server runtime.

2.2.3 Arrowhead API

There are three important application classes in Arrowhead that influence how the API is designed: standard, safety, and semantic. These application classesza are important because they are executed at specific times. Standard applications are those that can execute on 3rd party servers and have no special privileges. Thus, they are free to run any arbitrary code without any risk to Arrowhead. Safety applications ensure that regulations are being followed or that infrastructure settings are not pushed past extreme limits. These applications are always executed in conjunction with standard applications so the server runtime must execute these applications. Semantic applications are applications that convert information from a physical variable in the environment to a virtual variable. Virtual variables represent devices or environmental values that do not exist in a deployment. An example in HVAC scenarios is a virtual occupancy sensor, which is backed by an infrared motion sensor. Data applications that move data from the sensor-actuator network into Arrowhead storage are also in this category. Semantic applications are executed independently of safety and standard applications.

Arrowhead provides four sets of calls for applications to interact with Arrowhead and is summarized in Table 2.1. `QuerySensorValues` provides all the variable names, their current values, and their units, while `QueryActuatorValues` provides the equivalent values for actuators. Applications use these two queries to determine all the variables in the deployment. Sensors are separated from actuators so that applications know which variables can be directly controlled.

Applications write to variables using setpoints. A *setpoint* is formally defined as a range of values with a lower bound r_1 and upper bound r_2. The range allows applications with overlapping setpoint requirements to share use of a variable. For example, 200-500 lux is recommended for office environments [osh], humidity levels should be at 35-40% to prevent airborne sicknesses [US 91], and 30-60% for human comfort. If another application has any setpoints that overlap, then a setpoint can be achieved that satisfies both applications. In addition to the name of the variable and the range, `CreateSetpoint` requires a globally unique identifier. This identifier is used to group setpoints with a particular application as

Table 2.1: API Table

Call	**Return Values**	**Description**
QuerySensorValues()	[(time, name, val, units), ...]	Returns all sensor values and associated information.
QueryActuatorValues()	[(time, name, val, units), ...]	Returns all actuator values and associated information.
Begin(app)	OK/FAIL	Begins a set of setpoint creation calls.
CreateSetpoint(app, name, r_1, r_2)	OK/FAIL	Create a setpoint for variable.
Commit(app)	OK/FAIL	Ends a set of setpoint creation calls.
QuerySetpoints()	[(name, r_1, r_2), ...]	Query for all setpoints given in previous iteration.
QuerySetpointStatus(app)	string	Returns a string describing the status of setpoints for an application.
SetValue(name, val)	OK/FAIL	Sets a variable value.
QuerySolution()	[timestamp, (name, val), ...]	Query for device configuration solution.

well as provide a handle for feedback. Calls to `CreateSetpoint` must be wrapped with `Begin` and `Commit`. These calls are to ensure that the runtime does not begin acting on setpoints from an application until all setpoints have been specified.

While applications are always capable of receiving feedback from their setpoints by reading the sensor values, it is indirect and does not give insight during failure conditions. Thus Arrowhead also provides two extra calls that give feedback for applications to adjust their behavior. After applications have specified and committed their setpoints, they can query the status of them by using `QuerySetpointStatus` with their unique identifier. For example, if the environmental model does not have any information on how to control the setpoint of a particular sensor variable, then an error code can be returned. This allows applications to fallback to directly using actuators. Applications can also query the setpoints of other applications with `QuerySetpoints`. This can be used for lower priority applications that may want to yield setpoints to other applications. For example, an application that wishes to modify the thermostat variable to save energy may yield to a human comfort application. `SetValue` is for any application that needs to have direct access to Arrowhead variables. This is important for semantic applications. `QuerySolution` is designed for applications to determine what actuator configurations were actually used, and for troubleshooting applications to record actual device settings over time.

2.2.4 Server Runtime

The server runtime is responsible for storing sensor data, storing setpoints, servicing application calls, and interacting with the environmental model. The sensor data is collected from an existing sensor network and is important for two reasons: to train the model itself empirically and to determine what the current environmental conditions are. The server runtime operates on an *execution interval* for processing setpoints and giving feedback to applications.

At the beginning of each execution interval, the server runtime executes all the semantic applications to fill in variables, which do not exist in the physical environments. This ensures that virtual variables have the latest data in case the

semantic application is triggered at very low rates. Then the safety applications are executed to make sure certain setpoint requirements are met. Once those applications have been executed, the server runtime can accept setpoints from standard applications. After all applications have created setpoints for a particular interval, all setpoint conflicts are determined. A conflicting setpoint occurs when two applications attempt to set the same variable with mutually exclusive range values. If a conflict occurs, a failure notice will be added to both applications and all setpoints from the conflicting applications will be cancelled. The applications can later query to see why their setpoints failed. Once all valid setpoints have been determined. The server runtime sends the setpoints to the environmental model to determine which actuator device configurations should actually be used. These results are then automatically sent to an actuation network or a human operator.

2.2.5 Mobile Runtime

Deployments will often have actuation devices that must be controlled by a human operator via an ON/OFF switch, button setting, or a mechanical action. The mobile runtime is a background phone application designed to notify a trusted human operator who has the capability and privileges to actually perform actuation, not any arbitrary person. It works by using the `QuerySolution` call to determine which actuation devices need action. The mobile runtime, then, sends a notification to the operator saying which devices should be set to which configurations. After the operator performs these actions, he can confirm them in the application, which triggers a `SetValue` call back to the runtime server. Actuators that require human action are specially tagged and the mobile runtime knows what these are.

2.2.6 kNNR: A Simple Empirical Model

A key component of Arrowhead is the abstract environmental model which relates actuation devices with sensor values. This enables applications to focus on the actual policies themselves rather than how to achieve them. Our model, kNNR,

Table 2.2: Example progression of how raw data is converted into a set of snapshots. A_1, S_1, S_2 are variables with initial values of V_{A_1}, V_{S_1}, V_{S_2} respectively at time τ_0.

Order	**Raw Data**	**Snapshot**
		$< \tau_0, V_{A_1}, V_{S_1}, V_{S_2} >$
1	$< \tau_1, A_1, V_1 >$	$< \tau_1, V_1, V_{S_1}, V_{S_2} >$
2	$< \tau_2, S_1, V_2 >$	$< \tau_2, V_1, V_2, V_{S_2} >$
3	$< \tau_3, A_1, V_3 >$	$< \tau_3, V_3, V_2, V_{S_2} >$
4	$< \tau_4, S_2, V_4 >$	$< \tau_4, V_3, V_2, V_4 >$

uses the nearest neighbor model where first principles is applied by performing controlled actuation. That is, a human operator must decide which actuators to use to collect sample data. It is a model based on empirical data. When using the model, the input is a vector of setpoints that we wish to achieve, and the output is a vector of actuator configurations that will get us there. When training the model, an algorithm converts raw sensor data into data samples in the model.

Training

The details of the algorithm are outlined in 3 steps: convert raw sensor data R into grouped *snapshot* data σ, create *sample points* ρ from snapshot data, and compute *actuation settings*. These actuation settings are then given to an optimizer to decide the actuation settings to use.

$$R =< \tau, A|S, V_{A|S} > \tag{2.1}$$

The kNNR model is trained offline with historical data from all variables. A raw data value R is a tuple containing a timestamp τ, actuator A or sensor S, and the value V as shown in Equation 2.1. A snapshot is a vector of raw datapoints grouped together with a timestamp. The timestamp here represents an aggregate timestamp of all associated raw data. If data is collected in a synchronized manner, then grouping the datapoints is trivial, but this is not always possible because there may be multiple independent sensors.

$$\sigma =< \tau, \alpha, \beta > \tag{2.2}$$

Thus, our model groups data sequentially instead. A snapshot σ is described in Equation 2.2. τ is the timestamp of the snapshot, $\alpha =< V_{A_1}, V_{A_2}, \ldots >$ represents the set of actuators, and $\beta =< V_{S_1}, V_{S_2}, \ldots >$ represents the set of sensors. Table 2.2 illustrates an example of how raw data is converted into a chronological list of snapshots in kNNR. Each variable begins with a initial value, and values in the snapshot are updated as each new raw data value is received. The timestamp is also updated with each raw data value. The kNNR model does not necessarily create a vector based on all the variables. The modeler must specify what variables have relations.

Three kinds of relations exist among the data. A 1-to-1 relation means that an actuator can directly affect a sensor without any further dependencies. For example, a space heater in a thermal zone can solely adjust the temperature. It is certainly possible that many 1-to-1 relations exist for a single sensor. An N-to-1 relation means multiple actuators need to be used in order to affect a sensor. An example of this is a fan, which is dependent on a damper or window for new air. A fan, by itself, will only recirculate ambient air. Finally, a 1-to-N relation means that a single actuator can affect multiple variables. For example, using an HVAC air conditioning unit will often also filter the air, causing it to affect both temperature and particulates in the air. Our kNNR model currently only supports many independent 1-to-1 relations.

$$\rho =< \beta_{start}, \beta_{stop}, \alpha > \tag{2.3}$$

A sample ρ in the kNNR model is summarized in equation 2.3 and is comprised of an initial set of sensor values β_{start}, ending set of sensor values β_{stop}, and a set of actuator values α. The idea is that using a specific set of actuators will change the sensor values from β_{start} to β_{stop}. β_{start} is determined whenever α changes because this means an actuator's setting has changed. β_{stop} is determined based on a 30 minute timeout. This value is configurable based on how fast or slow-acting the actuators are.

The algorithm to create these samples from snapshots is done by first assuming all the snapshots are in chronological order and σ_i represents a snapshot where α has changed from the previous snapshot. Then for each σ_i, a sample is created by using α from σ_i, β_{start} as the β from σ_i and β_{stop} as the β from $\sigma_{i+30min}$ where $\sigma_{i+30min}$ is the snapshot 30 minutes ahead.

Usage

$$\Delta = \sum(\omega(||\beta_1 - \beta_i|| + ||\beta_2 - \beta_j||)) \tag{2.4}$$

Once all the samples have been created, the kNNR model takes as input starting sensor values β_i and target sensor values β_j, and determines the corresponding α. Intuitively, the best actuator setting is the one whose starting and ending sensor condition most closely matches the current conditions and setpoints respectively. We compute this using a distance metric. The kNNR distance metric, described in Equation 2.4, is a weighted sum of the differences between the setpoint vectors and the sample vectors for both the starting and ending conditions. A β_{start} and β_{stop} is extracted from each sample and the weight vector ω is used to help determine whether or not a change in value is significant. Once all distances have been computed, the kNNR algorithm selects 5 samples ρ with minimum distance Δ to the setpoint vector, and extracts each α to create a solution set. (5 can be adjusted based on how many candidate solutions are requested.)

After a solution set is determined, the kNNR model uses an optimizer to determine the final actuator setting to use. The optimizer can determine the costs of using certain actuators that were not visible in the model. An example optimizer for an HVAC scenario is one that operates the fastest or uses the least amount of energy. Of course, more sophisticated models can take into account these actuation costs before determining the solution set. We note that kNNR is not claimed to be optimal, indeed there are more predictive controllers [Nik01, DGM05]. However, our later results show that kNNR is sufficient for enabling model-based applications.

2.3 Evaluation - Applications

To demonstrate that Arrowhead can support real deployment scenarios and write real applications that take advantage of the Arrowhead runtime and API, we describe example uses from two application domains: pipeline monitoring and HVAC. In these application domains, we ask:

*Can we intuitively describe existing applications in Arrowhead that can achieve desired control?*Note, in this section we are not demonstrating that the applications will execute with the correct results in a real deployment because we do not have a real deployment. We only demonstrate its value for programming.

Our implementation of the Arrowhead server runtime is written as a Python daemon. It includes an XML-RPC server that exposes the Arrowhead API and writes to a database for handling setpoint information. A separate thread periodically extracts the setpoints for analysis. Our kNNR model is implemented as part of that thread. The mobile runtime is implemented in Android and simply acts as a background application that polls the XML-RPC server.

2.3.1 Pipeline Monitoring

We represent real deployment scenarios in Arrowhead from pipe monitoring [MLE08, MVO+05]. In the sewage pipe example, sewage enters the sewage network through a series of input sources: buildings, fields, and storm drains. These flows eventually merge into a *combined sewer line* which aggregates the sewage. The combined sewer line eventually drains into an *interceptor line*, which redirects sewage to a waste water treatment plant (WWTP). The WWTP treats the waste water before finally releasing it into a river or ocean. However, the WWTP and interceptor line can overflow during storm season, so a diversion structure is used to directly dump sewage from the combined sewer line into the river. In order to detect when sewage levels are nearing pipe capacity, a variety of pressure sensor nodes are placed along the combined sewage pipes and interceptor line. Diversion structures are placed along the interceptor line to redirect sewage out if an overflow occurs. Another example scenario is leak detection, which monitors pressure measurements

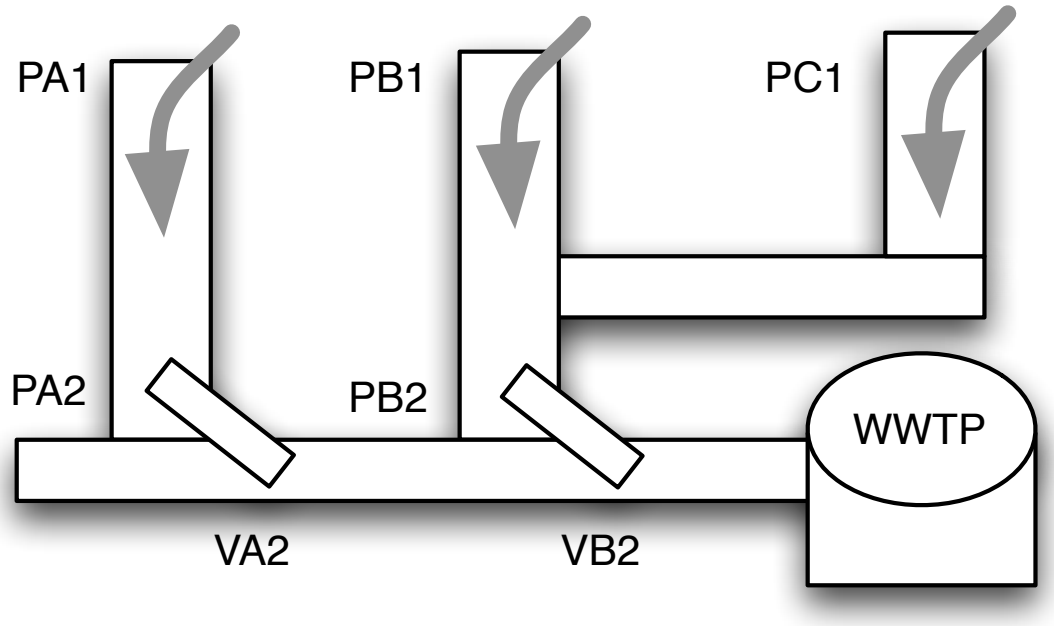


Figure 2.2: Deployment map for sewage deployment.

for a break event, analyzes a data window with the break event, and then estimates the break location based on wave timing analysis from pressure sensing along the pipes. Although designed for regular water pipes, it can also be used for sewage lines.

Figure 2.2 summarizes our reference sewage flow deployment and is based on the diagram from Montestruque and Lemmon [MLE08]. A full deployment would have over 20 diversion structures and over 100 sensor nodes, but the overall structure is similar. The variables that are relevant to Arrowhead for pipeline monitoring are the pressure sensor nodes and the actuator nodes for controlling the diversion structures. Each pressure node (measured in psi) can be represented as a sensor variable, PAX, where A represents a sewage segment, and X represents an identifying number. Each diversion structure valve is an actuator variable labeled VA where A represents the end location of the sewage segment. The units in this case is a percentage value between 0-100% representing how "open" the valve is. A fully open valve redirects all flow into the interceptor line. Relay and gateway nodes are also placed nearby to route sensor data to collection servers

for visualization, but they are not part of the actuation system, so we do not account for them in our formalization. However, these nodes are still important for collecting sensor data and we assume that a sensor network is in place for data collection. Thus, using only two kinds of variables, we can represent the sewage overflow problem.

Applications

We implement the sewage overflow application and leak detection application based on Figure 2.2 using algorithms from Montestruque and Lemmon [MLE08]. The goal of the sewage overflow application is to ensure that the none of the sewage lines overflow while maintaining maximum water treatment by the WWTP. First, the application must determine the flow capacity of each pipe including the interceptor pipe (constants), as well as acquire pressure measurements for each pipe path and convert them to flow measurements.

Then two cases arise: either a pipe segment is overflowing or the interceptor line itself is overflowing. If a pipe segment is overflowing, the algorithm uses a special feedback controller to control that diversion structure. If only the interceptor is overflowing, the algorithm sorts the capacity of each flow and the valve associated with it in descending cost order, where the cost represents the cost penalty of not diverting into the WWTP.

There are three conditions for controlling the valve when only the interceptor is overflowing. If the flow capacity of a pipe segment is less than the remaining available interceptor flow, then the valve is completely opened. If the flow capacity of a pipe segment is greater than the remaining interceptor flow, then the valve is opened to fill the remaining capacity. Otherwise, no remaining flow exists in the interceptor line and the valve must be closed. The algorithm can be translated into a Arrowhead application in the following way using the Python programming language and using the built-in XML-RPC client library to communicate with a Arrowhead runtime server:

```
s = RPC.QuerySensorValues()
RPC.Begin('CSO')
```

```
PA2 = SelectVal('PA2', s)
PB2 = SelectVal('PB2', s)
tuples = [('VA2', PresToFlow(PA2, ...), CFA2),
          ('VB2', PresToFlow(PB2, ...), CFB2)]
for name, FX, CFXY in tuples:
  if FX > CFXY:
    overflow := True
    SP = FeedbackControl(name, downstream(name))
    RPC.CreateSetpoint('CSO', name, SP, SP)

if overflow:
  RPC.Commit('CSO')
  return
tuples = [(CostVA2, CFA2, 'VA2'),
          (CostVB2, CFB2, 'VB2')]
tuples.sort()
tuples.reverse()
IFLOW = 0
for CostX, CFXY, name in tuples:
  if CFXY < IFCAP - IFLOW:
    IFLOW = IFLOW + CFXY
    RPC.CreateSetpoint('CSO', name, FlowToValve(CFXY),
                       FlowToValve(CFXY))
  else if CFXY > IFCAP - IFLOW:
    IFLOW = IFCAP
    RPC.CreateSetpoint('CSO', name,
                       FlowToValve(IFCAP - IFLOW),
                       FlowToValve(IFCAP - IFLOW))
  else:
    RPC.CreateSetpoint('CSO', name, 0, 0)
RPC.Commit('CSO')
```

This code implements the algorithm given above. *CFXY*, a constant, represents the flow capacity at XY. *CostVXY* is also a constant and represents the cost associated with using diversion structure XY. *IFLOW* is the current flow in the interceptor line, and *IFCAP* is the flow capacity of the interceptor line. The helper function *SelectVal* simply selects the value for a given Arrowhead variable. *FeedbackControl* is a separate feedback control algorithm that requires two physically adjacent nodes in a pipe segment and ensures the valve is configured to prevent an individual pipe segment from overflowing. To set valve settings, the code uses the

Arrowhead API to create setpoints directly on the actuators. *PresToFlow* converts pressure data into flow data, and *FlowToValve* converts flow data into a diversion structure setting. This setting is pre-computed.

In addition to this application, we can also write an additional leak detection application [MVO+05] on top of this deployment with minimal adjustment effort. (A full description of the algorithms can be found in that paper.) The only extra information that is necessary are several constants that can be determined before running any applications. The pressure measurements themselves already exist in the deployment as shown in Figure 2.2. For control, we assume there are valves on each pipe segment that can be closed to restrict flow, similar to diverson structures. The control aspect allows certain pipe segments to be closed off if a leak is detected. We give an example of this application written in GNU Octave [oct] that is based on PA1 in our sewage deployment scenario.

```
function answer = RPCCreateSetpoint(app, varname, low, high)
  shlcmd = sprintf("RPC CreateSetpoint %s %s %d %d",
   app, varname, low, high)
  [retcode, output] = system(shlcmd)
  answer = eval(output)
endfunction

global PressureData = [1, 2, 3]

function Driver()
  while(1)
    Sense()
    leak = DetectBreak()
    if(leak)
      [t1, t2] = AnalysisWindow(leak)
      [xb1, xb2] = LocateBreak(t1, t2)
      act = LocateActuator(xb1, xb2)
      RPCBegin('BREAK')
      RPCCreateSetpoint('BREAK', act, 0, 0)
      RPCCommit('BREAK')
    endif
    sleep(delay)
  endwhile
endfunction
```

```
function Sense()
  global PressureData
  svals = RPCSense()
  val = Select("PA1", svals)
  PressureData = [PressureData, val]
endfunction

function answer = DetectBreak()
  global PressureData
  [H, V] = Sensitivity threshold
  S = 0
  E = PressureData
  answer = 0
  for i=1:length(PressureData)
    S = max(S - E(i) - V, 0)
    if(S > H)
      answer = i
      break
    endif
  endfor
endfunction

function answer = LocateBreak(dt1, dt2)
  speed = Wave speed of pipe
  ethresh = Error threshold
  XM1, XM2 = distance between measurement point
                                and boundary
  if(abs(dt1 - (2 * XM1) / speed) < ethresh)
    xb2 = dt2 / dt1 * XM1
  endif
  if(abs(dt1 - (2 * XM2) / speed) < ethresh)
    xb1 = dt2 / dt1 * XM2
  endif
  # Other checks
  answer = [xb1, xb2]
endfunction
```

The algorithm is broken into distinct phases. The first phase continuously senses pressure readings until a break is detected (*BreakDetect*). Once the break is detected, the algorithm determines an analysis window (*AnalysisWindow*) of when the break occurred based on the length of the pipe. Then, using that window, the algorithm locates the break (*LocateBreak*) based on the length of the pipe, the

wave speed, and timing of when the wave is seen at the sensor. Finally, we add actuation control by shutting off the valve if there is a leak.

The Octave leak detection application has several complications over the previous HVAC and sewage appliations. First, because the Arrowhead API does not support querying historical data, we use a global state variable to keep track of it. In order to make API calls in Octave, we use the *system* function to invoke shell scripts that handle the XMLRPC. We use a *Driver* application that periodically executes the three different subroutines.

Repeated Applications

One benefit of using Arrowhead is that it allows applications to take advantage of the strengths of other programming languages and their libraries rather than requiring applications to be written in a proprietary system, such as enable reuse and maintenance of code.. This can be done either through parameterized functions, or through polymorphism if the language supports it. The leak detection application is a prime candidate for this, and each pipe segment with a pressure sensor can be considered an independent instantiation of the application. We can rewrite the driver and sense functions in the following way.

```
function Driver(sensor)
  while(1)
    Sense(sensor)
    leak = DetectBreak()
    if(leak)
      [t1, t2] = AnalysisWindow(leak)
      [xb1, xb2] = LocateBreak(t1, t2)
      act = LocateActuator(xb1, xb2)
      RPCBegin('BREAK')
      RPCCreateSetpoint('BREAK', act, 0, 0)
      RPCCommit('BREAK')
    endif
    sleep(delay)
  endwhile
endfunction

function Sense(sensor)
  global PressureData
```

```
  svals = RPCSense()
  val = Select(sensor, svals)
  PressureData = [PressureData, val]
endfunction

Driver('PA1')
```

The leak detection application can then be easily parameterized for all pipe segments through *Driver*. In fact, this leak detection code can be redeployed for any deployment as long as the variables and constants are known to the application.

Semantic Applications

As discussed before, semantic applications allow deployments to create more variables in the deployment than actually exist. It is not always possible to have every kind of sensor in a deployment due to cost. Having multiple kinds of physical sensors in the deployment is not cost effective, when one can be derived from the other. For example, the sewage overflow application really uses flow measurements, while the leak detection application uses pressure measurements. A better solution is to create semantic applications that perform this conversion, allowing other applications to be used on multiple deployments without adjusting for new variables. For a pipeline deployment, a flow rate can be computed from differential pressure changes using the Venturi effect. A function to compute the volumetric flow rate from pressure using that method is shown below.

```
import math
def PresToFlow(p1, p2, A1, A2, D):
  x = 2(p1 - p2)
  y = D(math.pow(A1/A2, 2) - 1)
  return A1 * math.sqrt(x/y)
```

The code requires two pressure sensors to compute the flow, but this is acceptable because two adjacent pipe segments will presumably have pressure sensors on both of them. p1 and p2 represent the pressure sensor readings, A1 and A2 represent the cross-sectional areas of the two pipes (constants), and D represents the density of the fluid (also constant). By having this semantic application

convert the data, applications now have the illusion that both sensors exist even though there may only be pressure sensors. Conversely, a deployment with only flow sensors can use the inverse semantic application to create pressure sensor data.

Manual Applications

We have largely been describing the sewage network when it enters underground drains. However, the sewage network can be extended into homes where sinks, toilets, and shower drains serve as the entry points of the sewage network. Because the occupants are ultimately in control of the water flow within the home, the occupants, as opposed to the munical sewage company, are responsible for any device actuation.

Consider the situation where a municipal sewage line becomes clogged. This can be detected with an application that determines if sewage flow has stopped for long periods of time. If this occurs, a backflow of sewage into the home could occur, resulting in flooding and costly repairs. A backflow prevention device can be installed near the house to redirect the sewage outside the home [Ass04]. This can conceptually be thought of as a diversion structure from the home side, as opposed to the WWTP side. Suppose we have a user-installed check valve near the home that should open if sewage flow has stopped. Then we can use the following code.

```
def Backflow(p1, p2, A1, A2, D, bpd):
  s = RPC.QuerySensorValues()
  P1 = SelectVal(p1, s)
  P2 = SelectVal(p2, s)
  F1 = PresToFlow(P1, P2, A1, A2, D)
  RPC.Begin('BACKFLOW')
  if F1 == 0:
    RPC.CreateSetpoint('BACKFLOW', bpd, 0, 0)
  else:
    RPC.CreateSetpoint('BACKFLOW', bpd, 1, 1)
  RPC.Commit('BACKFLOW')
```

In the code above, the parameters represent information necessary to convert the pressure to flow, as well as a backpressure device identifer. Its unit is open/close (1/0). If the flow for a particular pipe segment has stopped, then

the application attempts to open the backpressure device. Otherwise, it remains closed. Even though the backpressure device does not have any automated control, Arrowhead applications can use it like any other actuation variable. When Arrowhead produces a solution, a mobile runtime queries the solution and notifies the home owner that the backpressure device should be used. The home owner can then manually perform the actuation and confirm it in Arrowhead.

Model-based Applications

The sewage overflow problem is really about ensuring that the interceptor line into the WWTP does not overflow, and the algorithm from Montestruque and Lemmon [MLE08] is exactly about controlling the diversion structures to prevent this overflow. However, there already exists a natural relation between the diversion structure valve and the maximum amount of flow that can go through it into the interceptor line. Thus we can add a model relation between each flow sensor on the interceptor line and each actuation valve. This makes the oveflow application very simple to write, as shown below.

```
RPC.Begin('CSO')
RPC.CreateSetpoint('CSO', 'F-INT', 0, INT_FLOWCAP)
RPC.Commit('CSO')
```

The code uses a single setpoint on the flow of the interceptor line to ensure that it never goes past its flow capacity designated as *INT_FLOWCAP*. This application will only work with a model that is sophisticated enough to accurately account for the relation between diversion structures and pressure.

2.3.2 HVAC

To demonstrate that Arrowhead applies to multiple kinds of deployments, our other tangible example is with HVAC systems for a single building. At a high-level, modern building HVAC systems are divided into a hierarchy of three levels [Sal05]: the central plant, the building, and the thermal zone. The top-level is the central plant which provides heating and cooling elements to multiple buildings.

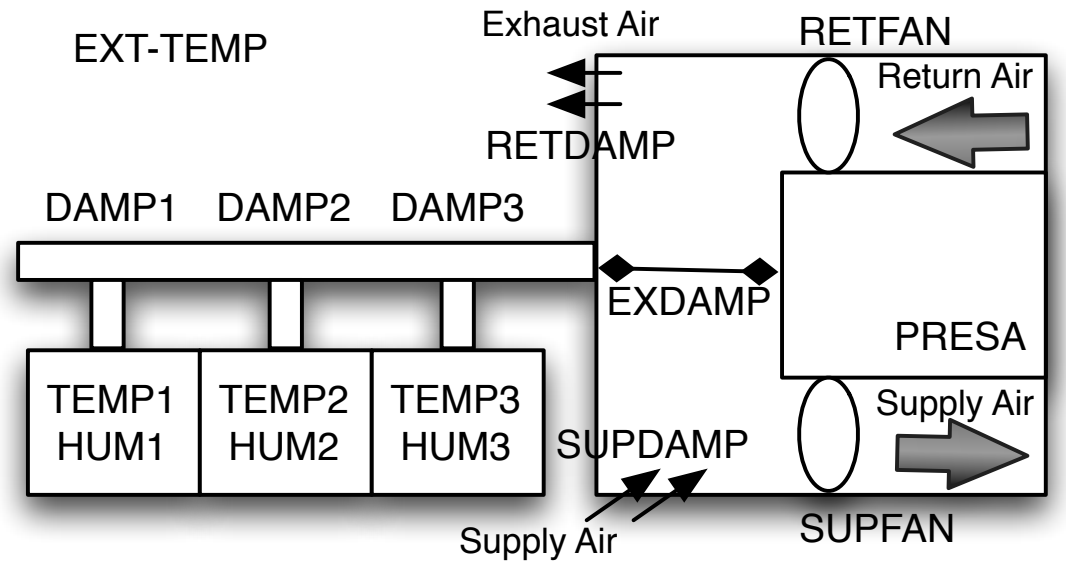


Figure 2.3: Deployment map for HVAC deployment.

Modern buildings typically use hydronics to provide thermal regulation due to its simplicity. This is done by using hot and chilled water as heat exchange mediums and transporting it among the buildings. Once the conditioned water reaches the building, building fans, dampers, and various air handlers distribute it throughout different thermal zones using air coils that transfer the heat between the water and the air.

Figure 2.3 shows an example of a deployment loosely based on information from Salsbury [Sal05] with the variables labeled. Each thermal zone has a temperature and humidity sensor (TEMPX and HUMX), and a damper that controls air flow (DAMPX). At the building level, a supply fan (SUPFAN) and return fan (RETFAN), combined with the return damper (RETDAMP), supply damper (SUPDAMP), and external damper (EXDAMP), regulate how much external and recycled air circulates in the building. There is also a pressure sensor (PRESA) to detect if air pressure in the ducts is high enough to cause stress and damage the air handling infrastructure. Finally, there is also an external temperature sensor (EXT-TEMP) to help optimize air flow. We only have one set of building level actuators and sensors in our example, but in reality, there are multiple sets of these devices depending on the size of the building.

The relations in a building can be far more complicated than a pipe network. Because of the multiple hierarchies in a building, there are relations between building actuators and sensors, and their thermal zone counterparts. In addition, sensors and actuators in adjacent thermal zones have spatial relations as well. As an example, the SUPFAN and RETFAN fans affect airflow in every thermal zone so there is a relation between those two building actuators and every sensor in every zone.

Applications

An example application suggested by Salsbury [Sal05] is the economizer. The goal of an economizer is to take advantage of both internal and external ambient conditions to condition a room environment rather than using devices that mechanically heat or cool the environment. We give Java code below that performs economization.

```
List<Object> params = new ArrayList<Object>();
List s = RPC.execute("QuerySensorValues", params);
List sp = RPC.execute("QuerySetpoints", params);
int EXT_T = SelectVal("EXT_TEMP", s);
int T = SelectVal("TEMP3", s);
int TS = SelectVal("TEMP3", sp);
params.add("ECON");
RPC.execute("Begin", params);
if((EXT_T < T && TS < T) ||
   (EXT_T > T && TS > T)) {
  RPC.execute("CreateSetpoint",
              new Object[]{"ECON", "EXDAMP",
                           new Integer(0),
                           new Integer(0)});
  RPC.execute("CreateSetpoint",
              new Object[]{"ECON", "SUPDAMP",
                           new Integer(100),
                           new Integer(100)});
  RPC.execute("CreateSetpoint",
              new Object[]{"ECON", "SUPFAN",
                           new Integer(100),
                           new Integer(100)});
}
```

```
/* Same for recycling internal air */
RPC.execute("Commit", params);
```

In the code above, the economizer uses TEMP3 as the reference temperature and is designed to cool a thermal zone if the outside temperature is cooler than the zone temperature and the setpoint is lower than the current temperature. Similarly, it will heat a thermal zone if the external temperature is higher than the zone temperature and the setpoint is higher than the current temperature. In these two cases, more external air is used to condition the thermal zone. There are also two other cases that condition a room by recirculating internal air. This Java application uses the Apache XML-RPC [apa] client library to make XML-RPC calls to Arrowhead.

We can also implement other applications in our HVAC deployment scenario. For example, occupancy-based HVAC control [ABG+11] is a standard application, which turns off HVAC components when a thermal zone is unoccupied. A simpler time-based setback HVAC control [Sal05] is also possible for deployments with no occupancy sensing. Suppose in our example deployment from Figure 2.3, we extend the scenario with sensing information from the UCSD deployment [ABG+11]. That is, each thermal zone has an infrared sensor variable, IRX. If a zone is determined not to be occupied by the infrared sensor, then the damper is closed, otherwise the damper is opened based on the setpoint to allow the proper amount of chilled air to flow into the zone. Deciding how wide to open the damper is done with a PID controller. A PID controller is a feedback control mechanism that determines how to use actuators based on direct sensor feedback. We give Python and Octave code for the occupancy-based control and PID controller respectively.

```
def Occupancy():
  s = RPC.QuerySensorValues()
  RPC.Begin('OCCUPANCY')
  IRX = SelectVal('IR1', s)
  if IRX:
    RPC.CreateSetpoint('OCCUPANCY', 'DAMP1', 0, 100)
  else:
    RPC.CreateSetpoint('OCCUPANCY', 'DAMP1', 0, 0)
```

```
  RPC.Commit('OCCUPANCY')

function PIDControl()
  svals = RPCSense()
  pval = Select("TEMP1", svals)
  P, I, D = Learned variables
  dval = f(P, I, D, pval)
  RPCBegin("PID")
  RPCCreateSetpoint("PID", "DAMP1", dval, dval)
  RPCCommit("PID")
endfunction
```

The code above illlustrates how a thermostat can be decomposed into smaller applications that run concurrently even with different languages. The occupancy application simply detects if zone 1 is occupied. If so, it allows the damper to be in any position. Otherwise, it closes the damper by setting it to 0%. The reason the range is left open when the zone is occupied is because it is not the occupancy application's responsbility to determine the exact setpoint. That responsibility is with the PID application, which senses the temperature as feedback and applies the proper damper settings. It is important to note that each individual application does not need to worry about the details of other applications.

Repeated Applications

Repeated applications can also be applied in this deployment. Unlike before where we simply used parameterized functions, if a language supports first-class functions, then an application generator can be used to create repeated applications. Below is an example of an application that generates HVAC occupancy applications.

```
def OccupancyHVAC(zone):
  def OccupancyHVACApp():
    s = RPC.QuerySensorValues()
    RPC.Begin('OCC' + zone)
    IRX = SelectVal('IR' + zone, s)
    if IRX:
      RPC.CreateSetpoint('OCC' + zone, 'DAMP' + zone, 0, 100)
```

```
    else:
      RPC.CreateSetpoint('OCC' + zone, 'DAMP' + zone, 0, 0)
    RPC.Commit('OCC' + zone)
  return OccupancyHVACApp

f = OccupancyHVAC('1')
g = OccupancyHVAC('2')
```

This application takes advantage of Python's inner function scoping rules to create functions that are parameterized on a thermal zone. A script can also be used to quickly generate applications for all thermal zones.

Semantic Applications

Semantic applications are also applicable in HVAC. Suppose in the occupancy HVAC application, instead of having an infrared sensor with a binary output, a camera is used that provides a stream of images for occupancy detection [ECPC11]. Below is an example of how to write an HVAC deployment with a camera (actuator CAMX) instead of infrared sensor.

```
s = RPC.QuerySensorValues()
image = SelectVal('CAM1', s)
if Classify(Image):
  RPC.SetValue('IR1', 1);
else:
  RPC.SetValue('IR1', 0);
```

In this example, *Classify* is an image processing routine that uses background subtraction. Thus each image only needs to be compared against a base reference image. If multiple applications attempt to set the same value, the behavior is undefined. This is usually not an issue because only trusted applications are allowed to use `SetValue`.

Manual Applications

An HVAC deployment often has many environment-affecting device such as space heaters, windows, and doors that are not connected in an automated way. With these devices, a human must manually perform the actuation and

notify Arrowhead that device settings have changed. As an example, consider a deployment with an economizer application that uses a window (WINX) instead of dampers. The window is also a percentage-open actuator variable.

```
List<Object> params = new ArrayList<Object>();
List s = RPC.execute("QuerySensorValues", params);
List sp = RPC.execute("QuerySetpoints", params);
int EXT_T = SelectVal("EXT_TEMP", s);
int T = SelectVal("TEMP3", s);
int TS = SelectVal("TEMP3", sp);
params.add("ECON")
RPC.execute("Begin", params);
if((EXT_T < T && TS < T) ||
   (EXT_T > T && TS > T)) {
  RPC.execute("CreateSetpoint",
              new Object[]{"ECON", "WIN3", 0, 0});
}
/* Same for recycling internal air */
RPC.execute("Commit", params);
```

In the Java code above, the application closes the window if the outside temperature is favorable. Alternatively, it could use a conversion function that determines the optimal position for the window to maintain a certain temperature. The WIN3 actuator is not actually automatable, and thus the mobile runtime will periodically call `QuerySolution` to determine if the window needs to be adjusted. If so, a notification is sent to the user. After the user performs actuation and acknowledges the mobile runtime, `SetValue` is called with the new window actuator setting.

Model-based Applications

In certain HVAC situations, applications are only concerned with what the final sensor environmental variables should be, not how to get there (via actuation). For example, applications that maintain government regulations such as those that set minimal lighting and ventilation rules can easily be done with model-based applications without having to worry about which actuators to use. The OSHA recommended indoor temperature range is $20°C$ to $23.5°C$ [osh]. Thus a very simple application for this application could be the following.

```
RPC.Begin('OSHA')
RPC.CreateSetpoint('OSHA', 'TEMP3', 20, 23.5)
RPC.Commit('OSHA')
```

Any application that wishes to set a temperature not within that range will encounter a conflict. The application can take further action if the desired temperature range is not satisfied for an extended amount of time.

2.4 Evaluation - Deployment

To demonstrate that model-based applications are possible in Arrowhead, we seek to construct an environmental model that does not require a high level of domain expertise, yet is still effective in enabling model-based applications. We use our kNNR model in two different deployments and use a basic thermostat application. It is important to note that Arrowhead can always use more complex models that are created by domain experts.

We deploy Arrowhead in a two-story single-family home, and a 2-bedroom apartment unit. Each room represents a thermal zone and is equipped with TelosB [PSC05] temperature, light, and humidity sensors, but the two deployments have different actuation devices and ambient conditions. The TelosBs run TinyOS [t2] and use CTP [GFJ+09] and DIP [LL08] for collecting sensor data and disseminating commands respectively. There are 12 Arrowhead variables in each deployment, as shown in Tables 2.3 and 2.4. TelosBs are also used to perform automated control for plug load devices through a control relay.

2.4.1 House Deployment

We instrumented nine different zones in a single-family two-story house with sensing capabilities, but only three of them had actuators deployed in them. Thermal zone 2 is a small room on the second floor with exposure to sunlight. It has an air filter, small fan, and a window. Thermal zone 5 is a large room on the first floor with minimal exposure to sunlight. It has a small fan. Thermal zone 7 is a large room on the first floor with exposure to sunlight. It has a large fan and a

Table 2.3: House deployment variables

Variable	**Units**	**Type**
Description		
TEMP2, TEMP5, TEMP7	C	Sensor
Temperature in thermal zone		
HUM2, HUM5, HUM7	RH%	Sensor
Relative humidity in thermal zone		
FAN2, FAN5, FAN7	On/Off	Actuator
Fan on/off in thermal zone		
AF2	On/Off	Actuator
Air filter on/off in thermal zone		
WINDOW2, WINDOW7	Open/Closed	Actuator
Window in thermal zone		

window. Our deployment is described in Arrowhead with the variables described in Table 2.3.

We continuously collected sensor data for 2 weeks and randomly adjusted actuation settings to record the impact of using different actuator combinations [1]. The first experiment we ran was to determine how impactful actuators could actually be in our deployment. We evaluate this with an occupancy-based thermostat application as shown in Section 2.3.2.

Figure 2.4 shows the results of various starting conditions on TEMP2, and a setpoint of $34°C$ and $30°C$. kNNR is capable of producing valid solutions for applications even when the starting conditions are outside of the model. For example, suppose our kNNR model only has data for a temperature range between X_1 and X_2. Then if an application creates a setpoint at X_2, but with a starting condition less than X_1, it will still yield the correct actuation solution. An example of this is seen in our deployment when the setpoint is $34°C$. For TEMP2 starting values below $30°C$, 60% of the solutions opted to use none of the actuators. Although not surprising, this is, indeed, the correct answer because there are no heating actuators in this deployment.

The results also show that kNNR will select "incorrect" actuation settings when no samples exist. This is seen when the setpoint is $30°C$ and the starting

[1] In a live deployment, this can be done over a long period of time over regular use scenarios.

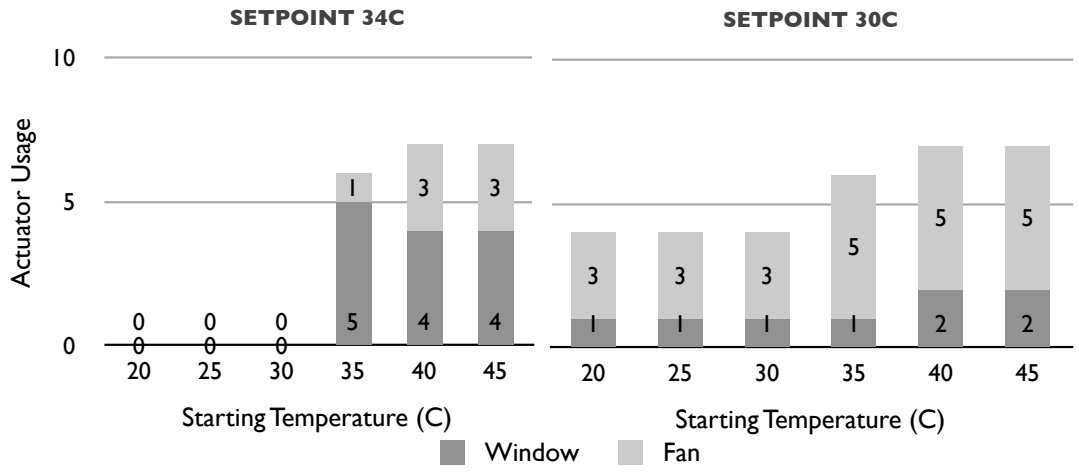


Figure 2.4: Actuation usage for an occupancy scenario in the house deployment with different starting values, and setpoints of 30C and 34C on TEMP2.

temperature is $20°C$. There are no heating actuators in our deployment, so using any device is intuitively incorrect. The reason our model chooses an incorrect solution is because the data in our model only has temperature changes that reach $30°C$ when the starting conditions are higher than $30°C$. Thus, the closest match is to use a cooling device.

This weakness is seen in Figure 2.5, which shows the range of different values for the three thermal zones. The min and max values represent the actual sensing range of the variables. The model min and max values represent the range that our kNNR model is able to use. Figure 2.5 shows that the kNNR model is capturing a narrower range of values than actually recorded by the sensor network. This issue stems from not accounting for natural sunlight that is affecting the environment. However, it can be inferred through a semantic application, and a new actuator variable (SUN). It does not matter that this actuator is not controllable by the system (neither automatically nor manually) because the samples would still accurately reflect the environment. Because kNNR is an empirical model, it is straightfoward to add samples and allow the thermostat application to take advantage of the new model.

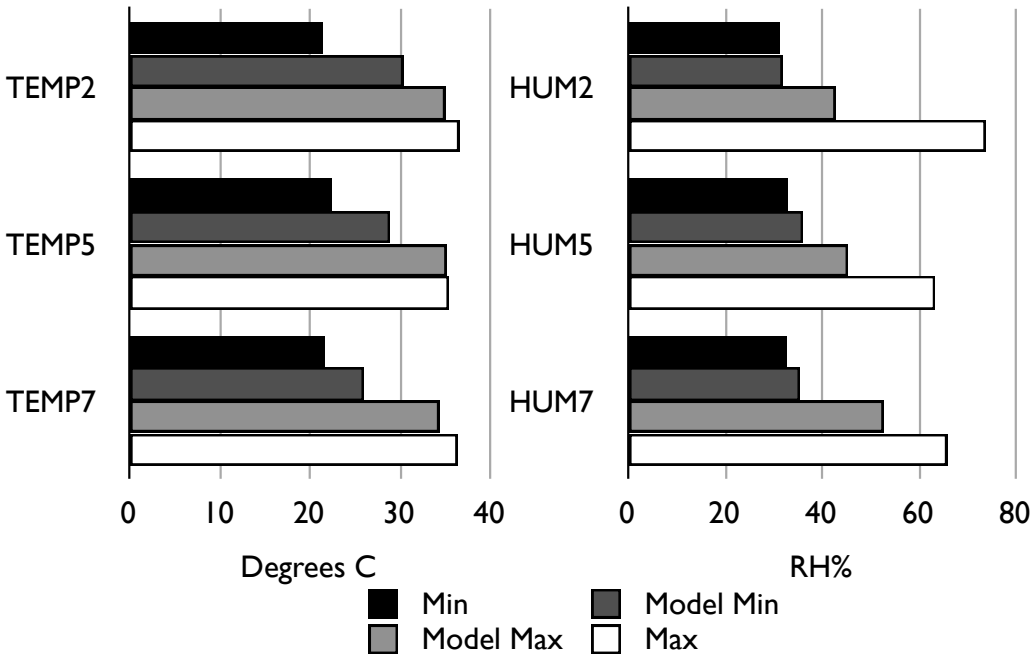


Figure 2.5: Range of sensor values for house deployment

Table 2.4: Apartment deployment variables

Variable	**Units**	**Type**
Description		
TEMP1, TEMP2, TEMP3	C	Sensor
Temperature in thermal zone		
HUM1, HUM2, HUM3	RH%	Sensor
Relative humidity in thermal zone		
FAN3	On/Off	Actuator
Fan in zone 3 (25.1W)		
AF3	On/Off	Actuator
Air filter in zone 3 (25.2W)		
WIN3	Open/Closed	Actuator
Window in zone 3		
SDD1	Open/Closed	Actuator
Sliding door in zone 1		
INF1	On/Off	Actuator
Infrared heater in zone 1 (482W)		
CEN	On/Off	Actuator
Central thermostat in apartment. 24.5$°C$ fixed setpoint.		

2.4.2 Apartment Deployment

We also deploy Arrowhead in a 2-bedroom apartment. This deployment is richer than the house deployment in that there are both heating and cooling elements. It is also in a smaller space, so the devices have more interactions. Table 2.4 summarizes the sensors and devices used. We have also included power measurements for relevant actuators. Similar to the previous deployment, we collected data over a two-week span and randomly used actuation devices to measure samples through kNNR. One major issue with the data samples is that the central heater actually cooled the air in several examples and the model picked up on this. This is because the fans start pushing airflow before the heating elements have time to actually heat the air. We resolved this issue by removing the samples that represent these edge conditions.

We also simulate our thermostat application in the apartment deployment, but with a setpoint of 23$°C$ and a starting temperature of 19$°C$ because these values are within our model for this deployment. We use two independent thermostat

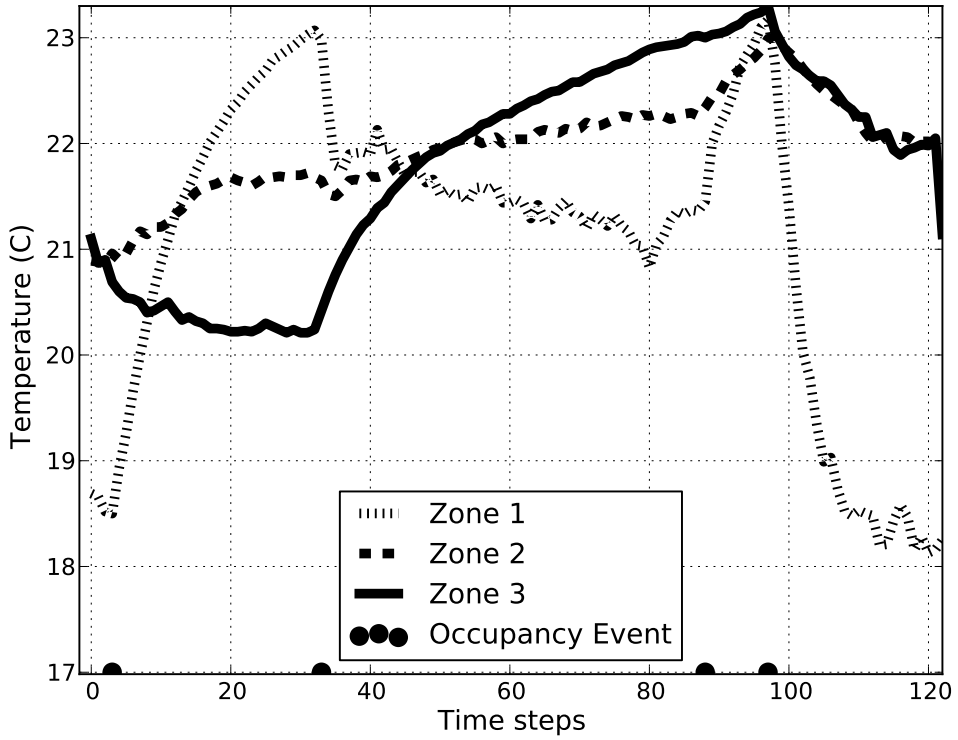


Figure 2.6: Temperature values of an occupancy scenario in the apartment deployment.

applications, one each for zones 1 and 3. Our scenario is to have a person begin in zone 1. After the setpoint is reached, the person moves from zone 1 to zone 3. This effectively creates a setpoint of $23°C$ in zone 3, and a setpoint of [0, 30] in zone 1, which means it does not matter what the temperature is in zone 1. After zone 3 has reached $23°C$, another person enters zone 1 so that both zones 1 and 3 have a setpoint of $23°C$. After that occurs, both occupants leave.

Figure 2.6 shows the results of running our apartment scenario with the kNNR model, as well as occupancy events where occupants enter or leave a zone. Each time step represents 30 seconds. The model chooses to use the infrared heater to increase the temperature in zone 1 and takes 14 minutes 33 seconds to accomplish this. After the user moves from zone 1 to zone 3, the model switches to using central heating to increase the temperature in zone 3 because there is no

local heating actuator there. It also decides to open the sliding deck door, which cools zone 3, but has no impact on application requirements. This takes 17 minutes 20 seconds. After a second person enters zone 1, the model need only close the sliding deck door to contain heat from the central thermostat. This process was considerably faster and took only 3 minutes 12 seconds. Finally, both people leave at the end, which turns off central heating.

To quantitatively evaluate our model-based thermostat, we look at power usage from this scenario. The total energy cost of using our model-based thermostat is aproximately 421kJ. In contrast, a naive solution would use central heating to achieve temperature goals for all zones. Central heating requires anywhere from 600W to 8000W+ to operate [mre], which means accounting for the energy cost just when the first person enters zone 1 would require at least 523kJ.

Our results from both deployments show that a useful programming environment that enables model-based applications can be built empirically without requiring advanced modeling from a domain expert. This allows application developers to quickly evaluate against real deployment scenarios. Domain experts can also later add more sophisticated models without requiring changes to applications.

2.5 Related work

Sensor-actuation is a cross-disciplinary topic and has been studied by a variety of people from both academia and industry. Our work touches primarily on programming, modeling, and how to integrate them for a deployment. Many deployments involve programming directly against a custom solution. Examples include camera tracking [CDBF04, KKP+06] or structural health monitoring [WSA01, FT10] solutions. However, we are more interested in structured approaches that can be reused. The primary examples of these include systems from SCADA or cyber-physical systems.

A SCADA system is a computer control system that is typically composed of a human interface component and a low-level actuator control component [US 06]. The human interface component provides visualization, data analysis, and control

semantics to a human operator. The control component of a SCADA system is responsible for low-level control and is often composed of a set of programmable logic controllers (PLC). There are also PID-based controllers that are used for localized control. For example, a zone temperature in an HVAC deployment is assumed to only be affected by zone-level actuation devices. SCADA systems are prevalent in gas pipe deployments [Ene07], factory control [JM86], and nuclear power plants [MRVB00]. In contrast, Arrowhead provides a higher-level actuation abstration.

From a programming perspective, Mottola and Picco give an exhaustive survey of such techniques [MP11], as well as Sugihara and Gupta [SG08]. Techniques range from abstracting away networks, mobility, space, or time, but very few programming techniques deal with actuation in a generic way. Deshpande et al describe a system for HVAC systems, and in theory can be used for any deployment, but no results have yet been published [DGM05]. Other systems such as sMAP [DHJT$^+$10] give a web programming abstraction where all sensor data and actuation is done via a RESTful interface. Another approach is macroprogramming [KGMG07, HSH$^+$08, HAAIKR11, VHXS10]. Macroprogramming solutions abstract away the networking, distributed, and hardware access of a deployment to allow developers to focus on their applications themselves.

Programming abstractions for cyber-physical system involves giving abstractions for dealing with the physical environment. This can either be for a specific operation, or for the whole application solution. An example of an abstraction for a specific operation is Hotline. Hotline [BLW$^+$11] gives a distributed shared memory abstraction for addressing physical variables. Applications can acquire a distributed lock to a shared resource (i.e. camera) and have mutually exclusive access to it for a certain duration. Work on computation platforms has also recently emerged, particularly for smart grids [TGW11]. This area of work is still actively developing. Arrowhead is complementary to these abstractions.

Lastly, from the modeling side, work has been done in trying to model both the environments and the applications themselves. Lee and Seshia describe hybrid systems that combine continuous and discrete methods [LS11]. This is

expected because physical processes are often continuous, whereas computational systems are usually discrete. At the environment model, techniques known as system identification [Lju08] aim to construct environmental models with a blend of white-box first principles of the environment and black-box empirical data. Once system identification is done, they can be used for advanced process control techniques like model predictive control [Nik01]. These models can readily be used with Arrowhead.

2.6 Discussion and Future Work

There are several issues that Arrowhead does not account for. The first is model failure. Arrowhead does not require any accuracy or fidelity requirements from the model. This means it is possible that the model may give a poor, or even unsafe, solution. However, an advanced model can give a confidence interval along with the solution. If that is not possible, an operator can take the solution and manually perform actuation. If an application decides to create setpoints on actuator variables directly instead of on sensor variables, our kNNR model will filter out any solutions whose actuator variable setpoints are not satisfied. Creating robust and accurate models has been and will continue to be an area of interest in sensor-actuator networks.

Also related to the model is whether it can be continuously updated online. Although we have described our kNNR model as being an offline model, it can be converted to an online model by periodically flushing out old samples and running the learning algorithm on new data. This technique can be applied to any empirical-based model.

Another issue is hardware failure. It is important that when actuation is performed in the runtime, it is actually performed in the physical world. Arrowhead assumes that a separate system is in place for hardware and network management, such as SNMS [TC05].

Security is also an important issue for sensor-actuator networks. Arrowhead safety applications prevent other applications from having unsafe setpoints, but it

does not provide any security relating to privacy or denial of service. It is currently possible for an application to query for all sensor and device information. However, it is possible for operators to selectively expose the API and apply standard network security features such as firewalls.

Arrowhead is also not well suited for deployments that need very fresh data and thus have very small intervals. This is because all the applications must also be executed frequently, which cannot be guaranteed with unknown network latency. Even when all applications are running locally, we have not measured how performant the runtime server can be.

2.7 Acknowledgements

Chapter 2, in part, has been submitted for publication and may appear as Arrowhead: Coordinating Actuation Roles Through a Sensor-Actuation Software Platform. Kaisen Lin, Ted Tsung-te Lai, David Chu, Hao-hua Chu, Rajesh Gupta. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Control Command Dissemination in WSN

3.1 Introduction

Commodity wireless sensor-actuation deployments require a dissemination protocol to propagate actuation commands to individual devices because they are likely to be added to a deployment incrementally. For example, a single fan may be added in one room of an HVAC deployment before more fans are later added to different thermal zones. A common dissemination protocol allows them to all communicate with the main controller and scale as the number of devices grows.

Dissemination protocols such as MNP [Wan04], XNP [Cro], Deluge [HC04], and Sprinkler [NASZ07] distribute new binaries into a network, enabling complete system reprogramming. Dissemination protocols such as Maté's capsule propagation [LGC05] and Tenet's task propagation [GGJ+06] install small virtual programs, enabling application-level reprogramming. Finally, dissemination protocols such as Drip [TC05] allow administrators to adjust configuration parameters and send RPC commands [WTT+06].

Dissemination protocols reliably deliver data to every node in a network using key, version tuples on top of some variant of the Trickle algorithm [LPCS04]. We describe these protocols and their algorithms in greater depth in Section 3.2.

The key characteristic they share is a node detects a neighbor needs an update by observing that the neighbor has a lower version number for a data item (key). The cost of this mechanism scales linearly with the number of data items: T data items require T version number announcements. Even though a protocol can typically put multiple announcements in a single packet, this is only a small constant factor improvement. Fundamentally, these algorithms scale with $O(T)$ for T total data items. This linear factor introduces a basic cost/latency tradeoff. Nodes can either keep a constant detection latency and send $O(T)$ packets, or keep a constant cost and have an $O(T)$ latency.

The key insight in this chapter is that dissemination protocols can break this tradeoff by aggregating many data items into a single advertisement. Because these aggregates compress information, they can determine that an update is needed, but cannot always determine which data item needs an update. Section 3.3 outlines existing dissemination algorithms and describes a new algorithm, *search*, that breaks the cost/latency tradeoff, enabling fast and efficient dissemination. By using a hash tree of data item version numbers, a protocol using search can discover an update is needed with $O(log(T))$ transmissions.

In simple collision-free and lossless network models, search works well. However, two problems can make the hash tree algorithm perform poorly in real networks. First, packet losses can make it difficult to quickly traverse the tree. Second, the multiple advertisements caused by packet losses are completely redundant: there is typically only one subtree to explore. Through controlled simulation experiments, we find that in cases of very high loss or when a large fraction of items require updates, the underlying constant factors can cause randomized scans to be more efficient than hash tree searches.

Section 3.3 presents an analytical framework to understand these tradeoffs. The analysis shows that whether periodic advertisements or searches is more efficient depends on three factors: network density, packet loss ratios, and the percentage of items that need updates. While they have similar efficiency when reasonably close to their equality point, one can be a factor of two more efficient than the other at the edges. This analysis indicates that a scalable dissemination

protocol can get the best of both worlds by using a hybrid approach, dynamically switching between algorithms based on run-time conditions.

Section 3.4 presents such a protocol, called DIP (DIssemination Protocol). DIP continuously measures network conditions and estimates whether each data item requires an updates. Based on this information, it dynamically chooses between a hash tree-based search approach and scoped randomized scanning. DIP improves searching performance by combining hashes over ranges of the key space with a bloom filter. Hashes allow it to detect whether there are version number inconsistencies while Bloom filters let it quickly pinpoint the source of the inconsistency.

Section 3.5 evaluates DIP in simulation and on a mote testbed. In simulated clique networks, DIP sends up to 30-50% fewer packets than either scanning or searching and is correspondingly 30-50% faster. In the Intel Mirage multihop 80 node testbed, DIP sends 60% fewer packets than scanning or searching. In some cases, DIP sends 85% fewer packets than scanning, the dominant algorithm in use today. By improving its transmission efficiency, DIP is also able to disseminate faster: across real, multihop networks, DIP is 60% faster for a few items and over 200% faster for many items. Section 3.6 presents how DIP relates to prior work.

These results show that DIP is significantly more efficient than existing approaches. This improvement comes at a cost of an additional $log(log((T))$ bits of state per data item for T items. Section 3.7 discusses the implications of these findings. The tradeoffs between scanning and searching touch on a basic tension in sensornet protocol design. While searching can find inconsistencies quickly by exchanging higher-level metadata, its deterministic operation means that it cannot leverage the communication redundancy inherent to wireless protocols. While scanning can take advantage of this redundancy through randomization, it does so by explicitly avoiding any complex metadata exchange. DIP 's results suggest the complex tradeoffs between randomized versus deterministic algorithms in wireless networks deserve further study.

This chapter makes three research contributions. First, it proposes DIP , an adaptive dissemination protocol that can scale to a large number of items.

Second, it introduces using a bloom filter as an optimization to update detection mechanisms in dissemination protocols. Third, it evaluates DIP and shows it outperforms existing dissemination protocols, reducing transmission costs by 60% and latency by up to 40%. These results suggest the complex tradeoffs between randomized versus deterministic algorithms in lossy networks deserve further study.

3.2 Motivation and Background

Efficiently, quickly, and reliably delivering data to every node in a network is the basic mechanism for almost all administration and reprogramming protocols. Matévirtual machines disseminate code capsules [LGC05]; Tenet disseminates tasks [GGJ+06]; Deluge [HC04], Typhoon [LMET08] and MNP [Wan04] disseminate binary images; Drip disseminates parameters [TC05] and Marionette builds on Drip to disseminate queries [WTT+06].

3.2.1 Trickle

All of these protocols use or extend the Trickle algorithm [LPCS04]. Trickle periodically broadcasts a summary of the data a node has, unless it has recently heard an identical summary. As long as all nodes agree on what data they have, Trickle exponentially increases the broadcast interval, thereby limiting energy costs when a network is stable. When Trickle detects that other nodes have different data, it starts reporting more quickly. If a node hears an older summary, it sends an update to that node.

In practice, protocols assign keys to data items and summaries use version numbers to determine if data is newer or older. For example, the MatéVM assigns each code capsule a unique number. Installing new code in the network involves selecting a capsule, incrementing its version number, and installing the new version on a single source node. That node starts quickly advertising it has a new version, shrinking its advertisement interval to a small value (e.g., one second). Neighbors hear the advertisement and quickly advertise they have an old version, causing the

source to broadcast the update and spread the new code.[1] This process repeats throughout the network until all nodes have the update. Trickle's transmission rate slows down, following the exponential interval increase rule up to a maximum interval size (e.g., one hour).

Systems use Trickle because they are efficient and scale logarithmically with node density. As nodes suppress redundant advertisements, Trickle can scale to very dense networks. This suppression is not perfect: packet losses cause the number of redundant advertisements to scale logarithmically with network density. By constantly adjusting its advertisement interval, Trickle advertises new data quickly yet advertises slowly when a network is consistent.

3.2.2 A Need for Scalability

While using Trickle enables dissemination protocols to scale to dense networks, no protocol currently scales well to supporting a large number of data items. As sensornet applications grow in complexity and nodes have more storage, administrators will have more parameters to adjust, more values to monitor, and a need for a larger number of concurrent capsules, tasks, or queries.

When an administrator injects new data to a single node, that node knows the data is newer. Therefore, disseminating new data with Trickle is comparatively fast. The more challenging case is when nodes need to detect that there is new data. This case occurs when disconnected nodes rejoin a network. Both the old and new nodes think that the network is up to date, and so advertise at a very low rate.

Because current protocols advertise (key, version) tuples, their transmission costs increase linearly with the number of distinct data items. To detect that a data item is different, a node must either transmit or receive a tuple for that item. This approach causes the cost/latency product of a trickle to scale with $O(T)$, where T is the total number of data items. Some protocols, such as Drip and Deluge, maintain a constant latency by keeping a fixed maximum interval size and

[1] MNP [Wan04] extends simple trickles in that it uses density estimates to decide which node sends the actual update.

disseminating each item with a separate trickle. As the number of items grows, the transmission rates of these protocols grow with $O(T)$. Other protocols, such as Tenet's task dissemination, keep a constant communication rate so detection latency grows with $O(T)$.

As sensor systems grow in complexity, linear scalability will become a limiting factor in the effectiveness of these protocols: it will force administrators to choose between speed and efficiency. The next section quantifies these tradeoffs more precisely, and introduces a new hash-tree based algorithm that resolves this tension, so dissemination protocols can simultaneously be efficient and fast.

3.3 Protocol Tradeoffs

Dissemination protocols have two main performance metrics: detection latency and maintenance cost. Maintenance cost is the rate at which a dissemination sends packets when a network is up-to-date. Traditionally, these two metrics have been tightly coupled. A smaller interval lowers latency but increases the packet transmission rate. A larger interval reduces the transmission rate but increases latency. Trickle addresses part of this tension by dynamically scaling the interval size, so it is smaller when there are updates and larger when the network is stable. While this enables fast dissemination once an update is detected, it does not help with detection itself.

Protocols today use two approaches to apply Trickle to many data items. The first establishes many parallel Trickles; the second uses a single Trickle that serially scans across the version numbers to advertise. This section proposes a third approach, which uses a hash tree to obtain constant detection latency and maintenance cost. To achieve this, searching introduces an $O(log(T))$ overhead when it detects an update is needed.

3.3.1 Scanning and Searching

Parallel detection uses a separate Trickle for each data item. Because the maximum trickle interval is fixed, parallel detection provides a detection latency

Table 3.1: Scalability of the three basic dissemination algorithms.

Protocol	**Latency**	**Cost**	**Identify**
Parallel Scan	$O(1)$	$O(T)$	$O(1)$
Serial Scan	$O(T)$	$O(1)$	$O(1)$
Search	$O(1)$	$O(1)$	$O(log(T))$

bound independent of the number of items. However, this bound comes at a cost: parallel detection has a maintenance cost of $O(T)$.

Serial detection uses a single Trickle for all items. Each transmission contains a selection of the (key,version) tuples. Because serial detection scans across the tuples, it requires $O(T)$ trickle intervals to transmit a particular tuple. Therefore, serial detection has a latency of $O(T)$.

The parallel and serial approaches represent the $O(T)$ cost/latency product that basic trickles impose. One can imagine other, intermediate approaches, say where both cost and latency increase as $O(\sqrt{T})$. In all such cases, however, the cost/latency tradeoff persists.

A third approach is to *search* for different items using a hash tree, similar to a binary search. When a node sends an advertisement, it sends hashes of version numbers across ranges of data items. When a node hears an advertisement with a hash that does not match its own, it sends hashes of sub-ranges within that hash. This simple protocol backs up one tree level on each transmission to prevent locking.

When a network is stable, nodes advertise the top-level hashes that cover the entire keyspace. As these hashes cover all items, searching can detect new items in $O(1)$ time and transmissions. Determining which item is new requires $O(log(T))$ transmissions. As these transmissions can occur at a small Trickle interval rate, the latency of identifying the items is insignificant compared to detection.

While searching is much more efficient in detecting a single new item, it can be inefficient when there are many new items. This can occur, for example, if an administrator adds nodes that require all of the software updates and programs running in the network. Since searching pays an $O(log(T))$ cost for each item, with N new items its cost will be $O(N \cdot log(T))$. In the worst case, this can be

Table 3.2: Network parameters.

Term	Meaning
T	Total data items
N	New data items
D	Node density
L	Loss ratio

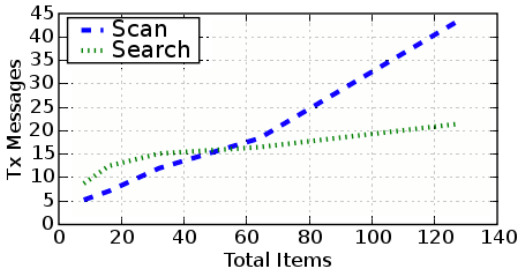


Figure 3.1: As T increases but N is constant, the chances a scan will find a new item goes down, and searches become more effective.

$O(T \cdot log(T))$, which is more expensive than the $O(T)$ of scanning approaches. Table 3.1 summarizes these tradeoffs, and Figure 3.1 shows which algorithm is better as T changes.

3.3.2 Analysis

Loss and density affect protocol performance. In the rest of this chapter, we describe networks with the terms in Table 3.2. Trickle introduces a communication redundancy R of $log_{\frac{1}{L}}(D)$. This comes from the probability that a node with an area of density D will advertise even if $R-1$ nodes have already advertised because it lost those packets.

In the case of a parallel scan protocol, these extra transmissions are completely redundant: there will be $R \cdot T$ transmissions per interval, and detection latency remains $O(1)$. In serial scan protocols, these extra transmissions are not completely redundant: nodes may be at different points in their scan, or might be advertising a random subset. Because they are not redundant, scanning's latency goes down: these extra transmissions further cover the keyspace. Therefore, the

detection latency of parallel scans are $O(\frac{T}{R})$.

Extra messages in search protocols are redundant for the same reason they are in serial scans. When nodes detect a hash mismatch, they will all respond with the same set of sub-hashes. Furthermore, if a node does not hear a sub-hash, it assumes consistency and backs up one level in the hash tree: heavy packet loss can slow tree traversal.

Searching is typically advantageous when N is small compared to T. If N is large, then scanning is effective because a random selection of items is likely of finding an inconsistency. In contrast, searching requires traversing the tree, introducing control packet overhead. When N is small, this overhead is less than the number of packets a scan must send to find a new item.

Together, these tradeoffs mean that which of the algorithms performs best depends on network conditions. High R and N values improve scanning performance. But when N is small, searching is more efficient. Furthermore, the two approaches are not mutually exclusive; a protocol can search until it determines the new item is in a small subset, at which point R may make scanning that subset more efficient than continuing the search. Thus an ideal protocol should switch from a scan to a search when nodes are down to their last few items and adjust to network conditions. The next section proposes such as protocol.

3.4 DIP

DIP is a hybrid data detection and dissemination protocol. It separates this into two parts: detecting that a difference occurs, and identifying which data item is different. DIP dynamically uses a combination of searching and scanning based on network and version metadata conditions. To aid its decisions, DIP continually estimates the probability that a data item is different. DIP maintains these estimates through message exchanges. When probabilities reach 100%, DIP exchanges the actual data items. It is an eventual consistency protocol in that when data items are not changing, all nodes will eventually see a consistent set of data items.

This section starts with a broad overview of how DIP works. It introduces

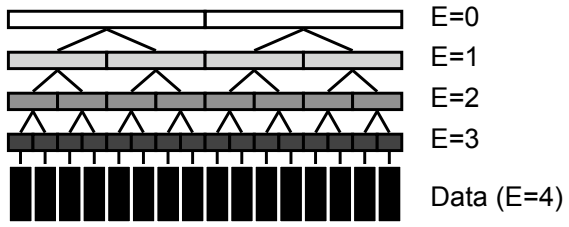


Figure 3.2: Example estimate values for a 16-item hash tree.

DIP 's metadata structures, the messages it exchanges, its use of Trickle, and the details of its estimate system before finally describing the algorithms DIP applies when receiving and transmitting packets.

3.4.1 Overview

DIP stores a version number for each data item. In the steady state where all nodes are up to date and have the same versions, DIP uses Trickle to send hashes that cover all of the version numbers. Nodes that receive hashes which are the same as their own know they are consistent with their neighbors. If a node hears a hash that differs from its own, it knows that a difference exists, but does not know which specific item or who has the newer version.

In addition to the version number, DIP maintains a soft-state estimate of whether a given item differs from a neighbor's. It is soft in that if estimates are temporarily inaccurate or lost, the protocol will still proceed. In contrast, version numbers must be correct for consistency and correctness.

When DIP detects a hash of length H that differs, it gives each item covered by the hash a conservative estimate of $\frac{1}{H}$. This estimate is conservative because at least one of the H items is different.

DIP sends advertisements that improve its estimate accuracy by using smaller hashes. For example, a node that receives a differing hash of length H can respond by sending two hashes of length $\frac{H}{2}$. As Figure 3.2 shows, one can think of these levels of hashes defining a hash tree over the version number set; going down the tree involves sending smaller hashes, while going up the tree involves sending longer hashes.

Identifying which data item is different and which node has the newer version requires exchanging actual version numbers. In the hash tree, version numbers are hashes of length 1. Section 3.3 showed how if the probability of a version number difference is large enough, then transmitting a random subset of the version numbers can be more efficient than traversing the hash tree. To take advantage of this behavior and determine the transition point, DIP monitors network conditions, such as Trickle communication redundancy. Rather than always walk to the bottom of the hash tree, DIP starts sending precise version information when estimates reach a high enough value that suggest random scanning would be more efficient.

3.4.2 Metadata

DIP maintains a version number and unique key for each data item. As a result of having a unique key, it also assigns each data item an index in the range of $[0, T-1]$. DIP can describe a data item i as a tuple (k_i, v_i) where k_i is the data item key and v_i is its version number. The implementation of DIP we describe in this chapter uses 32-bit version numbers, to preclude wrap-around in any reasonable network lifetime; smaller or larger values could also be used.

In addition to version numbers, DIP maintains estimates of whether an item is different. DIP stores estimates as small integers in the range of $[0, log(T)]$.[2] An estimate value of E means that DIP detected a difference at level E in the hash tree. With a tree branching factor of b, this means a hash that covers $\frac{T}{b^{E+1}}$ items.

Together, these two pieces of metadata are $log(V) + log(log(T))$ bits per data item, where V is the maximum version number. In practice, $log(V)$ is a small constant (e.g., 4 bytes). Compared to the basic Trickle, which requires $O(T)$ state, DIP requires slightly more, $O(T + T \cdot log(log(T)))$, as it must maintain estimates. In practice, the $log(log(T))$ factor is a single byte for simplicity of implementation, so DIP uses 5 bytes of state per data item in comparison to standard Trickle's 4 bytes.

[2]The implementation we describe in Section 3.5 actually stores values in the range $[0, log(T)+2]$ to save a few bits in transmit state.

3.4.3 Messages

The prior section described the per-item state each DIP node maintains with which to make protocol decisions. This section describes the types of messages DIP nodes use to detect and identify differences in data sets among their neighborhood. DIP , like Trickle, is an address-free, single-hop gossip protocol that sends all messages as link-layer broadcasts. DIP seamlessly operates across multihop networks by applying its rules iteratively on each hop. DIP uses three types of messages: data, vector, and summary.

Data Messages: Data messages transmit new data. They have a key k_i, a version number v_i, and a data payload. A data message unambiguously states whether a given item is different. On receiving a data message whose version number is newer than its own, DIP installs the new item.

Vector Messages: Vector messages hold multiple key, version tuples. The tuples may have non-consecutive keys. Vector messages, like data messages, unambiguously state whether a given item is different. They do not actually update, however.

Summary Messages: Figure 3.3 illustrates a complete summary message. Summary messages contain a set of summary elements and a salt value. Each summary element has two indices describing a set of version numbers, a summary hash over the set, and a bloom filter of the set. The salt value is a per-message random number that seeds the summary hash and bloom filter to protect from hash collisions.[3] When the size of the set covered by the summary hash is small enough, the filter can circumvent several hash tree levels to find which item is inconsistent. The number of summary elements in a summary message determines the branching factor of the DIP hash tree.

The summary hash function SH is $SH(i_1, i_2, s)$ where i_1 and i_2 are two indices representing left and right bounds of the search, and s is a salt value. Its output is a 32-bit hash value. For example, $SH(0, T-1, s)$ would be a hash over all the current version numbers for all data items. The specific function is a one-at-a-time hash using a combination of bit shifts and XOR operations, which an

[3]We borrow the term "salt" from UNIX password generation [MT79].

Salt s			
Begin_1	End_1	BloomFilter_1	SummaryHash_1
...			
Begin_n	End_n	BloomFilter_n	SummaryHash_n

Figure 3.3: DIP Summary Message

embedded microcontroller can compute without much difficulty.

Each summary message has a bloom filter, a probabilistic data structure that can be used to test an item's membership. Computing a bloom filter of length B bits involves taking a hash $BH(i, v, s)$ of each covered item, where i is the item index, v is the version number, and s is the salt. The result of each BH modulo B is a single bit position that is set to 1 in the filter[4]. If two bloom filters share an (index, version) tuple, they will both compute the same bit to be 1: there are no false negatives, where both sets agree but the filters differ. Bloom filters can have false positives, where two sets differ but have the same filter.

Each of the three message types provides DIP with different information. Data and vector messages can identify which data items are different. However, the detection cost of finding the item is $O(T)$. Summary messages can tell DIP that there is a difference, but not definitively which item or items differ. Summary messages have a $O(1)$ detection cost and a $O(log(T))$ identification cost. In practice, for reasonable T values (below 1,000), the bloom filter significantly reduces the identification cost.

3.4.4 Updating Estimates

The prior two sections explained the state that DIP maintains and the messages it exchanges. As version numbers only change on data updates, estimates constitute most of the complexity of DIP 's state management. This section explains how DIP adjusts its estimates in response to receiving packets. The next section explains how DIP decides what packets to send.

Section 3.4.2 stated that estimate values E are in the range of $[0, log(T)]$,

[4]Bloom filters generally use k hash functions to set k bits. In DIP, $k = 1$

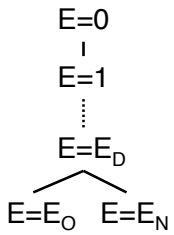


Figure 3.4: Estimate values. $E_D = \lfloor log_b(T) \rfloor$, where b is the number of summary elements in a summary message (the branching factor). E_O denotes a neighbor has an older item, while E_N denotes a neighbor has a newer item.

where a 0 represents a belief that the data item is consistent with all neighbors, and value of $log(T)$ represents certainty that the item is different We denote $log(T)$ as E_D, as it denotes a difference exists. In addition to this range, DIP reserves two extra values: E_O, which denotes a nearby node has an older version, and E_N, which denotes a neighbor has a newer version. These two values are necessary because E_D only denotes that a node has a different version than one of its neighbors, but not the difference direction. Figure 3.4 summarizes the estimate values.

DIP uses the information in data, vector, and summary packets to update its estimates. On receiving a packet that indicates a neighbor has the same version number for an item, DIP decrements that item's estimate, to a minumum of zero. In the base case, when all nodes agree, estimates converge to zero. On receiving a packet that indicates a neighbor has a different version number, DIP improves its estimates for the relevant items as well as it can from the information it receives. More precisely, DIP adjusts estimate values with the following rules:

1. Receiving a vector or data message with an older version number sets that item to E_0 unless it is E_N.
2. Receiving a vector or data message with the same version number decrements that item's E, to a minimum of 0.
3. Receiving a data message with a newer version number, it updates the item and sets it to E_O.
4. Receiving a vector with a newer version number sets that item to E_N.

5. Receiving a summary element with a differing hash of length H sets E for all items that hash covers to be the maximum of their current E and $log(T) - log(H)$.
6. Receiving a summary element with a differing hash and differing bloom filter sets all items with a differing bloom filter bit to E_D.
7. Receiving a summary element with a matching hash decrements the E of all items the hash covers, to a minimum of 0.

The first two rules are identical for vector and data messages. In the case of receiving an older version number, DIP sets the item to be E_O, denoting it should send an update, unless the item is already E_N, denoting it should receive an update. Sending an update with an out-of-date item is a waste, so a node prefers to wait until it is up-to-date before forwarding updates.

The third and fourth rules define what occurs when DIP receives a packet with a newer version number. If the version number is in a vector message, DIP knows it needs an update, so sets the item to E_N. If the version number is in a data message, then DIP has received the update, which it installs. DIP then sets the item to E_O, as chances are another node nearby needs the update as well.

The last three rules define how DIP responds to summary messages. Like data and vector messages, receiving a matching summary decrements estimate values. When DIP receives a differing hash that can provide a more precise estimate, it increases its estimates to the value the hash will allow. If the bloom filter allows DIP to pinpoint that an item is different, then it sets that item's estimate to E_D. It determines this by checking the bloom filter bit for each item index, version pair. If the bit is not set, then there is certainly a difference. Because summary messages contain multiple summary entries, a single message can trigger one, two, or all three of rules 5, 6 and 7.

3.4.5 Transmissions

Section 3.4.4 described what happens on a message reception. This section describes how DIP decides which message types to transmit and what they contain.

DIP uses a Trickle timer to control message transmissions. In the maintenance state when no data items are different, the DIP Trickle timer is set at a maximum interval size of τ_h. As soon as a difference is detected, the Trickle interval shrinks to the minimum interval of τ_l. When all estimates return to 0, DIP doubles its Trickle interval until τ_h is reached again.

DIP also uses hierarchical suppression to further prevent network flooding. Messages of the same type may suppress each other, but summary messages cannot suppress vector messages. This is because vector messages are more precise and are often used near the end of a search. This hierarchical suppression prevents nodes from suppressing more precise information.

DIP 's transmission goal is to identify which nodes and items need updates. It accomplishes this goal by increasing estimate values. In the steady state, when all nodes are up to date, DIP typically sends summary messages whose summary elements together cover the entire set of version numbers. All DIP transmissions are link-layer broadcasts.

All tuples in vector messages and summary elements in summary messages have the same estimate value: a packet always represents a single level within the DIP hash tree. DIP always transmits packets which contain information on items with the highest estimate values. Together, these two rules mean that DIP transmissions are a depth-first, parallelized, search on the hash tree.

We now describe the decisions DIP makes. DIP 's decisions are made based on local information to each node. This, coupled with the soft-state properties of the estimates, allow nodes running DIP to seamlessly leave and join both singlehop and multihop networks.

If an item has an estimate of $E = E_O$, DIP sends a data message. Because receiving an update causes DIP to set E to E_O, forwarding an update takes highest priority in DIP . Of course, hearing the same data message, or other packets that decrement E may prevent this data transmission; however, since DIP is based on randomized Trickle timers, in practice it soon discovers if a node is out of date and increases E to E_O.

If $E \neq E_O$, DIP compares the vector message and summary message costs of

identifying a difference. If v key/version pairs can fit in a vector message and d data items need to be covered (computed from E), then DIP requires $\frac{d}{v}$ transmissions to scan through all d items. For summary messages, DIP requires $E_D - E$ (the number of levels until the bottom of the hash tree) transmissions. Assuming lossless communication, DIP transmits summaries when $(E_D - E) > \frac{d}{v}$ and vectors when $(E_D - E) < \frac{d}{v}$. This inequality means that if an item has an estimate of E_D or E_N, DIP always transmits a vector message.

Because real networks are not lossless, DIP adjusts its decision based on the degree of communication redundancy it heard in the last trickle interval. Hearing more messages in a given interval is related to a failure in the suppression mechanism and most likely caused by a loss rate or dense network. DIP accounts for this by changing the vector message calculation from $\frac{d}{v}$ to $\frac{d}{c \cdot v}$ where c is the redundancy.[5] If it takes $\frac{d}{v}$ vector messages to determine a new data item, but it is receiving c messages, then weight the required number of vector messages by a factor of c.

When DIP transmits version messages, it selects a random subset of the data items which have the highest estimate value. As Section 3.3 showed, randomization is critical as communication redundancy increases with density.

When DIP transmits summary messages, it performs a single linear pass across the data items to find contiguous runs of items that have the highest estimate value. It generates summary elements for these items that are one level lower on the hash tree. For example, if DIP finds a run of items with an estimate of E, it generates summary elements which each cover $\frac{T}{b^{E+1}}$ items. While the items within each summary element must be contiguous, the summary elements themselves do not need to be.

Finally, when DIP transmits, it decrements the estimate values of all data items the transmission covers. In the case of data and vector messages, it decrements the estimates of the items version numbers that are in the packet. In the case of summary messages, it decrements the estimates of all items covered by a summary element.

[5] We borrow the term c from Trickle's communication counter.

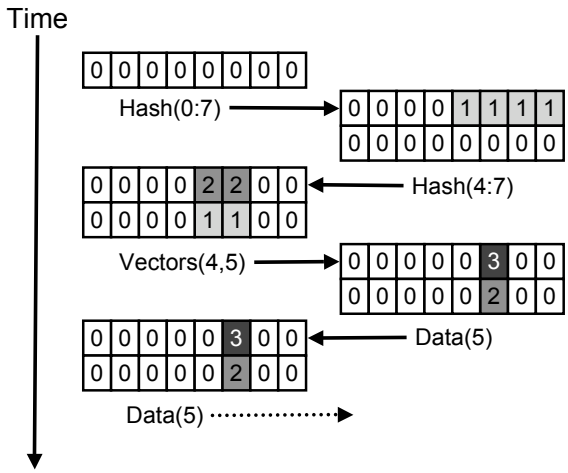
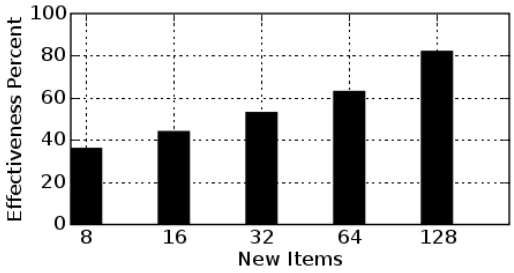


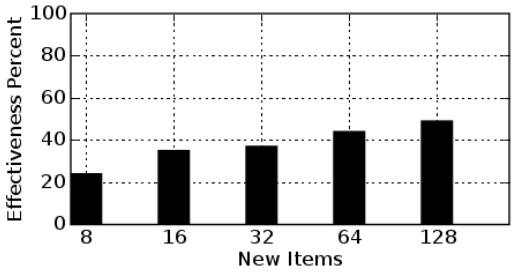
Figure 3.5: Two nodes exchanges summaries and vectors to determine that the left node needs an update for item 5. The arrays show each node's estimate values during each step of the packet exchange. In this example, summaries contain 2 summary elements.

3.4.6 Example

To give a concrete example of what a DIP search looks like and how the estimate update rules work, Figure 3.5 shows two nodes running DIP detecting that one node needs an update for item 5. For simplicity, this example assumes bloom filters never help, and DIP only sends vector messages at the bottom of the hash tree. Each time a node transmits a summary message or a vector message, it reduces the estimate of the covered items. First, the left node transmits a full summary. The right node sees that the right summary hash (covering items 4–7) is different, so replies with summary hash of 4–5 and 6–7. The left node sees that 4–5 is different, so replies with the version numbers of 4 and 5. The right node replies with the data for item 5. The left node rebroadcasts the data in case any neighbors do not have it, which propagates the update acros the next hop. The nodes exchange a few more summary messages to make sure that the different summary hashes covered only one update.



(a) D = 2.



(b) D = 32.

Figure 3.6: Bloom filter effectiveness with L = 0%, T = 256 with two different densities and varying N.

3.5 Evaluation

We divide our evaluation of DIP into two parts. In the first part, we look at improvements. We measure the effectiveness of DIP 's two primary optimizations: bloom filters and using scans when the chance of hitting a new item is high. These experiments are all in simulation. In the second part, we look at comparisons. We compare DIP against a scan and search protocols in simulation and a real network. In simulation, we measure how parameters affect DIP 's transmission costs by measuring the cost for the whole network to complete all updates. On a mote testbed, we compare the performance of the three algorithms for different N.

3.5.1 Methodology

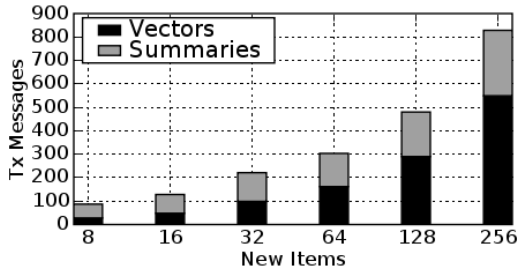
We implemented DIP in TinyOS 2.0: it compiles to 3K of program code. In our implementation of DIP , we had 2 summary elements per summary message,

and 2 key/version tuples per vector messages. The maximum Trickle interval was set at one minute, and the minimum interval at one second, though these numbers can be adjusted on a real deployment. We also implemented intelligent versions of the serial scan and search algorithms that use DIP -style estimates. Using estimates greatly improves the performance of these algorithms as they maintain some state on what to advertise. All three protocols use a single underlying Trickle. Scanning sends two (key,version) tuples per packet and sequentially scans through items. Searching uses a binary hash tree. When search identifies a different item, it updates the item and resets to the root of the tree.

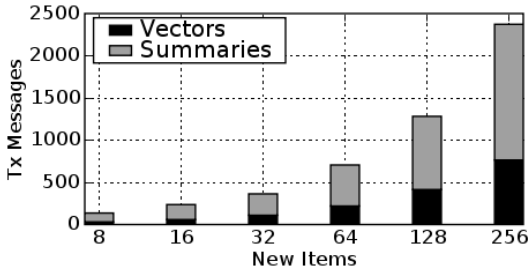
To evaluate simple clique networks and multihop simulations, we used TOSSIM, a discrete event network simulator that compiles directly from TinyOS code [LLWC03]. In TinyOS 2.0, TOSSIM uses a signal-strength based propagation and interference model. Nodes have uniformly distributed noise along a range of 10dB and a binary SNR threshold of 4dB. The model distinguishes stronger-first from stronger-last collisions, such that if a stronger packet arrives in the middle of a weaker one, the radio does not recover it (this is how the CC2420 radio on Telos and micaZ nodes behaves). Therefore, lossless communication in TOSSIM does not mean all packets arrive successfully: there can still be collisions. Fully connected networks are not collision-free because TOSSIM models radio RX/TX turnaround times.

To set a uniform loss rate, we configured TOSSIM to have noise values in the range of -110dBm to -100dBm and tuned the signal strength to obtain the desired loss rate. For a loss rate of 10%, we set link strengths to be -96.5dBm; for 20%, -97dBm; for 30%, -97.5dBm; for 40%, -98dBm, and for 50% we set the signal strength to be -99dBm. These values are not a simple linear progression as uniform loss distributions would suggest because TOSSIM samples noise several times during a packet. We ran several iterations for validity and averaged their results when applicable.

To collect empirical data, we ran our experiments on the Mirage testbed [CBA+05]. It is composed of 100 MicaZ motes that are distributed throughout an office environment. Each MicaZ consists of an Atmel Atmega128L microcontroller,



(a) L = 0.



(b) L = 20.

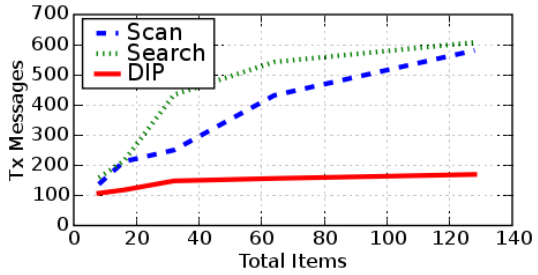
Figure 3.7: Message types used by DIP with T = 256, D = 32, two different loss rates, and a varying N.

128K program flash, and an 802.15.4 compatible radio transceiver that transmits at 250kbps. We instrumented the DIP code to write to the UART various statistical information when events occur. Then using a UART to TCP bridge, we listened on on the port of each node and collected the information at millisecond granularity. Although there are 100 nodes in the network, we could only connect to 77 of them for data gathering purposes.

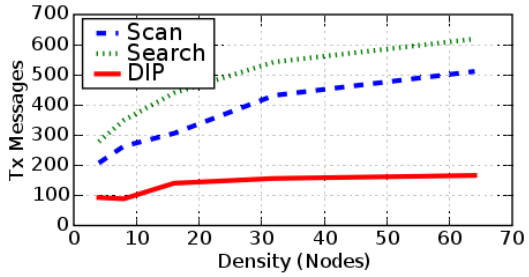
3.5.2 Improvements

We evaluate how bloom filters improve DIP 's performance by measuring how many summary messages successfully used a bloom filter to identify which item needed an update. Detecting with a bloom filter enables DIP to circumvent subsequent summary exchanges to traverse the tree, reducing the number of summary transmissions.

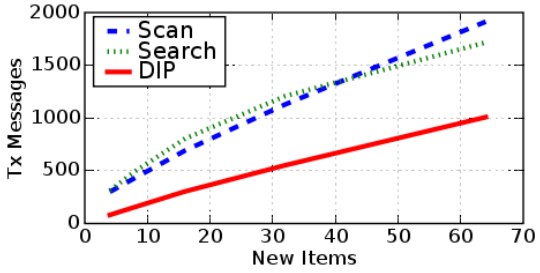
Figure 3.6(a) shows results from the very simple case of a pair of TOSSIM



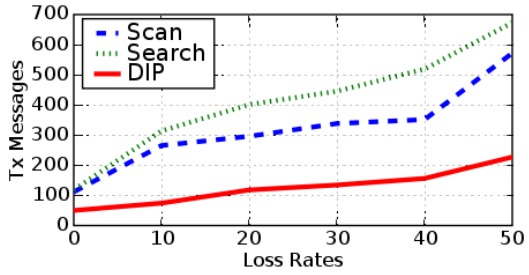
(a) Total items (T).



(b) Density (D).



(c) New items (N).



(d) Loss rate (L).

Figure 3.8: DIP compared with scan and search protocols. By default, N=8 (new items), T=64 (total items), D=32 (clique size), and L=40% (loss rate). Each figure shows how varying parameter affects DIP relative to other protocols.

nodes with lossless communication. For different N, bloom filter optimizations have a hit rate of 35%-80%.

Figure 3.6(b) shows that bloom filters are also effective in a 32-node network, albeit less so than a single node pair. At higher densities, different nodes will require different items over time. Thus the issue is not that a single node requires many items, but rather many nodes require a few items, making N small for each node.

To better understand the decisions that DIP makes, we measured the distribution of transmitted summary and vector messages for different loss rates and new items in a 32 node clique. Figure 3.7(a) shows that as the number of new items increases, DIP uses more vectors and at a higher proportion. This is a result of DIP 's dynamic adjustment to the number of new items. When the network becomes lossy as shown in Figure 3.7(b), an increase in N causes DIP to use more vectors and at a larger proportion, but this increase is not as fast as in a lossless network. High loss slows the increase of estimate values, leading to more summary messages and delaying when it uses vector messages. Although the X-axis on Figures 3.7 show N doubling, DIP 's total transmission count does not double, as using pure searches might suggest: its message adaptation allows it to take advantage of high hit rates.

These results show that bloom filters improve DIP 's detection and DIP dynamically improves its transmission policy based on network conditions.

3.5.3 Protocol Comparisons (TOSSIM)

Because there are four possible parameters, we explore each one independently using TOSSIM. For simplicity, these experiments are all fully connected networks with a uniform loss rate.

In our first experiment, we evaluated the scalability of each protocol by having a constant number of new data items ($N = 8$) while varying up the total number of data items (T). We measured the total number of transmissions required to update each node. Figure 3.8(a) shows the results. As expected, searching requires $O(log(T))$ transmissions and scanning requires $O(T)$. DIP performs much

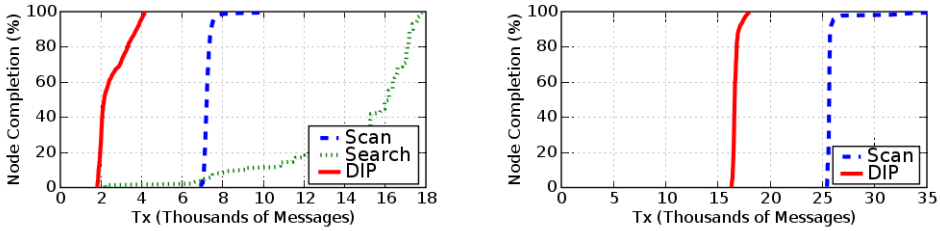
better because it detects items quickly through the bloom filter. Even though messages are lost, a single successful bloom filter will identify the items and thus not require a full search. Furthermore, DIP keeps a narrow search by estimate-based back outs, making bloom filters more effective.

Figure 3.8(b) shows how density affects scanning and searching performance. Because both improved protocols handle redundancy, they have similar lines. Scanning, however, is better than searching because there is no overhead associated with scanning. At high densities, the improved scan protocol does not pay extra to cover its entire keyspace. DIP again performs the best because it has the benefits of both low overhead through bloom filters, but also being able to find items quickly through searching.

Figure 3.8(c) shows how performance changes as the number of new data items changes. When there are many new data items, the $log(T)$ search overhead becomes noticeable, but not significant due to the fact our search protocol implementation handles redundancy. The scan protocol scales linearly because each additional new item requires a constant amount of identification overhead. When almost all data items are new, it may seem counterintuitive that searching is better. The reason for this is because after nodes have been updating, the few remaining items at the end are hard to identify. Thus, the early scan advantage cancels out at the end. DIP scales linearly, but at a smaller constant factor.

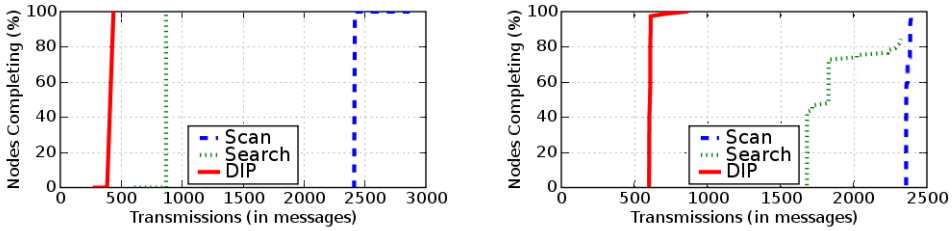
Figure 3.8(d) shows how performance changes as the loss rate increases and the results are similar to that of figure 3.8(b) due to redundancy.

In a multihop network, we are interested in how many transmissions are required for the whole network to detect new items. In multihop topologies, nodes must request new data from neighbors, while at the same time servicing other neighbors. We used the 15 by 15 sparse grid in the TinyOS 2.0 TOSSIM libraries, which uses Zuniga et al.'s hardware covariance matrix [ZK04] to model link asymmetries and other real-world effects. We modified the multihop implementations of the scan and search protocols to perform better in multihop situations as well. The scan protocol re-advertises items 3-4 times after receiving an item, while the search protocol uses estimates to back out rather than resetting.



(a) N = 8 (b) N = 32

Figure 3.9: Transmission costs to complete reception of all new data items on 15 by 15 grid with 64 total items and two different values of new items (TOSSIM).



(a) N = 8 (b) N = 32

Figure 3.10: Transmission costs to complete reception of all new data items on Mirage with 64 total items and two different values of new items.

Our first experiment examines transmission costs when $N = 8$ and $T = 256$. Figure 3.9(a) shows the results. The scanning protocol completes a large majority of nodes very close together, but the last few nodes take exceedingly long. When different items are discovered, the scan protocol repeatedly transmits high estimate items until estimate levels have decreased. At the end, when only a few nodes have old items, scanning cannot find those few items very well. Searching performs far worse because nodes are both senders and receivers in multihop topologies and searching requires efficient pairwise communication. DIP has similar performance to scanning until around half the nodes are complete. Afterwards, it begins searching and completes the remaining nodes. The gap between when the first and last node completes is marginally smaller in DIP . DIP 's tail begins to occur when a majority of the nodes have finished. Finally, DIP uses only 40% as many transmissions as scanning, a 60% improvement.

In the second experiment, shown in Figure 3.9(b), there were 32 new items instead of 8. DIP completed disseminating to all nodes within 18,000 transmissions. The overhead of searching caused the search protocol (not shown) to not finish within 35,000 transmissions. Multihop topologies force nodes to send and receive in different cells. This is problematic for searching, which requires pairwise communication to succeed. In contrast, the scan protocol finished with just over 35,000 transmissions and exhibited a very long tail, as scans are inefficient at finding the last few items. DIP has a shorter tail due to its ability to identify items through the bloom filter.

3.5.4 Protocol Comparisons: Mirage

We ran two testbed experiments with $T = 64$. We measured how many transmissions updated the whole network, and timed out each experiment after 400 seconds.

Figure 3.10(a) shows the per-node completion CDF for $N = 8$. DIP completed with 436 transmissions, while the search and scan protocols required 868 and 2867 respectively. DIP and the search protocol had steep curves meaning the nodes completed within a short time of each other. This is because DIP and

the search protocol (which was modified) back out after receiving or sending new items, keeping searches narrow enough for nodes in other cells to find items. The modified scan protocol is also steep, but has a long tail, which is caused from being unable to find the last few items quickly after most of the network has finished. Furthermore DIP finished with a much better overall time compared to the other modified protocols. This is due to DIP 's ability to identify items faster through its bloom filters. In terms of time, DIP took 86 seconds to deliver all 8 items while scanning took and searching took 107 and 143 seconds, respectively, a speedup of 24-60%.

Figure 3.10(b) shows results for $N = 32$. With more new items, search's overhead grows, and its performance relative to scanning degrades. While DIP was able to disseminate to every node in approximately 860 packets, neither scanning nor searching completed in over 2500 packets: both timed out. As dissemination layers today use scanning algorithms, DIP reduces the cost by up to 60%. While neither scanning nor searching was able to complete in 400 seconds, DIP took 131 seconds, a speedup of over 200%. This means, for example, when introducing new nodes to a network, DIP will bring them up to date to configuration changes up to 200% faster than existing approaches.

3.6 Related Work

DIP draws heavily from prior work in reliable data dissemination. Unlike protocols for wired networks, such as SRM [FJM+95] and Demers' seminal work on epidemic database replication [DGH+87], DIP follows Trickle's approach of using local wireless broadcasts [LPCS04]. Trickle's suppression and rate control protects DIP from many of the common problems that plague wireless protocols, such as broadcast storms [NTCS99]. Existing systems, such as Deluge [HC04], Maté [LGC05], and Tenet [GGJ+06] use protocols that assume the number of data items is small. DIP relaxes this assumption, enabling scalable dissemination for many items. Unlike flooding and broadcast protocols such as RBP [SHSM06], DIP provides complete reliability as long as the network is connected.

DIP's hashing is similar to Merkle hash trees [Mer79], a common mechanism in secure systems. As Merkle hash trees need to minimize computation, each level is a hash of the hashes of the level below it. As the tree stores all of these hashes, changing a single leaf requires only updating $log(n)$ hashes. In contrast, DIP dynamically computes hashes of leaf values on demand. This approach stems from how the resource tradeoffs between sensor nodes and traditional storage systems differ: on a sensor node, the RAM to store a hash tree is expensive, while CPU cycles to hash a range of version numbers is cheap.

Bloom filters have a long history in networked systems, including web caching, Internet measurements, overlay lookups, and keyword searches [BM04]. Keyword searches are the most similar, but with an opposite purpose: while they find similarities in filters, DIP seeks to find differences. Bloom filters are commonly used in distributed and replicated IP systems (e.g., PlanetP [CAPMN03]), but to our knowledge DIP represents their first use in wireless dissemination.

The tradeoffs between deterministic and randomized algorithms appear in many domains. At one extreme of data reliability, standard ARQ algorithms repeat lost data verbatim. At the other extreme, fountain codes [Lub02] used randomized code blocks to reliably deliver data. At the cost of a small data overhead ϵ, a fountain code requires no explicit coordination between the sender and receiver, trading off a bit of efficiency for simplicity and robustness. There are also many techniques that lie between these extremes, such as incremental redundancy (or Hybrid ARQ) [Sol03], which randomizes code bits sent on each packet retransmission. Similarly, Demers et al.'s landmark paper on consistency in replicated systems explored the tradeoffs between deterministic and randomized algorithms [DGH$^+$87]. The Trickle algorithm adds another level of complexity to these tradeoffs due to its inherent transmission redundancy.

3.7 Conclusion

This chapter presents DIP , an adaptive dissemination algorithm that uses randomized and directed algorithms to quickly find needed updates. To achieve

this, DIP maintains estimates of the probability that data items are different and dynamically adapts between its algorithms based on network conditions. This adaptation, combined with bloom filters, enables DIP to efficiently support disseminating a large number of data items and achieve significant performance improvements over existing approaches. The tradeoffs between searching and scanning show a basic tension between deterministic and randomized algorithms. Acting optimally on received data works best in isolation, but in the case of redundancy, multiple nodes each taking a sub-optimal part of the problem can together outperform a locally optimal decision. DIP leverages this observation to greatly improve dissemination efficiency; it remains an open question whether other wireless protocols can do the same.

3.8 Acknowledgements

Chapter 3, in part, is a reprint of the material as it appears in Data Discovery and Dissemination with DIP. Kaisen Lin, Philip Levis. In IPSN 2008: 7th International Conference on Information Processing in Sensor Networks, pages 433-444, Washington, DC, USA 2008, IEEE Computer Society. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Sensing and Actuation with Mobile Web Applications

4.1 Introduction

Mobile phones serve as an important piece of commodity sensor-actuation systems. In Chapter 2, we used mobile phones as a way to notify building operators when manual actuation should be performed. In this chapter, we extend the use of mobile phones to serve as sensors, as well as a JavaScript application execution environment. This has the potential to enable sensor-actuation applications to be executed directly from a website.

Web browsers provide an increasingly rich execution platform. Unfortunately, browsers have been slow to expose hardware devices to JavaScript [Fla06], the most popular client-side scripting language. This limitation has become particularly acute as sensor-rich devices like phones and tablets have exploded in popularity. A huge marketplace has arisen for mobile applications that leverage data from accelerometers, microphones, GPS units, and other sensors. Phones also have increasingly powerful computational and storage devices. For example, graphics processors (GPUs) are already prevalent on phones, and using removable storage devices like SD cards, modern phones can access up to 64GB of persistent data.

Because JavaScript has traditionally lacked access to such hardware, web developers who wanted to write device-aware applications were faced with two unpleasant choices: learn a new plugin technology like Flash which is not supported by all browsers, or learn a platform's native application language (e.g, the Win32 API for Windows machines, or Java for Android). Both choices limit the portability of the resulting applications. Furthermore, moving to native code eliminates a key benefit of the web delivery model—applications need not be installed, but merely navigated to.

4.1.1 A Partial Solution

To remedy these problems, the new HTML5 specification [Hica] introduces several ways for JavaScript to access hardware. At a high-level, the interfaces expose devices as special objects embedded in the JavaScript runtime. For example, the `<input>` tag [OHM] can reflect a web cam object into a page's JavaScript namespace; the page reads or writes hardware data by manipulating the properties of the object. Similarly, HTML5 exposes geolocation data through the `navigator.geolocation` object [Pop]. Browsers implement the object by accessing GPS devices or network cards that triangulate signals from wireless access points.

As a result, there are two distinct models for creating device-aware programs:

- Applications can be written using native code or plugins, and gain the performance that results from running close to the bare metal. However, users must explicitly install the applications, and the applications can only run on platforms supporting their native execution environment.

- Alternatively, applications can be written using cross-platform HTML5 and JavaScript. Such applications do not require explicit installation, since users just navigate to the application's URL using their browser. However, as shown in the example above, HTML5 uses an inconsistent set of APIs to name and query each device, making it difficult to write generic code. Fur-

thermore, by exposing devices through extensions of the JavaScript interpreter, the entire JavaScript runtime becomes a threat surface for a malicious web page trying to access unauthorized hardware—once a web page has compromised the browser, nothing stands between it and the user's devices. Unfortunately, modern browsers are large, complex, and have many exploitable vulnerabilities [DHM08, RLZ09, Sec08].

Ideally, we want the best of both worlds—device-aware, cross-platform web pages that require no installation, but whose security does not depend on a huge trusted computing base like a browser.

4.1.2 Our Solution: Gibraltar

Our new system, called Gibraltar, uses HTTP as a hardware access protocol. Web pages access devices by issuing AJAX requests to a *device server*, a simple native code application which runs in a separate process on the local machine and exports a web server interface on the localhost domain. If a hardware request is authorized, the device server performs the specified operation and returns any data using a standard HTTP response. Users authorize individual web domains to access each hardware device, and the device server authenticates each AJAX request by ensuring that the referrer field [FGM+99] represents an authorized domain.

Unlike HTML5, Gibraltar does not require the browser to be fully trusted. Indeed, in Gibraltar, the browser is sandboxed and incapable of accessing most devices. However, a corrupted or malicious browser can send AJAX requests to the device server which contain snooped referrer fields from authorized user requests. To limit these attacks, Gibraltar uses *capability tokens* and *sensor widgets* [HS10]. Before a web page can access hardware, it must fetch a token from the device server. The page must tag subsequent hardware requests with the fresh capability.

To prevent a malicious browser from surreptitiously requesting capabilities from the device server, Gibraltar employs sensor widgets. Sensor widgets are ambient GUI elements like system tray icons that indicate which hardware devices are currently in use, and which web pages are using them. Sensor widgets help a user to detect discrepancies between the set of devices that she expects to be in

use, and the set of devices that are actually in use. Thus, sensor widgets allow a user to detect when a compromised browser is issuing HTTP hardware requests that the user did not initiate.

Using these mechanisms, a compromised browser in Gibraltar has limited abilities to independently access hardware (§4.5). However, a malicious browser is still the conduit for HTTP traffic, so it can snoop on data that the user has legitimately fetched and send that data to remote hosts. Gibraltar does not stop these kinds of attacks. However, Gibraltar is complementary to information flow systems like TightLip [YMC07] that can prevent such leaks.

4.1.3 Advantages of Gibraltar

Gibraltar's device protocol has four primary advantages:

- **Ease of Deployment:** Gibraltar allows device-aware programs to ship as web applications that do not need to be installed. The device server does need to be installed, but it can ship alongside the browser and be installed at the same time that the browser itself is installed.

- **Security:** Compared to HTML5-style approaches which expose hardware by extending the JavaScript interpreter, Gibraltar has a much smaller attack surface. Gibraltar's HTTP protocol is a narrow waist for hardware accesses, and the device server is much simpler than a full-blown web browser; for example, our device server for Android phones is only 7613 lines of strongly typed Java code, instead of the million-plus lines of C++ code found in popular web browsers. Using capability tokens and sensor widgets, Gibraltar can also prevent (or at least detect) many attacks from malicious web pages and browsers. HTML5 cannot stop or detect any of these attacks.

- **Usability:** An HTTP device protocol provides a uniform naming scheme for disparate devices and makes it easy for pages to access non-local devices. For example, a page running on a user's desktop machine may want to interact with sensors on the user's mobile phone. If a Gibraltar device server runs on the phone, the page can access the remote hardware using the same interface

that it uses for local hardware—the only difference is that the device server is no longer in the localhost domain.

- **Backwards Compatibility:** It is straightforward to map HTML5 device commands to Gibraltar calls. Thus, to run a preexisting HTML5 application atop Gibraltar, a developer can simply include a translation library that converts HTML5 calls to Gibraltar calls and preserves Gibraltar's security advantages. The library can use Mugshot-style interpositioning [MHE10] to intercept the HTML5 calls.

Since Gibraltar uses HTTP to transport hardware data, a key question is whether this channel has sufficient bandwidth and responsiveness to support real device-driven applications. To answer this question, we wrote a device server for Android mobile phones, and modified four non-trivial applications to use the Gibraltar API. Our evaluation shows that Gibraltar is fast enough to support real-time programs like games that require efficient access to hardware data.

4.2 Design

Gibraltar uses privilege separation [PFH03] to provide a web page with hardware access. The web page, and the enclosing browser which executes the page's code, are both untrusted. Gibraltar places the browser in a sandbox which prevents direct access to Gibraltar-mediated devices. The small, native code device server resides in a separate process from the browser, and executes hardware requests on behalf of the page, exchanging data with the page via HTTP.

As shown in Figure 4.1, a Gibraltar-enabled page includes a JavaScript file called `hardware.js`. This library implements the public Gibraltar API. The component `hardware.js` fetches authentication tokens as described in Section 4.2.1, and translates page-initiated hardware requests into AJAX fetches as described in Section 4.3. `hardware.js` also receives and deserializes the responses. Note that `hardware.js` is merely a convenience library that makes it easier to program against Gibraltar's raw AJAX protocol; Gibraltar does not trust `hardware.js`,

and it does not rely on `hardware.js` to enforce the security properties described in Section 4.5.

Note that the device server resides in the localhost domain, whereas the Gibraltar-enabled page emanates from a different, external origin. By default, the same-origin policy would prevent the `hardware.js` in the web page from fetching cross-origin data from the localhost server. However, by using the HTTP header `Access-Control-Allow-Origin` [Wor08], the device server can instruct the browser to allow the cross-origin Gibraltar fetches. This header is supported by modern browsers like IE9 and Firefox 4+. In older browsers, `hardware.js` communicates with the device server using an invisible frame with a localhost origin; this frame exchanges Gibraltar data with the regular application frame using `postMessage()`. Similarly, Gibraltar can use a remote-origin frame to deal with off-platform devices.

4.2.1 Authenticating Hardware Requests

In Gibraltar, device management consists of three tasks: manifest authorization, session establishment, and session teardown. Figure 4.2 provides the relevant pseudocode in the device server. We discuss this code in more detail below.

Manifest authorization: On mobile devices like Android, users authorize individual applications to access specific hardware devices. Similarly, in Gibraltar, users authorize individual web domains like `cnn.com` to access individual hardware devices. When a page contacts the device server for the first time, the page includes a *device manifest* in its HTTP request. The manifest is simply a list of devices that the page wishes to access. The device server presents this manifest to the user and asks whether she wishes to grant the specified access permissions to the page's domain. If so, the device server stores these permissions in a database. Subsequent page requests for devices in the manifest will not require explicit user action, but if the page requests access to a new device, the user must approve the new permission.

Session management: Since Gibraltar hardware requests are expressed as HTTP fetches, a natural way for the device server to authenticate a request is to inspect its referrer field [FGM+99]. This is a standard HTTP field which indicates the URL (and thus the domain) of the page which generated the request. Unfortunately, a misbehaving browser can subvert this authentication scheme by examining which domains successfully receive hardware data, and then generating fake requests containing these snooped referrer fields. This is essentially a replay attack on a weak authenticator.

To prevent these replay attacks, the device server grants a *capability token* to each authorized web domain. Before a page in domain `trusted.com` can access hardware, it must send a session establishment message to the device server. The device server examines the referrer of the HTTP message and checks whether the domain has already been granted a token. If not[1], the server generates a unique token, stores the mapping between the domain and that token, and sends the token to the page. Later, when the page sends an actual hardware request, it includes the capability token in its AJAX message. If the token does not match the mapping found in the device server's table, the device server ignores the hardware request.

A page sends a session teardown message to the device server when it no longer needs to access hardware, e.g., because the user wants to navigate to a different page. Upon receipt of the teardown message, the server deletes the relevant domain/token mapping. `hardware.js` can detect when a page is about to unload by registering a handler for the JavaScript `unload` event.

Sensor widgets: Given the capability scheme, a misbehaving browser that can only spoof referrers cannot fraudulently access hardware—the browser must also steal another domain's token or retrieve a new one from the device server. As we show in Section 4.5, cross-domain token stealing is difficult in modern browsers

[1] We restrict each domain to a single token for security reasons that we describe in Section 4.5.1. However, this restriction does not prevent a domain from opening multiple device-aware web pages on a client—the pages can inform each other of the domain's token using the JavaScript `postMessage()` API.

which use memory isolation to protect domains. However, nothing prevents a browser from autonomously downloading a new security token in the background under the guise of an authorized domain, and then using this token in its AJAX requests. To prevent this attack, we use sensor widgets [HS10], which are ambient GUI elements like system tray icons that glow, make a noise, or otherwise indicate when a particular hardware device is in use. Sensor widgets also indicate the domains which are currently accessing hardware. Thus, if the browser tries to autonomously access hardware using a valid token, the activity will trigger the sensor widgets, alerting the user to a hardware request that she did not initiate.

The sensor widgets are implemented within the device server, not the browser. However, the browser can try to elude the widgets in several ways. In Section 4.5, we provide a fuller analysis of Gibraltar's security properties.

4.2.2 The Gibraltar API

Figure 4.3 lists the client-side Gibraltar API. Before a web page can issue hardware commands, it must get a new capability token via `createSession()`. Then, it must send its device manifest to the device server via `requestAccess()`. The device server presents the manifest to the user and asks her to validate the requested hardware permissions.

Sensor API

To provide access to sensors like cameras, accelerometers, and GPS units, Gibraltar provides a one-shot query interface and a continuous query interface. In keeping with JavaScript's event-driven programming model, `singleQuery()` and `continuousQuery()` accept an application-defined callback which Gibraltar invokes when the hardware data has arrived. The functions also accept the name of the device to query, and a device-specific `params` value which controls sensor-specific properties like the audio sampling bitrate. `continuousQuery()` takes an additional parameter representing the query frequency.

Different devices will define different formats for the `params` object, and different formats for the returned device data. However, much like USB devices,

Gibraltar devices fall into a small set of well-defined classes such as storage devices, audio devices, and video devices. Thus, web pages can program against generic Gibraltar interfaces to each class; the device server and `hardware.js` can encapsulate any device-specific eccentricities.

Figure 4.3 also describes a sensor management interface. The power controls allow a page to shut off devices that it does not need; the device server ensures that a device is left on if at least one application still needs it. `sensorAdded()` and `sensorRemoved()` let applications register callbacks which Gibraltar fires when devices arrive or leave. These events are useful for off-platform devices like Bluetooth headsets and Nike+ shoe sensors [Nik].

Processor API

Multi-core processors and programmable GPUs are already available on desktops, and they are starting to ship on mobile devices. To let web pages access these extra cores, Gibraltar exports a simple multi-processor computing model inspired by OpenCL [Khr], a new specification for programming heterogeneous processors.

A Gibraltar *kernel* represents a computational task to run on a core. Kernels are restricted to executing two types of predefined functions. *Primitive functions* are geometric, trigonometric, or comparator operations. Gibraltar's primitive functions are similar to those of OpenCL. *Built-in functions* are higher-level functions that we have identified as particularly useful for processing hardware data. Examples of such functions are FFT transforms and matrix operations.

A web page passes a kernel to Gibraltar by calling `enqueueKernel()`. To execute a parallel vector computation, the page calls `setKernelData()` with a vector of arguments; Gibraltar will instantiate a new copy of the kernel for each argument and run the kernels in parallel. A web page can also create a computation pipeline by calling `enqueueKernel()` multiple times with the same or different kernel. Gibraltar will chain the kernels' inputs and outputs in the order that the kernels were passed to `enqueueKernel()`. The page sets the input data for the pipeline by passing a scalar value to `setKernelData()`.

Once an application has configured its kernels, it calls `executeKernels()` to start the computation. Gibraltar distributes the kernels to the various cores in the system, coordinates cross-kernel communication, and fires an application-provided callback when the computation finishes.

Storage API

The final set of calls in Figure 4.3 provide a key/value storage interface. The namespace is partitioned by web domain and by storage device; a web domain can only access data that resides in its partitions. To support removable storage devices, Gibraltar fires connection and disconnection events like it does for off-platform sensors like Bluetooth headsets.

HTML5 DOM storage [Hicb] also provides a key-value store. However, DOM storage is limited to non-removable media, and it does not explicitly expose the individual devices which are used for the underlying stable storage.

4.2.3 Remote device access

As we mentioned earlier, some devices may reside off-platform. If those devices run a Gibraltar server which accepts external connections, a web page can seamlessly access those devices using the same interface that it uses for local ones. This capability enables many interesting applications. For example, in Section 4.6, we evaluate a game that runs on a desktop machine but uses a mobile phone with an accelerometer as a motion-sensitive controller. In this example, the web page runs on the desktop machine, but the device server runs on the phone.

A device server accepts connections from localhost clients by default (subject to the authentication rules described in Section 4.2.1). For security reasons, a device server should reject connections from arbitrary remote clients. Thus, users must explicitly whitelist each external IP address or dynamic DNS name [Vix97] that wishes to communicate with a device server. This is accomplished in a fashion similar to how the user authorizes device manifests (§4.2.1).

4.2.4 Sandboxing the Browser

Gibraltar is agnostic about the mechanism that prevents the browser from accessing Gibraltar devices. For example, mobile platforms like Android, iOS, and the Windows Phone already provide a device ACL infrastructure that makes it easy to prohibit applications from accessing forbidden hardware. However, Gibraltar is compatible with other isolation techniques like hardware virtualization or binary rewriting.

4.3 Implementation

Client-side Library: `hardware.js` encodes device requests using a simple XML string. Each request contains a security token, an action to perform, the target device, and optional device-specific parameters. For example, a request to record microphone data includes a parameter which represents the recording duration. Device responses are also encoded using XML. The response specifies whether the request succeeded, and any data associated with the operation. The device server encodes binary data in `Base64` format so that `hardware.js` can represent data as JavaScript strings.

Android Device Server: On Android 2.2, we implemented the device manager as a servlet for the i-jetty web server [Bea]. A servlet is a Java software module that a web server invokes to handle certain URL requests. The Gibraltar servlet handles all requests for Gibraltar device URLs. The servlet performs the authentication checks described in Section 4.2.1, accesses hardware using native code, and returns the serialized results. We refer to our Android implementation of Gibraltar as GibDroid.

The GibDroid device server has different probing policies for low throughput sensors and high throughput sensors. For low throughput devices like cameras, GibDroid accesses the sensor on demand. For devices like accelerometers that have a high data rate, the GibDroid server continuously pulls data into a circular buffer. When a page queries the sensor, the device server returns the entire buffer, allow-

ing multiple data points to be fetched in a single HTTP round trip. Currently, GibDroid provides access to accelerometers, GPS units, cameras (both single pictures and streaming video), microphones, local storage, and native computation kernels.

Before a web page can receive hardware data from the device server, it must engage in a TCP handshake with the server and send an HTTP header. For devices with high data rates like accelerometers and video cameras, creating an HTTP session for each data request can hurt performance, even with sample batching. Thus, GibDroid allows the device server to use Comet-style [CM08] data pushes. In this approach, `hardware.js` establishes a persistent HTTP connection with the device server using a "forever frame." Unlike a standard frame, whose HTML size is declared in the HTTP response by the server, a forever frame has an indefinite size, and the server loads it incrementally, immediately pushing a new HTML chunk whenever new device data arrives. Each chunk is a dynamically generated piece of JavaScript code; the code contains a string variable representing the new hardware data, and a function call which invokes an application-defined handler. Forever frames are widely supported by desktop browsers, but currently unsupported by many mobile browsers. Thus, the device server reverts to request-response for mobile browsers.

GibDroid can stream accelerometer data and video frames using Comet data pushes. To handle video, our current implementation uses data URIs [Mas98] to write `Base64`-encoded data to an `<image>` tag [2]. Many current browsers limit data URIs to 32 KB data; thus, data URIs are only appropriate for sending previews of larger video images. Our current GibDroid implementation displays video frames with a pixel resolution of 530 by 380. The device server uses Android's `setPreviewCallback()` call to access preview-quality images from the underlying video camera.

GibDroid supports the execution of kernel functions. However, our evaluation Android phone does not have secondary processing cores. Therefore, GibDroid kernels run in separate Java threads that time-slice the single processor with other

[2] We eventually plan to make `hardware.js` write video frames to the bitmap of an HTML5 `Canvas` object [Hica].

applications.

As shown in Figure 4.4, GibDroid places sensor widgets in the standard notification bar that exists in all Android phones. The notification bar is a convenient place to put the widgets because users are already accustomed to periodically scanning this area for program updates. We are still experimenting with the visual presentation for the widgets, so Figure 4.4 represents a work-in-progress.

Windows PC Device Server: We also wrote a device server for Windows PCs. This device server, written in C#, currently only provides access to the hard disk, but it is the target of active development. In Section 4.6.1, we use this device server to compare Gibraltar's performance on a multi-core machine to that of HTML5.

4.4 Applications

In this section, we describe four new applications which use the Gibraltar API to access hardware. We evaluate the performance of these applications in Section 4.6.

Our first application is a mapping tool similar to MapQuest [Map]. This web page uses GPS data to determine the user's current location. It also uses Gibraltar's storage APIs to load cached maps tiles. More specifically, we assume that the phone's operating system prefetches map tiles, similar to how the Songo framework prefetches mobile ads [KLLB]. The operating system stores the map tiles in the file system; for each cached tile, the OS adds the key/value pair `(tileId,fileSystemLocation)` to the mapping application's Gibraltar storage area. When the map application loads, it determines the user's current location and calculates the set of tiles to fetch. For each tile, it consults the tile index in Gibraltar storage to determine if the tile resides locally. If it does, the page loads the tile using an `<img>` tag with a `file://` origin; otherwise, the page uses a `http://` origin to fetch the image from the remote tile server.

The popular native phone application *Shazam!* identifies songs that are playing in the user's current environment. *Shazam!* does this by capturing micro-

phone data and applying audio fingerprint algorithms. Inspired by *Shazam!*, we built Gibraltar Sound, a web application that captures a short sound clip and classifies it as *music*, *conversation*, *typing*, or *other ambient sound*. To classify sounds, we used Mel-frequency cepstrums (MFCC) for feature extraction, and Gaussian Mixture Models (GMM) for inference [LPL+09]. We implemented MFCC and GMM as native built-in kernels.

Our final applications leverage Gibraltar's ability to access off-platform devices. These pages load on a desktop machine's browser, but use Gibraltar to turn a mobile phone into a game controller. The first application, Gibraltar Paint, is a simple painting program in which user gestures with the phone are converted into brush strokes on a virtual canvas. Gestures are detected using the phone's accelerometer.

We also modified a JavaScript version of Pacman [Lan] to use a Gibraltar-enabled phone as a controller for a game loaded on the desktop browser—tilting the phone in a direction will cause Pacman to move in that direction. HTML5 cannot support the latter two applications because it lacks an API for remote device access.

4.5 Security

Any mechanism for providing hardware data to web pages must grapple with two questions. First, can it ensure that each device request was initiated by the user instead of a misbehaving browser? Second, once the hardware data has been delivered to browser, can the system prevent the browser from modifying or leaking that data in unauthorized ways? Gibraltar only addresses the first question, but it is complementary to systems that address the second. In Section 4.5.1, we describe the situations in which Gibraltar can and cannot prevent fraudulent hardware access. In Section 4.5.2, we describe how Gibraltar can be integrated with a taint tracking system to minimize unintended data leakage.

4.5.1 Authenticating Hardware Requests

In Gibraltar, there are five kinds of security principals: the user, the Gibraltar device server, the underlying operating system, web pages, and the web browser. Gibraltar does not trust the last two principals. More specifically, Gibraltar's security goal is to prevent unauthorized web pages from accessing hardware data, and faulty web browsers from autonomously fetching such data. Gibraltar assumes that the OS properly sandboxes the browser, preventing the browser from directly accessing Gibraltar-mediated hardware; Gibraltar is agnostic to the particular sandboxing mechanism that is used, e.g., binary rewriting, virtual machines, or OS-enforced device ACLs provided by platforms like Android and iOS. Gibraltar assumes that the device server is implemented correctly, that the user can inform the device server of authorized web sites without interference, and that the operating system prevents the web browser from directly tampering with the device server. Thus, the only way that a faulty web page or browser can access hardware is by subverting the AJAX device protocol.

As shown in Figure 4.2, the device server will only respond to a hardware request if the request has an authorized referrer field and a valid authentication token; furthermore, the authorized domain cannot have another open session involving a different token. Thus, Gibraltar's security with respect to device D can be evaluated in the context of three parameters: whether the attacker can fake referrer fields, whether the attacker can steal tokens from domains authorized to access D, and whether the user currently has a legitimate, active frame belonging to a legitimately authorized domain.

Gibraltar assumes that the operating system correctly routes packets to the device server. Thus, the device server can reject arbitrary connections from off-platform entities by verifying that the source in each AJAX request has a localhost IP address. If a user wants to associate a device server with a web page that resides off-platform, she must whitelist the external IP address, or notify the device server and the web page of a shared secret which enables the device server to detect trusted external clients. For example, the client web page might generate a random number and include this number in the first AJAX request that it sends

to the device server. When the server receives this request, it can present the nonce to the user for verification.

4.5.2 Securing Returned Device Data

The browser acts as the conduit for all AJAX exchanges, and it can arbitrarily inspect the JavaScript runtimes inside each page. Thus, once the browser has received hardware data (either because a user legitimately fetched it, or because the browser stole/fetched a token and acquired the data itself), neither Gibraltar nor HTML5 can prevent the browser from arbitrarily inspecting, modifying, or disseminating the data.

Suppose that, through clever engineering, the browser cannot be subverted by malicious web pages. Further suppose that the browser is trusted not to fake referrer fields, steal tokens from authorized domains, or otherwise subvert the Gibraltar access controls. Even in these situations, malicious web pages can still leak hardware data to remote servers. For example, suppose that the user has authorized domain `x.com` to access hardware, but not `y.com`. The same-origin policy ostensibly prevents JavaScript running on `http://x.com/index.html` from sending data to `y.com`'s domain. However, this security policy is easily circumvented in practice. For example, the JavaScript in `x.com`'s page can read the user's GPS data and create an iframe with a URL like `http://y.com/page.index?lat=LAT_DATA long=LON_DATA`. By loading the frame, the browser implicitly sends the GPS data to `y.com`'s web server.

If the browser is trusted, it can prevent such leakage by tracking the information flow between Gibraltar AJAX requests and externalized data objects like iframe URLs. This is similar to what TaintDroid [EGgC+10] does, although TaintDroid tracks data flow through a Java VM instead of a browser.

If the browser is untrusted, we can place the taint tracking infrastructure outside of the browser, e.g., in the underlying operating system. However, regardless of where the taint tracker resides, it must allow the user to whitelist certain pairs of domains and hardware data. For example, suppose that the user has authorized only `x.com` to access the GPS unit. Whenever the data flow system de-

tects that GPS data is about to hit the network, it must ensure that the endpoint resides in `x.com`'s domain, e.g., by doing a reverse DNS lookup on the endpoint's IP address.

4.6 Evaluation

In this section, we ask two fundamental questions about Gibraltar's performance. First, is an HTTP channel fast enough to support high frequency sensors and interactive applications? Second, is Gibraltar competitive with HTML5 in terms of performance?

As described in Section 4.3, we wrote device servers for two platforms. The first server runs on Android 2.2 phones, and we tested it on two handsets: a Nexus One with 512 MB of RAM and a 1 GHz Qualcomm Snapdragon processor, and a Droid X with 512 MB of RAM and a 1 GHz Texas Instruments OMAP processor. We also wrote a device server for Windows PCs. We tested that server on a Windows 7 machine with 4 GB of RAM and an Intel Core2 processor with two 2.66 GHz cores.

4.6.1 Access Latency

Multi-core machines: We define a device's *access latency* as the amount of time that a client perceives a synchronous device operation to take. Figure 4.5 shows access latencies for the hard disk on the dual-core desktop machine. Each bar represents the average of 250 trials, with each read or write involving 1 KB of data. HTML5 disk accesses were implemented using the DOM storage API [Hicb], whereas Gibraltar disk accesses were handled by the device server and accessed a partitioned region of the local file system owned by the device server. All reads targeted prior write addresses, meaning that the reads should hit in the block cache inside the device server or the HTML5 browser.

The absolute latencies for Gibraltar's disk accesses are small on both Firefox 3.6 and IE8. For example, a Gibraltar-enabled page on IE8 can read 1 KB of data with a latency of 0.62 ms; on Firefox, the page can perform a similar read with 2.58

ms of latency. While Gibraltar's read performance is worse than that of HTML5, it is more than sufficient to support common use cases for local storage, such as caching user data to avoid fetching it over a slow network.

For disk writes on both browsers, Gibraltar is more than five times faster than HTML5. This is because the Gibraltar device server asynchronously writes back data, whereas Firefox and IE have a write-through policy. Switching Gibraltar to a write-through policy would result in similar performance to HTML5, since the primary overhead would be mechanical disk latencies, not HTTP overhead.

Single-core machines: Our desktop machine had a dual-core processor, meaning that the device server and the web browser rarely had to contend for a core. In particular, once the device server had invoked a `send()` system call to transfer device data to the browser, the OS could usually swap the browser immediately onto one of the two cores. On a single core machine, the browser might have to wait for a non-trivial amount of time, since multiple processes besides the browser are competing for a single core.

Figure 4.6 depicts access latencies to the Null device on the Droid X phone (the Null device immediately returns an empty message). We use `setsocketopt()` to disable the TCP Nagle algorithm, we prod TCP into sending small packets immediately instead of trying to aggregate several small packets into one large one. This decreases the average access latency from 87 ms to 78 ms, and the standard deviation from 34 ms to 25 ms. By raising the priority of the device server thread and the receiving browser thread, we can further decrease the latency to 67 ± 18 ms. However, the raw performance is still worse than in the dual-core case due to scheduling jitter. For example, looking at the results for individual trials, we saw access latencies as low as 29 ms, and as high as 144 ms.

Multi-core processors are already pervasive on desktop systems, and new mobile phones and tablets like the LG Optimus 2X have dual-core processors. Thus, we expect that scheduling jitter will soon become a non-issue for Gibraltar. In the rest of this section, we provide additional evaluation results using the single-core Nexus One phone. We show that even on a single-core machine, Gibraltar is

already fast enough to support interactive applications.

Accessing Sensors on the Nexus One: Figure 4.7 depicts the access latency for various devices on the Nexus One phone. The accelerometer and the GPS unit are the sensors that applications query at the fastest rate. Figure 4.7 shows that the accelerometer can be queried 9.4 times a second, and the GPS unit can be queried 6.6 times a second. As we discuss in Section 4.6.4, these sampling rates are sufficient to support games and interactive mapping applications.

Accessing the camera or the microphone through Gibraltar is much more expensive than accessing the accelerometer. However, most of the latency arises from the inherently expensive initialization costs for those devices. GibDroid adds 160 ms to the inherent cost of sampling 10 seconds of audio data, and 560 ms to the inherent cost of taking a picture. In both cases, the bulk of the Gibraltar overhead came from the `Base64` encoding that the device server must perform before it can send binary data to the application.

The results in Figure 4.7 used the request-response version of the Gibraltar protocol. On browsers that support forever frames (§4.3), Gibraltar can use server-push techniques to decrease client-perceived access latencies to devices. Figure 4.8 quantifies this improvement for desktop browsers accessing phone hardware over a wireless connection. For example, for video on Firefox, frame access latencies decreased from 126 ms to 83 ms; this improved the streaming rate for live video from 8 frames per second to 12. For the accelerometer, access latencies decreased from 173 ms to 22 ms, allowing the client to fetch accelerometer readings at a rate of 45 Hz. This was close to the native hardware limit of 50 Hz. Note, however, that the performance gains in both cases arose not just from the server-push technique, but from the fact that the device server and the web browser ran on different machines (and thus different processors). This ameliorated some of the scheduling jitter that arises when the device server and the browser run on the same core.

4.6.2 Sampling Throughput

Low access latencies improve the freshness of the data that the client receives. However, the client may still be unable to receive data at the native sampling frequency. Thus, the device server continuously gathers information from high data rate devices like the accelerometer and the GPS unit. When the server gets a read request for such a device, it returns all of the data that has accumulated since the last query. Thus, an application can analyze the entire data stream even if it cannot access every sample at the native data rate.

Figure 4.9 depicts GibDroid's sampling throughput using the built-in Android browser to access phone hardware. Each bar represents the maximum number of data samples accessible per second to a native application, a Gibraltar page using an inner iframe, and a Gibraltar page in which the outer iframe directly issues AJAX requests. Throughput degradation is less than 5% for all devices. Figure 4.9 also shows that cross-frame `postMessage()` overhead was minimal. Note that the accelerometer throughput was greater than the Null throughput because GibDroid batched multiple accelerometer samples per HTTP response.

4.6.3 Power

On mobile devices, minimizing power consumption is extremely important. To measure Gibraltar's impact on battery life, we attached a Monsoon Power Monitor [Mon] to the Nexus One. Due to space constraints, we elide a full evaluation of Gibraltar's power usage. Instead, we simply note that the energy cost for using a Gibraltar-enabled page (i.e., the cost for running the browser and the Gibraltar device server) is similar to the cost of running any other native application. For example, the cost of making a phone call is 773 milliwatts, whereas the cost of running a Gibraltar-enabled page that accesses the accelerometer is 803 milliwatts.

4.6.4 Applications

For the final part of our evaluation, we examine the performance of the four Gibraltar-enabled applications that we described in Section 4.4. We evaluated all

four applications on the GibDroid platform.

Our map application took an average of 64 ms to load a cached map tile, but 372 ms to fetch one from the Internet. This result is not surprising, since accessing local storage should be faster than pulling data across the wide area.

For our audio classification application, the key performance metric is how long the classification takes. For a 52 KB WAV file representing 10 seconds of data, feature extraction took approximately 6 seconds, and classification of the result took 1.5 seconds. These experiments used a JavaScript implementation of the classification algorithms. For larger audio files, the application could use Gibraltar's native computation kernels to boost performance.

We evaluated Paint and Pacman by running them on a Chrome desktop browser which communicated with GibDroid through a USB cable. Paint was able to sense 9.83 motions per second; this number is an application-level latency that includes the Gibraltar access latency and the overhead of updating the HTML `Canvas` object. Pacman had similar performance. In both cases, the phone was able to control the application with no user-perceived delay. We plan to run further tests over a wireless network which allows the phone to be untethered from the desktop.

4.7 Related Work

In Section 4.1, we described the disadvantages of using native code plugins like Flash to provide hardware access to web pages. We also described why HTML5 is a step in the right direction, but not a complete solution.

Like Gibraltar, Maverick [RG11] provides web pages with hardware access. Maverick lets web developers write USB drivers using JavaScript or NaCl. Maverick sandboxes each untrusted page and USB driver; the components exchange messages through the trusted Maverick kernel. Maverick differs from Gibraltar in three key ways. First, Maverick is limited to the USB interface, whereas Gibraltar's client-side JavaScript library can layer arbitrary hardware protocols atop HTTP. Second, unlike USB, HTTP also provides straightforward support for off-platform

devices. Third, Maverick does not have a mechanism like sensor widgets that detects misbehaving applications. Thus, Maverick cannot prevent buggy or malicious pages from using the driver infrastructure in ways that the user did not intend. Maverick does have better performance than the current implementation of Gibraltar since Maverick provides IPC via NaCl native code mechanisms instead of via standard HTTP over TCP. However, with kernel support for fastpath localhost-to-localhost TCP connections, and/or NIC support for offloading TCP-related computations to hardware, we believe that Gibraltars performance can approach that of Maverick.

PhoneGap [Pho] is a framework for building cross-platform, hardware-aware mobile applications. A PhoneGap application consists of JavaScript, HTML, CSS, and a bundled chrome-less browser whose JavaScript runtime has been extended to export hardware interfaces. Like Gibraltar, PhoneGap allows developers to write device-aware applications using the traditional web stack. Compared to Gibraltar, PhoneGap has three limitations. First, PhoneGap's hardware interface is philosophically equivalent to the HTML5 interface, and thus has similar drawbacks with respect to interface and security. Second, a PhoneGap program is a native application and must be explicitly installed, unlike a Gibraltar web page. Third, PhoneGap applications run within the `file://` protocol, not the `http://` protocol. Thus, unlike Gibraltar web pages, PhoneGap programs are not restricted by the same domain policy. This allows a PhoneGap program to load multiple frames from multiple domains and manipulate their data in ways that would fail in the `http://` context and violate the security assumptions of the remote domains.

In Palm's webOS [All09], applications are written in JavaScript, HTML, and CSS. However, these programs are not web applications in the standard sense, because they rely on webOS' special runtime, and they will not execute inside actual web browsers. The webOS runtime is a customized version of the popular WebKit browser engine. It exposes HTML5-style device interfaces to applications, and thus suffers from the problems that we discussed in prior sections.

Microkernel browsers like OP [GTK08] and Gazelle [WGM+09] restructure the browser into multiple untrusted modules that exchange messages through a

small, trusted kernel. Gibraltar's device server is somewhat like a trusted microkernel which mediates hardware access. However, previous microkernel browsers do not change the hardware interface exposed to web pages, since these browsers use off-the-shelf JavaScript runtimes that use the HTML5 interface.

Several projects from the sensor network community expose hardware data using web protocols [DRAPZ04, PKGZ08, YD09]. However, these systems do not address the security challenges involved in authenticating hardware requests that emanate from potentially untrustworthy browsers. Gibraltar also exports a richer interface for device querying and management.

4.8 Conclusions

Gibraltar's key insight is that web pages can access hardware devices by treating them like web servers. Gibraltar sandboxes the browser, shifts authority for device accesses to a small, native code device server, and forces the browser to access hardware via HTTP. Using this privilege separation and sensor widgets, Gibraltar provides better security than HTML5 hardware interfaces; the resulting API is also easier to program against. Experiments show that the HTTP device protocol is fast enough to support real, interactive applications that make frequent hardware accesses.

4.9 Acknowledgements

Chapter 4, in part, is a reprint of the material as it appears in Gibraltar: Exposing Hardware Devices to Web Pages using AJAX. Kaisen Lin, David Chu, James Mickens, Li Zhuang, Feng Zhao, Jian Qiu. In WebApps 2012: 3rd USENIX Conference on Web Application Development, Boston, MA, USA, 2012. The dissertation author was the primary investigator and author of this paper.

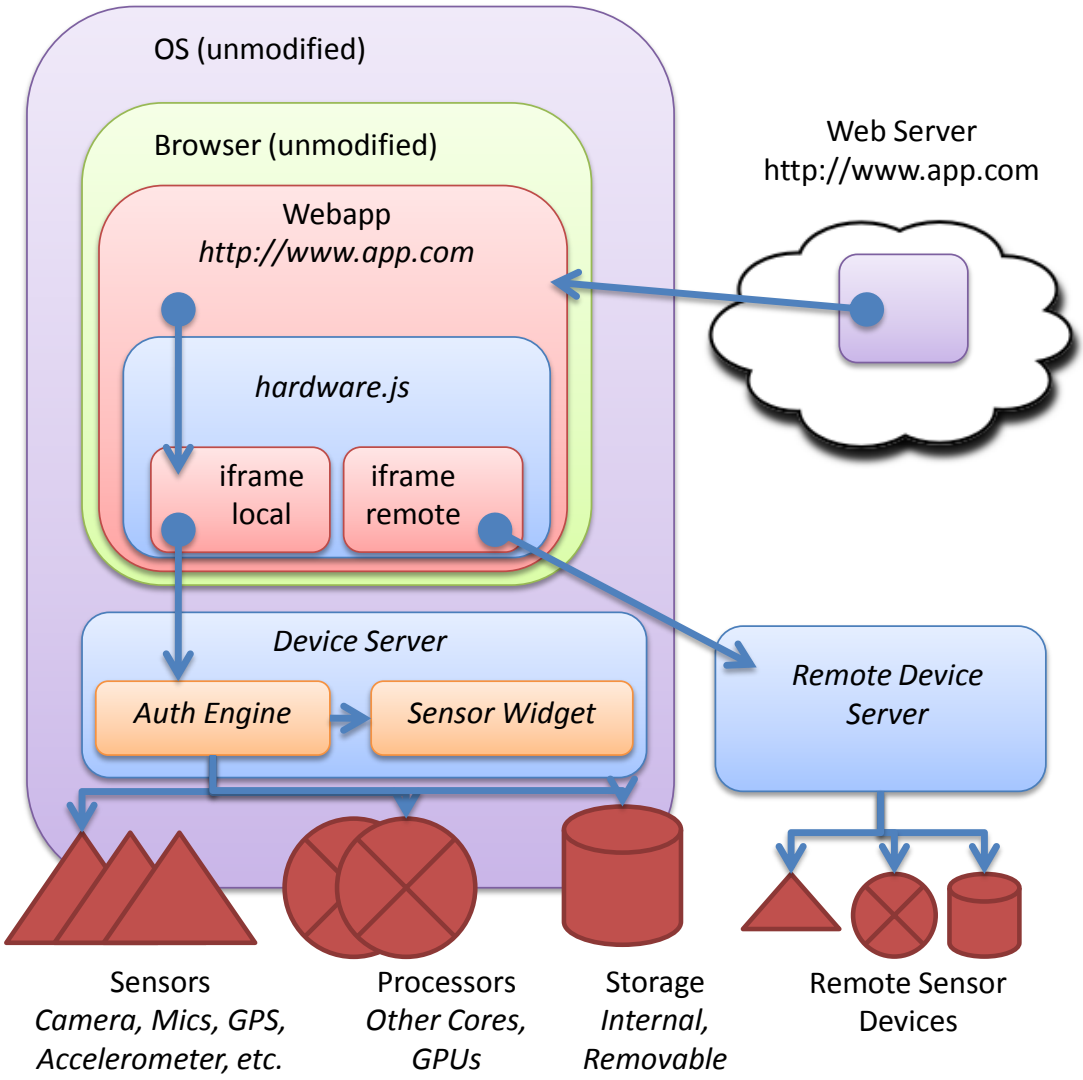


Figure 4.1: Gibraltar Architecture.

```
void handle_request(req){
   resp = new AJAXResponse();
   switch(req.type){
     case OPEN_SESSION:
        if(!active_tokens.contains(req.referrer)){
           resp.result = "TOKEN:" + makeNewToken();
           active_tokens[req.referrer] = resp.result;
        }
        break;

     case DEVICE_CMD:
        if(!authorized_domains[req.device].contains(
                                       req.referrer) ||
           (active_tokens[req.referrer] == null) ||
           (active_tokens[req.referrer] != req.token)){
           resp.result = "ACCESS DENIED";
        }else{
           resp.result = access_hardware(req.device,
                                         req.cmd);
           sensor_widgets.alert(req.referrer,
                                req.device);
        }
        break;

     case CLOSE_SESSION:
        if(active_tokens[req.referrer] == req.token)
            active_tokens.delete(req.referrer);
        break;
   }
   sendResponse(resp);
}
```

Figure 4.2: Pseudocode for device server.

Call	Description
`createSession()`	Get a capability token from the device server.
`destroySession()`	Relinquish a capability token.
`requestAccess(manifest)`	Ask for permission to access certain devices.
`singleQuery(name, params)`	Get a single sensor sample.
`continuousQuery(name, params, period)`	Start periodic fetch of sensor samples.
`startSensor(name)`	Turn on a sensor.
`stopSensor(name)`	Turn off a sensor.
`sensorAdded(name)`	Upcall fired when a sensor is added.
`sensorRemoved(name)`	Upcall fired when a sensor is removed.
`getSensorList()`	Get available sensors.
`enqueueKernel(kernel)`	Queue a computation kernel for execution.
`setKernelData(parameters)`	Set input data for the computation pipeline.
`executeKernels()`	Run the queued kernels on the input data.
`put(storename,key,value)`	Put value by key.
`get(storename,key)`	Get value by key.

Figure 4.3: Summary of `hardware.js` API. All calls implicitly require a security token and callback function.

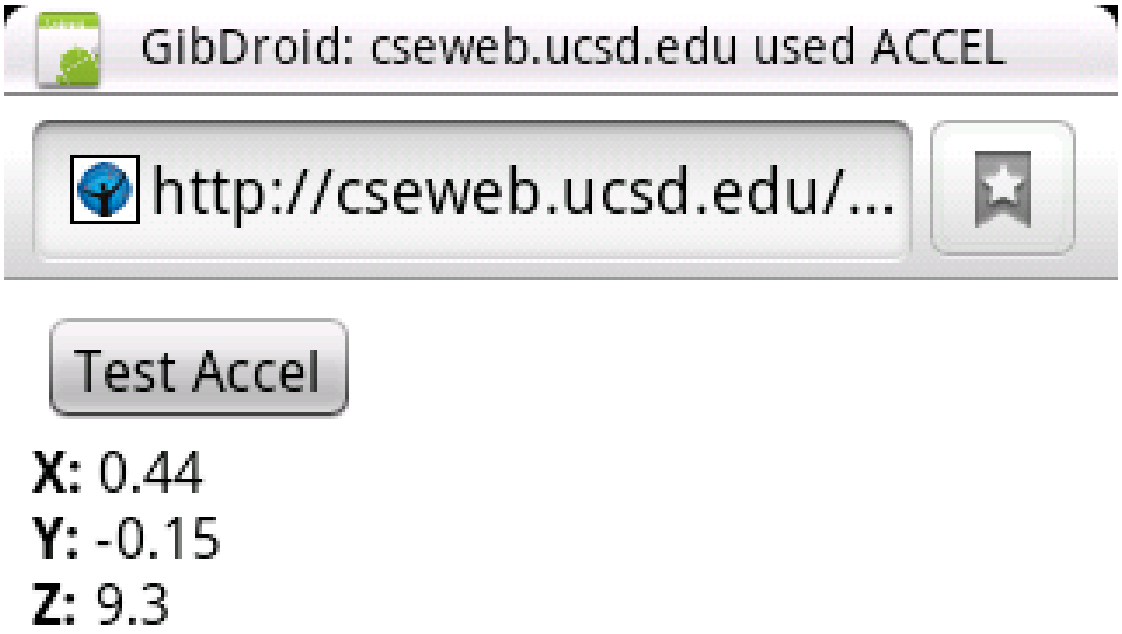


Figure 4.4: GibDroid uses the Android notification bar to hold sensor widgets.

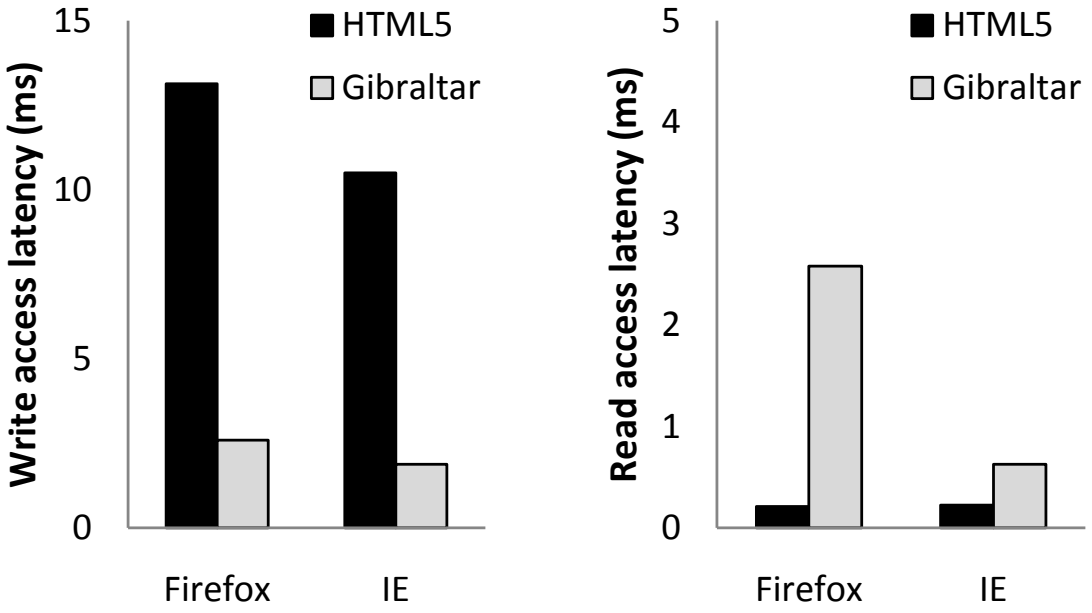


Figure 4.5: Read and write latencies to the hard disk on the dual-core desktop machine.

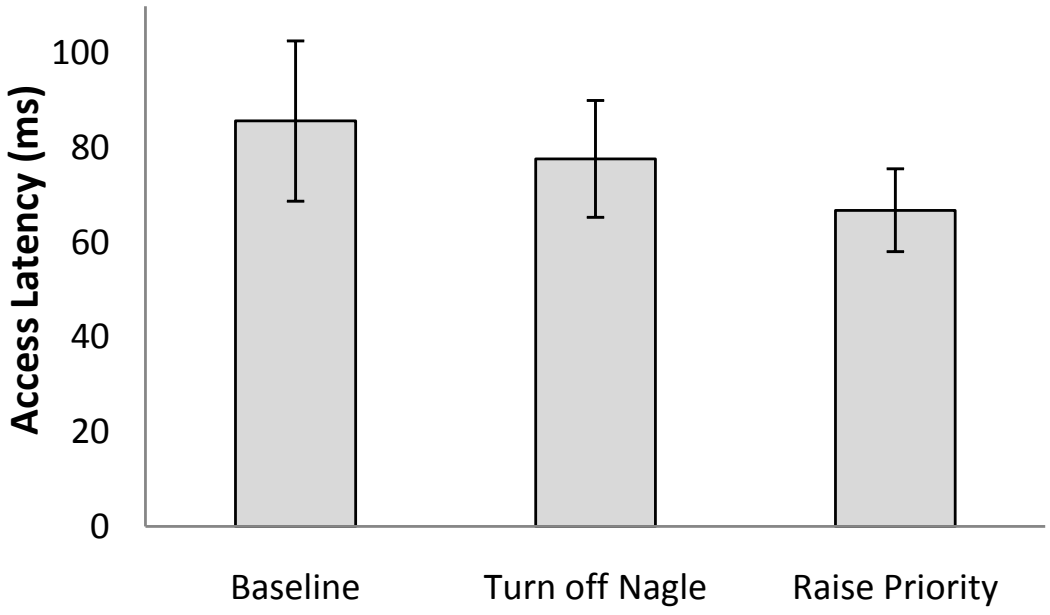


Figure 4.6: GibDroid Null-device access latencies (single-core phone). Error bars represent one standard deviation.

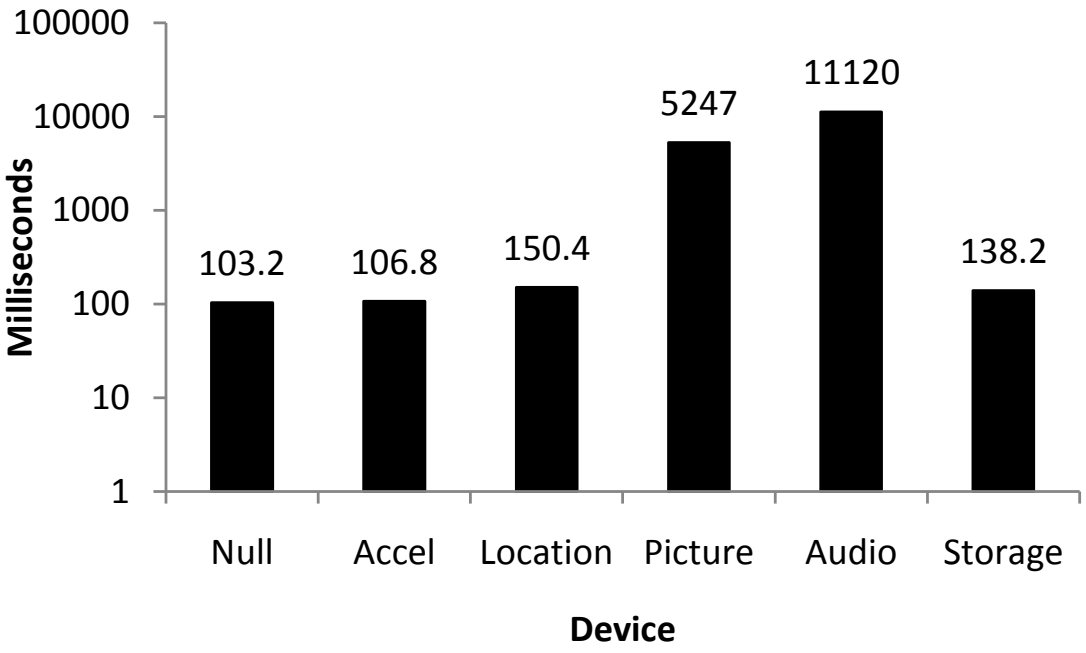


Figure 4.7: Nexus One access latencies (mobile browser accessing local hardware). Note that the y-axis is log-scale.

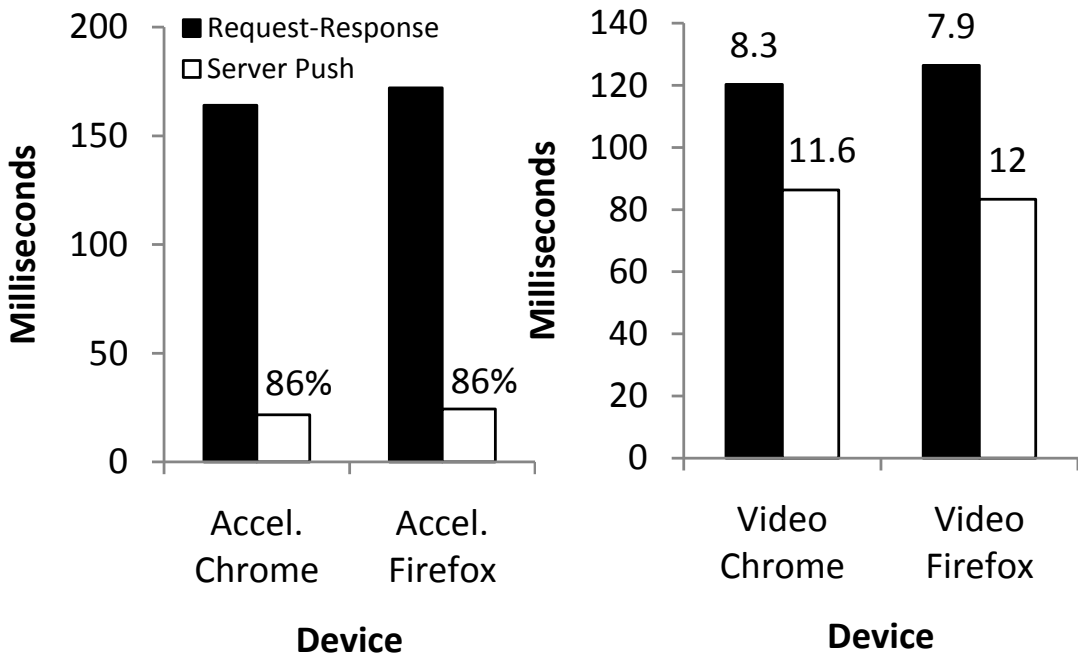


Figure 4.8: Nexus One access latencies (desktop browser accessing phone hardware). Top-bar numbers for accelerometer represent improvements in sample frequency; top-bar numbers for video represent frame rates.

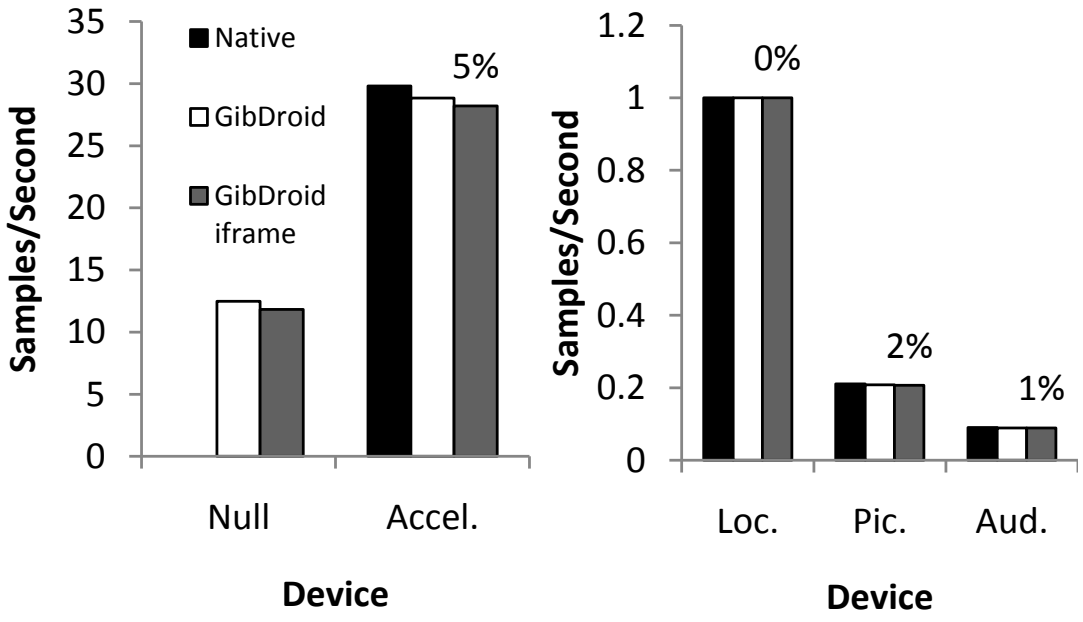


Figure 4.9: GibDroid sampling throughputs. Top-bar fractions show the percentage slowdown of Gibraltar with an iframe compared to native execution.

Chapter 5

Conclusion

Sensor-actuator control systems are no longer just for large enterprises. Cheap wireless sensor networks combined with mobile phones have made commodity control systems a reality in any environment. For example, home thermostats can easily be augmented with space heaters, fans, or air filters. This capability to extend the infrastructure is important because it is often difficult to change fixed infrastructure once it is in place. Furthermore, the research and industry communities have proven that significant gains in efficiency and cost savings can be made even with incremental improvements to the sensing and control infrastructure.

Our overall contribution is that we have created a system that commoditizes sensor-actuation systems, and thus enables ordinary consumers to perform automation in their environment. Through this work, physical sensors and actuators are represented as variables, thus enabling control applications to be written like any other piece of software. This work includes three major subcontributions.

We described how to build a sensor-actuation software platform that is capable of supporting multiple applications on a single deployment through a software API. As a result, applications can be written and executed from any environment, which allows applications to be written in their most natural language. The software platform also includes an environmental model that captures the relations between actuator and sensor variables, thus allowing applications to focus on achieving certain environmental conditions rather than how to do it. We demonstrated how to implement such applications in HVAC and pipeline scenarios, as

well as performed a prototype deployment in a home automation scenario.

We also described a new protocol for dissemination of small data in an ad-hoc wireless sensor-actuator network. This is important for disseminating actuation commands throughout the network. Each piece of data is identified by a unique key and versioned to determine the newest information. It then uses an eventual-consistency epidemic controlled broadcast protocol to propagate hash summaries and bloom filters of the data set. As sensor-actuation nodes exchange summaries, they are able to determine exactly which data items are newer. In a test of over 70 nodes deployed in a building environment, our protocol consumes 20-60% less network, and operates 200% faster compared to naive approaches.

Finally, we detailed how to integrate a mobile phone into a sensor-actuation environment. Not only is it used to communicate information between the sensor-actuator deployment and the environment operator, it is also used as a sensing device and execution platform for JavaScript applications. This allows sensor-actuator applications to be deployed right off the web. A local device web server enables JavaScript applications to acquire access to native phone hardware. Careful security considerations are given to the device server to ensure that websites cannot steal sensitive phone information. We also develop a prototype device server for Android and show that its performance on a Google Nexus One can support streaming video with energy usage comparable to active web browsing.

5.1 Future Work

Many issues remain before a sensor-actuation software platform can be fully adopted. One important issue is being able to reuse applications on multiple deployments. This has the potential to catalyze adoption by allowing operators to browse through numerous applications and select the ones that are relevant. Currently, an application must know what specific variables are available for a particular deployment.

Security has always been an issue with control systems as witnessed by Stuxnet, and will become even more important as control systems are more widely

adopted. Privacy, authenticity, and availability are all important issues when using control systems so that environment operators have trust in the system. However, application writers need access to real-time data and they indirectly control actuation devices. This means that potentially sensitive sensor data must leave the deployment and any actuation decisions must be safe for the deployment.

Another aspect that we have not considered is usability. Important questions include: How difficult would it be to maintain the extra actuation devices? Would operators be open to manually performing actuation? What kind of problems might a deployment run into when deployed long-term? These are all issues that might arise when deploying in a live environment, and this dissertation aims to elicit research and discussion in these areas.

Bibliography

[ABG+11] Yuvraj Agarwal, Bharathan Balaji, Rajesh Gupta, Seemanta Dutta, and Thomas Weng. Duty-cycling buildings aggressively: The next frontier in hvac control. In *IPSN '11: Proceedings of the 10th international conference Information Processing in Sensor Networks*, 2011.

[All09] Mitch Allen. *Palm webOS: The Insider's Guide to Developing Applications in JavaScript using the Palm Mojo Framework*. O'Reilly Media, Sebastopol, CA, 2009.

[apa] Apache XML-RPC. http://ws.apache.org/xmlrpc/.

[Ash09] Kevin Ashton. That 'internet of things' thing. *RFID Journal*, 2009.

[Ass04] American Water Works Association. *Recommended Practice for Backflow Prevention and Cross-Connection Control*. American Water Works Association, 2004.

[Axi] Axis Communications. Axis 214 PTZ Network Camera. http://www.axis.com/files/manuals/um_214_37546_en_0911.pdf.

[Bea] Jan Bartel and et. al. i-Jetty: Webserver for the Android mobile platform. http://code.google.com/p/i-jetty.

[BLW+11] Rahul Balani, Kaisen Lin, Lucas F Wanner, Jonathan Friedman, Rajesh K Gupta, and Mani B Srivastava. Programming support for distributed optimization and control in cyber-physical systems. In *Proceedings of the Second International Conference on Cyber-Physical Systems (ICCPS 2011*, April 2011.

[BM04] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. 1(4):485–509, 2004.

[CAPMN03] Francisco Matias Cuenca-Acuna, Christopher Peery, Richard P. Martin, and Thu D. Nguyen. Planetp: Using gossiping to build content addressable peer-to-peer information sharing communities. In

HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, page 236, Washington, DC, USA, 2003. IEEE Computer Society.

[CBA+05] B. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. Parkes, J. Shneidman, A. Snoeren, and A. Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors (EmNets)*, 2005.

[CDBF04] Cash J. Costello, Christopher P. Diehl, Amit Banerjee, and Hesky Fisher. Scheduling an active camera to observe people. In *Proceedings of the ACM 2nd international workshop on Video surveillance & sensor networks*, VSSN '04, pages 39–45, New York, NY, USA, 2004. ACM.

[CM08] David Crane and Phil McCarthy. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0.* Apress, New York, NY, 2008.

[Cro] Crossbow, Inc. Mote in network programming user reference. http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM Press, 1987.

[DGM05] Amol Deshpande, Carlos Guestrin, and Samuel R. Madden. Resource-aware wireless sensor-actuator networks. *IEEE Data Engineering*, 28, 2005.

[DHJT+10] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. smap: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 197–210, New York, NY, USA, 2010. ACM.

[DHM08] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering Heap Overflow Exploits with JavaScript. In *Proceedings of USENIX Workshop on Offensive Technologies*, 2008.

[DRAPZ04] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin. pREST: A REST-based Protocol for Pervasive Systems. In *Proceedings of IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, October 2004.

[ECPC11] V.L. Erickson, M.A. Carreira-Perpinan, and A.E. Cerpa. Observe: Occupancy-based system for efficient reduction of hvac energy. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 258 –269, april 2011.

[EGgC+10] William Enck, Peter Gilbert, Byung gon Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of OSDI*, 2010.

[Ene07] Energy Information Administration (US Dept. of Energy). About U.S. Natural Gas Pipelines. http://www.eia.gov/pub/oil_gas/natural_gas/analysis_publications/ngpipeline/fullversion.pdf, June 2007.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[FJM+95] Sally Floyd, Van Jacobson, Steve McCanne, Ching-Gung Liu, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 342–356. ACM Press, 1995.

[Fla06] David Flanagan. *JavaScript: The Definitive Guide, Fifth Edition.* O'Reilly Media, Inc., Sebastopol, CA, 2006.

[FT10] Eric B. Flynn and Michael D. Todd. A bayesian approach to optimal sensor placement for structural health monitoring with application to active sensing. *Mechanical Systems and Signal Processing*, 24(4):891 – 903, 2010.

[GFJ+09] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 1–14, New York, NY, USA, 2009. ACM.

[GGJ+06] Omprakash Gnawali, Ben Greenstein, Ki-Young Jang, August Joki, Jeongyeup Paek, Marcos Vieira, Deborah Estrin, Ramesh Govindan, and Eddie Kohler. The TENET architecture for tiered sensor networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (Sensys)*, 2006.

[GTK08] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416, Washington, DC, USA, 2008. IEEE Computer Society.

[HAAIKR11] M.S. Hossain, A.B.M. Alim Al Islam, M. Kulkarni, and V. Raghunathan. usetl: A set based programming abstraction for wireless sensor networks. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 354 –365, april 2011.

[HAM05] Jyh-How Huang, Saqib Amjad, and Shivakant Mishra. Cenwits: a sensor-based loosely coupled search and rescue system using witnesses. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 180–191, New York, NY, USA, 2005. ACM.

[HC04] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM Press.

[Hica] Ian Hickson. HTML5: A vocabulary and associated APIs for HTML and XHTML. http://dev.w3.org/html5/spec/.

[Hicb] Ian Hickson. Web Storage: Editor's Draft, August 20, 2010. http://dev.w3.org/html5/webstorage.

[HS10] Jon Howell and Stuart Schechter. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Proceedings of W2SP*, 2010.

[HSH+08] Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 225–238, New York, NY, USA, 2008. ACM.

[HSL+11] Timothy W. Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I. Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. The hitchhiker's guide to successful residential sensing deployments. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 232–245, New York, NY, USA, 2011. ACM.

[JM86] Albert T. Jones and Charles R. McLean. A proposed hierarchical control model for automated manufacturing systems. *Journal of Manufacturing Systems*, 5(1):15 – 25, 1986.

[KGMG07] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 200–210, New York, NY, USA, 2007. ACM.

[Khr] Khronos Group. OpenCL. http://www.khronos.org/opencl.

[KKP+06] Aman Kansal, William Kaiser, Gregory Pottie, Mani Srivastava, and Sukhat Gaurav. Virtual high-resolution for sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 43–56, New York, NY, USA, 2006. ACM.

[KLLB] Emmanouil Koukoumidis, Dimitrios Lymberopoulos, Jie Liu, and Doug Burger. Improving Mobile Search User Experience with SONGO. Microsoft Research Tech Report MSR-TR-2010-15, February 18, 2010.

[Lan] Norbert Landsteiner. How to Write a Pacman Game in JavaScript. http://www.masswerk.at/JavaPac/pacman-howto.html.

[LCL+12] Ted Tsung-te Lai, Wei-ju Chen, Kuei-han Li, Polly Huang, and Hao-hua Chu. Triopusnet: Automating wireless sensor network deployment and replacement in pipeline monitoring. In *IPSN '12: Proceedings of the 11th ACM/IEEE conference on Information processing in sensor networks*, 2012.

[LGC05] Philip Levis, David Gay, and David Culler. Active sensor networks. In *Second USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2005.

[Lju08] Lennart Ljung. Perspectives on system identification. In *Proceedings of the 17th IFAC World Congress*, July 2008.

[LL08] Kaisen Lin and Philip Levis. Data discovery and dissemination with dip. In *Proceedings of the 2008 International Conference on Information Processing in Sensor Networks (IPSN 2008)*, pages 433–444, Washington, DC, USA, 2008. IEEE Computer Society.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Simulating large wireless sensor networks of tinyos motes. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[LMET08] Chieh-Jan Mike Liang, Razvan Musaloiu-Elefteri, and Andreas Terzis. Typhoon: A reliable data dissemination protocol for wireless sensor networks. In *Proceedings of 5th European Conference on Wireless Sensor Networks (EWSN)*, pages 268–285, 2008.

[LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[LPL+09] Hong Lu, Wei Pan, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys '09*, pages 165–178, New York, NY, USA, 2009. ACM.

[LS11] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems. A Cyber-Physical Systems Approach*. http://LeeSeshia.org, 2011.

[Lub02] Michael Luby. Lt codes. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 271–282, 2002.

[Map] MapQuest. Mapquest maps. http://www.mapquest.com/.

[Mas98] L. Masinter. The "data" URL Scheme. RFC 2397 (Draft Standard), August 1998.

[Mer79] R. Merkle. Secrecy, authentication, and public key systems. Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, 1979.

[MHE10] J. Mickens, J. Howell, and J. Elson. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of NSDI*, San Jose, CA, April 2010.

[MLE08] L. Montestruque, M. Lemmon, and LLC EmNet. CSOnet: a metropolitan scale wireless sensor-actuator network. In *International Workshop on Mobile Device and Urban Sensing (MODUS)*, 2008.

[Mon] Monsoon Soluitions Inc. Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[MP11] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43:19:1–19:51, April 2011.

[mre] How much electricity does my stuff use? http://michaelbluejay.com/electricity/howmuch.html.

[MRVB00] Randall J. Mumaw, Emilie M. Roth, Kim J. Vicente, and Catherine M. Burns. There is more to monitoring a nuclear power plant than meets the eye. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 42(1):36–55, Spring 2000.

[MT79] Robert Morris and Ken Thompson. Password security: a case history. *Commun. ACM*, 22(11):594–597, 1979.

[MVO+05] Dalius Misiunas, John Vitkovsky, Gustaf Olsson, Angus Simpson, and Martin Lambert. Pipeline break detection using pressure transient monitoring. *Water Resources Planning and Management, Journal of*, 131(4):316–325, July/August 2005.

[NASZ07] Vinayak Naik, Anish Arora, Prasun Sinha, and Hongwei Zhang. Sprinkler: A reliable and energy efficient data dissemination service for extreme scale wireless networks of embedded devices. *IEEE Transactions on Mobile Computing*, 6(7):777–789, 2007.

[Nik] Nike. Nike+. http://nikerunning.nike.com/nikeos/p/nikeplus/en_US/.

[Nik01] Michael Nikolaou. *Model Predictive Controllers: A Critical Synthesis of Theory and Industrial Needs*. Advances in Chemical Engineering Series, Academic Press, 2001.

[NTCS99] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 151–162. ACM Press, 1999.

[oct] GNU Octave. http://www.gnu.org/software/octave/.

[OHM] I. Oksanen and D. Hazael-Massieux. HTML Media Capture. http://www.w3.org/TR/html-media-capture/.

[OPFF09] T.G. Overly, Gyuhae Park, K.M. Farinholt, and C.R. Farrar. Piezoelectric active-sensor diagnostics and validation using instantaneous baseline data. *Sensors Journal, IEEE*, 9(11):1414 –1421, nov. 2009.

[osh] OSHA Ergonomic Solutions: Computer Workstations. http://www.osha.gov/SLTC/etools/computerworkstations/wkstation_enviro.html.

[PFH03] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of USENIX Security*, 2003.

[Pho] Phonegap. PhoneGap. http://www.phonegap.com/.

[PKGZ08] Bodhi Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny web services for sensor device interoperability. In *Proceedings of IPSN*, pages 567–568, Washington, DC, USA, 2008. IEEE Computer Society.

[Pop] Andrei Popescu. Geolocation API specification. http://dev.w3.org/geo/api/spec-source.html.

[PSC05] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 48, Piscataway, NJ, USA, 2005. IEEE Press.

[RDML05] Nishkam Ravi, Nikhil Dandekar, Preetham Mysore, and Michael L. Littman. Activity recognition from accelerometer data. In *Proceedings of the 17th conference on Innovative applications of artificial intelligence - Volume 3*, IAAI'05, pages 1541–1546. AAAI Press, 2005.

[RG11] D. Richardson and S. Gribble. Maverick: Providing Web Applications with Safe and Flexible Access to Local Devices. In *Proceedings of USENIX WebApps*, Boston, MA, June 2011.

[RLZ09] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of USENIX Security*, 2009.

[Sal05] Timothy Salsbury. A survey of control technologies in the building automation industry. In *International Federation of Automatic Control World Congress (IFAC)*, 2005.

[Sar83] G. Saridis. Intelligent robotic control. *Automatic Control, IEEE Transactions on*, 28(5):547 – 557, may 1983.

[Sec08] Secunia. Secunia Advisory SA29787: Mozilla Firefox Javascript Garbage Collector Vulnerability. http://secunia.com/advisories/29787, April 21 2008.

[SG08] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4:8:1–8:29, April 2008.

[SHSM06] Fred Stann, John Heidemann, Rajesh Shroff, and Muhammad Zaki Murtaza. Rbp: robust broadcast propagation in wireless networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 85–98, New York, NY, USA, 2006. ACM.

[SLV11] János Sallai, Ákos Lédeczi, and Péter Völgyesi. Acoustic shooter localization with a minimal number of single-channel wireless sensor nodes. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 96–107, New York, NY, USA, 2011. ACM.

[Sol03] E. Soljanin. Hybrid arq in wireless networks. DIMACS Workshop on Network Information Theory, 2003.

[SSKM07] Makoto Suzuki, Shunsuke Saruwatari, Narito Kurata, and Hiroyuki Morikawa. A high-density earthquake monitoring system using wireless sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 373–374, New York, NY, USA, 2007. ACM.

[t2] TinyOS. http://www.tinyos.net.

[TC05] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 121 – 132, jan.-2 feb. 2005.

[TGW11] Muhammad Umer Tariq, Santiago Grijalva, and Marilyn Wolf. Towards a distributed, service-oriented control infrastructure for smart grid. In *Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, ICCPS '11, pages 35–44, Washington, DC, USA, 2011. IEEE Computer Society.

[TMD+07] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the darpa grand challenge. In Martin Buehler, Karl Iagnemma, and Sanjiv

Singh, editors, *The 2005 DARPA Grand Challenge*, volume 36 of *Springer Tracts in Advanced Robotics*, pages 1–43. Springer Berlin / Heidelberg, 2007.

[US 91] US Environmental Protection Agency. Indoor Air Facts No. 4 (revised) Sick Building Syndrome. http://www.epa.gov/iaq/pubs/sbs.html, February 1991.

[US 06] US Army. *Supervisory Control and Data Acquisition (SCADA) Systems for Command, Control, Communications, Computer, Intelligence, Surveiillance, and Reconnaissance (C4ISR) Facilities*, January 2006.

[VHXS10] Pascal A. Vicaire, Enamul Hoque, Zhiheng Xie, and John A. Stankovic. Bundle: a group based programming abstraction for cyber physical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 32–41, New York, NY, USA, 2010. ACM.

[Vix97] P. Vixie. Dynamic Updates in the Domain Name System (DNS Update). RFC 2136 (Draft Standard), April 1997.

[Wan04] Limin Wang. MNP: multihop network reprogramming service for sensor networks. In *Proceedings of the Second ACM Conference On Embedded Networked Sensor Systems (SenSys)*, pages 285–286, New York, NY, USA, 2004. ACM Press.

[WGM+09] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The Multi-principal OS Construction of the Gazelle Web Browser. In *Proceedings of USENIX Security*, pages 417–432, 2009.

[Wor08] World Wide Web Consortium (W3C). Access Control for Cross-Site Requests. W3C Working Draft, September 12 2008.

[WSA01] H. A. Winston, F. Sun, and B. S. Annigeri. Structural health monitoring with piezoelectric active sensors. *Journal of Engineering for Gas Turbines and Power*, 123(2):353–358, 2001.

[WTT+06] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM.

[YD09] D. Yazar and A. Dunkels. Efficient Application Integration in IP-Based Sensor Networks. In *Proceedings of BuildSys*, November 2009.

[YMC07] Aydan R. Yumerefendi, Benjamin Mickle, and On P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of NSDI*, 2007.

[ZK04] Marco Zuniga and Bhaskar Krishnamachari. Analyzing the transitional region in low power wireless links. In *First IEEE International Conference on Sensor and Ad hoc Communica tions and Networks (SECON)*, 2004.