OWL PLATFORM: DESIGN AND EVALUATION OF A PRACTICAL SENSOR NETWORK SYSTEM

By

ROBERT S. MOORE II

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

written under the direction of

Richard P. Martin

and approved by

New Brunswick, New Jersey

October, 2014

ABSTRACT OF THE DISSERTATION

Owl Platform: Design and Evaluation of a Practical Sensor Network System

By ROBERT S. MOORE II

Dissertation Director:

Richard P. Martin

As wireless sensor networks (WSNs) become larger and more widespread, new systems are required to manage them more effectively. Owl Platform is a multi-layered architecture which attempts to provide a simple, efficient infrastructure for managing data generated by WSNs. A proof of concept implementation is provided as a set of network protocols, multi-language libraries, prototype services, and applications supporting research and commercial activities. The motivation, design, and evaluation of this system are explored and compared to past and current alternatives. Such a system is capable of scaling to tens of thousands of devices, supported by off-the-shelf hardware.

Acknowledgments

I wish to thank the many different people who have helped me throughout the years. First, I would like to acknowledge the support and guidance that my advisor, Prof. Richard P. Martin, has given me over the past 9 years. At the start of my graduate studies, he brought me into a project that inspired some of the most rewarding years of my life.

I am grateful to my family, especially my parents. They tought me to question everything, although I may have taken them too literally at times. They instilled in me a strong work ethic and an enduring appreciation for education and knowledge that has served me so well in my life. Without these I would never have been able to come this far.

My research associates at WINLAB have been wonderful guides to me over the past several years. Ben Firner is an incredibly talented individual who has taught me as much about being a successful graduate student as he has about programming and radios. Rich Howard showed me how much fun it can be to explore science and engineering, and how important it is to keep a careful record of what I do. Prof. Yanyong Zhang has helped me hone my writing and is almost directly responsible for most of my publications. Eitan Fenson has been a mentor to me both literally in the I-Corps program and more generally as I begin my journey out of academia and into business.

The support staff at LCSR have always treated me with respect and kindness, listening to my ideas and even championing some of them over the years. I have never been the easiest person for you to work with, and I greatly appreciate your patience and understanding. In particular, Doug Motto provided many great insights that guided the development of the Owl Platform libraries, making it more accessible and easy to use. Hanz, Lars, Rick, Rob Toth, Charles, and Don, thank you for keeping things running while I wrought havoc in your halls. Thank you Augustine and Jessy for your contributions to both GRAIL and Owl Platform, always willing to go the extra mile.

Finally, but never least, I want to thank my wife Eunji for standing by my side, especially over the past year. You have always motivated me to be my best, never allowing me to give-up when I got tired. Without you I may have become a perpetual student, and I want to thank you for everything you've done.

Dedication

For Eunji, my loving wife.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction and Problem Definition

A wireless sensor network (WSN) is a collection of small autonomous computing devices capable of communication over a wireless network and deployed with the goal of capturing details about the environment in which they are placed. In recent years as the cost of computer components decreases and wireless capabilities improve, the development of small, low-cost wireless sensors has increased tremendously [20]. In just the last 5 years, the rise of the smart phone as a common personal accessory has transformed human beings into walking sensor platforms. When these devices are connected via wireless computer networks, it becomes possible to obtain information about physical surroundings on scales never before possible.

Traditional sensor networks include seismometers, weather stations, and even astronomers comparing readings of the night sky. The development of global communication networks has enabled many of these devices to be connected in large-scale collaborative data gathering tasks. Combined with the recent development of low-cost commodity wireless communication networks like Wi-Fi and Z-Wave, it is now possible for sensors to be deployed in great numbers over large areas to collect unprecedented quanitites of information.

Coincident with the development of wireless sensor hardware is the attendant development of software to control, manage, and coordinate these devices [23, 3, 2]. These are necessary goals, since WSNs have neither the large computing and energy resources of traditional computers nor the relatively low numbers of such computers. New techniques are being developed to support self-localization, self-organization, network routing, in-network processing, and even energy harvesting and sharing among devices [17]. Just as important as managing the networks themselves is how to manage they data they create once it exits the network.

Many of these systems, however, address only a specific set of goals or applications. They focus on an initial goal that needs to be met, and the resulting design is one that may be incapable of supporting applications with very different data types, functional requirements, and scales. For example, a system designed to gather weather data around the globe will have different priorities, processes, and data structures than one designed to control the appliances inside a person's home. In order to build generalized, effective systems for managing and manipulating WSN data, it is necessary to assume that today's hardware and motivations will not be tomorrow's. *To build such systems, a top-down approach focused on lowering barriers to entry and ease of development, results in a more flexible design than those centered around specific hardware devices or network topologies.*

1.1 Contribution

Owl Platform is a multi-layered WSN data management architecture focused on simple interfaces, flexible data structures, and data re-use. From its beginnings as a research tool used to explore wireless localization algorithms, it has expanded to support a wide range of goals and applications. The goal of this thesis is to describe Owl Platform as an architecture for manipulating WSN data and application development, evaluate the performance of a prototype implementation, and describe the experiences of its developers and users.

Chapter 2 describes the origins of Owl Platform and its growth over time to meet the needs of its users. Motivated by an indoor localization system that had outgrown its original design, its design is guided by three basic assumptions: Wireless Sensor Networks are wireless, they produce data, and some data is only useful for a limited period of time.

The system is broken up into five layers with WSNs on the bottom and Applications on the top. Interacting with the WSNs are one or more Aggregators, components that consolidate and filter the WSN data stream. Aggregators provide components in the Analysis Layer, called Solvers, with an interface that allows the selection of specific physical hardware types, specific transmitters, and data flow rates. As Solvers translate and transform the WSN data into a virtual representation of the environment, that data is stored in a World Model. The World Model provides access to this virtual representation via set of Requests which can be used by Applications to present the data to users, or by Solvers to provide more complex analyses of the data by combining it with external services.

Chapter 3 describes the systems and products that have motivated and guided the development of Owl Platform. Infrared and ultrasonic badges worn by researchers set the stage for systems that provide automated monitoring and context-based services to the people that use them. More recent work on “smart” homes and offices describe the types of applications that are possible by exploiting environments rich with sensors and networked devices. Finally, the scale of global sensor networks utilizing the framework of the World Wide Web to provide interoperability demonstrates the effectiveness of tools which are not overly specialized.

Chapter 4 focuses on the need to avoid making too many assumptions when dealing with WSN data. Some sensors can be queried and reprogrammed while others blindly report their readings until their power sources are exhausted. The design choices and thinking behind them are explored and described. While Owl Platform is intended to be a sort of “jack of all trades” among WSN data management systems, it is a specialist of none. The advantages and limitations of this approach are presented and justified.

Chapters 5 and 6 explore the performance of the two primary components of the Owl Platform’s architecture: the Aggregator and the World Model. Chapter 5 studies the effect on Aggregator performance as a system scales in various dimensions. It examines the effect of an increasing volume of sensor data arriving from WSNs, how the number of Solvers requesting that data impacts processing time, and how the complexity of Solver requests for WSN data has the potential to create bottlenecks in the system. It demonstrates how the system is capable of handling workloads comparable to hundreds of thousands of sensors, and also discovers some shortcomings in the current prototypes.

Chapter 6 looks at two different choices for the World Model storage engine, and how that choice impacts performance. It also takes the first steps in profiling the performance of the World Model as the amount of stored data increases over time. Each of the World Model’s interfaces are evaluated and benchmarking measurements are made to use as guideposts for future improvements. A discussion of the different prototype implementations’ limitations and possible improvements concludes the evaluation.

Chapter 7 describes the different ways that Owl Platform has been used since its creation. The various applications that have been developed on top of Owl Platform are described, and the experiences of their authors are summarized. Undergraduate students have built a range of applications including a stolen chair detector, a context-aware power switch, and reminders to keep windows closed during cold weather. Researchers have built radio tomography systems, augmented reality applications, and conducted high-density wireless experiments using Owl Platform as a foundation. A newly-developed commercial product highlights the robustness and reliability of a system already proven in the laboratory.

Finally, Chapter 8 wraps things up and looks at possible future directions for Owl Platform. A combination of lessons learned and new ideas, it plots a course for an improved system that provides greater performance, richer features, and more flexible interfaces.

Chapter 2
Architecture

2.1 Background and Predecessor

The architecture of Owl Platform is highly motivated by its predecessor the General Real-time Adaptable Indoor Localization (GRAIL) system. In 2007, GRAIL was originally intended to demonstrate off-line analysis of indoor localization algorithms for wireless networks (version 1) [27]. Later it would be rewritten for online analysis and real-time evaluation of algorithms (version 2) [6]. To that end, it was focused primarily on the collection and processing of signal measurements in order to produce location data.

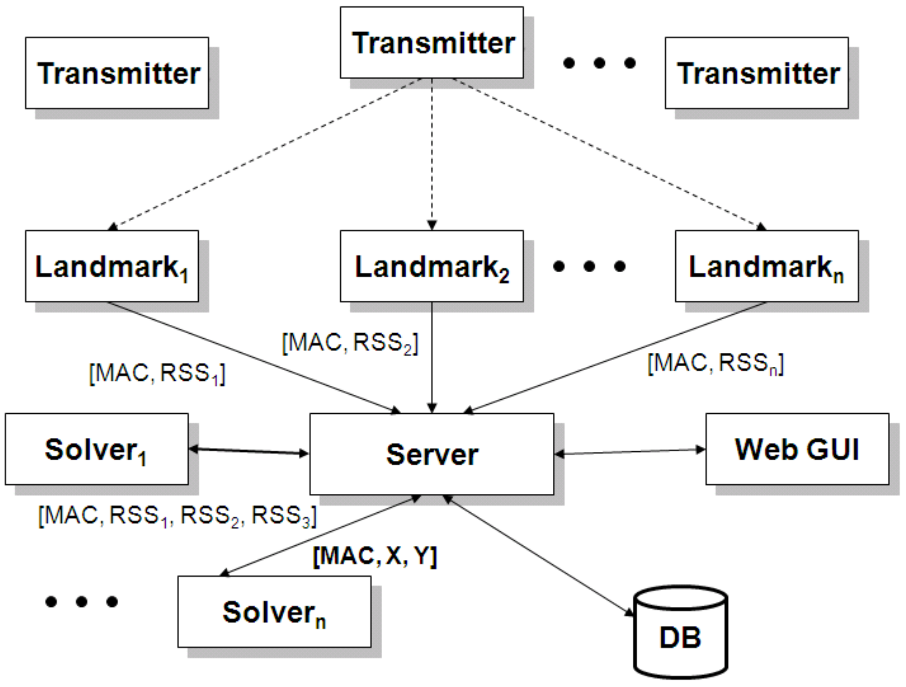


Figure 2.1: Architecture diagram of GRAIL version 2.

The architecture of GRAIL version 2 can be seen in Figure 2.1. It centers on a central component, the **Server** which collects, organizes, and stores the information needed for wireless localization. To provide data streams for the Server, a number of wireless receivers called **Landmarks** are deployed throughout an area of interest such that at any point two or more Landmarks can receive the broadcast of any transmitting wireless device, or **Transmitter**. Each time a Transmitter broadcasts a wireless frame, Landmarks receive it and extract the relevant information, including Received Signal Strength Indicators (**RSSI**), Angle of Arrival (**AoA**), and Time of Arrival (**ToA**). This data is then aggregated, processed, and prepared for submission to a **Solver**, which uses the data to produce an estimation of the Transmitter location. All of the data, including Landmark configuration and layout details, raw transmitter data, and estimated results, were stored in a database by the Server for retrieval by the graphical user interface (GUI) or later analysis by researchers.

This design was very effective and reasonably efficient for the purposes of wireless localization research, including Solver algorithm design and evaluation. New Solvers could be written, but only if they used the same interface to the Server, and only if they returned the same type of data. Worse yet, the interface between the Server and Solver components relied on a shared file system, as well as careful management of specially-named files used for inputs and outputs. This created a significant bottleneck in the expansion of the system when new or multiple types of Solvers were written. The Server did not provide access to the raw data stream to Solvers, and any time a new form of analysis was needed for a Solver (e.g., sliding-window variance of RSSI values), the Server component had to be updated to include all of the requisite commands, formats, and algorithms. Over months and years of updates and additions, the Server supported so many new features, that it was almost impossible to maintain. The only part that remained constant was the protocol between the sensor network and the Server.

The data format used to transfer the raw wireless data from the Landmarks to the Server is called a **Sample**. It originally represented IEEE 802.11 (“Wi-Fi”) frames, and included fields for RSSI, Quality of Service (QoS) priority, Media Access Control (MAC) address, sequence number, raw header, and even a histogram of RSSI values. In practice, many of the fields were often left unpopulated either because the values were not necessary for the localization being performed, or because they were not available to be accessed through the operating system device driver available. As a concept, the Sample was very effective, but its implementation was overly specific and unwieldy for general use. Although the system still performed the functions needed, it becmae clear that it would need to be replaced.

In 2008, support for the recently-developed Pipsqueak wireless sensors was added[11]. Because of their small size, low cost, and ease of deployment, it became possible to explore new types of applications including passive (device-free) localization and reporting of sensor data (temperature). One of the first uses of the new devices was to emulate the workflow of a hospital emergency department, localizing each of the staff members and patients. The system behaved well except that it could not produce the data in “real time”. The localization algorithm required approximately 10-15 seconds to localize all of the 8-10 participants based on Samples and positions from another 20 “fiduciary” transmitters placed throughout the experiment area. In order to produce a useful interface for real-time applications, the Server was rewritten to spawn multiple copies of a Solver, sending only a subset of desired devices for localization to each copy. This reduced the latency of each location estimation, but resulted in a much more complex Server code base and runtime environment.

A new GUI was also developed to assist in system development, evaluation, management, and deployment. The new GUI provided a real-time holistic view of the RSSI data arriving in the Landmark Samples. It produced charts, graphs, and maps, and allowed the users to easily view localization results, signal strength maps, and measured signal propagation characteristics for the system. Once again a new set of functions needed to be implemented in the Server. The new interface also required direct access to the raw Sample values, and a new streaming protocol was designed and implemented on top of the the old “interactive” protocol used to configure and run the system.

These new applications were far removed from the system’s original purpose of RSSI-based device localization, and maintenance and development were becoming increasingly difficult. The system architecture, with the Server and database components central to its operation, was unable to deal with emerging requirements. Expanded deployments to multiple floors and separate buildings further strained the system to the point of breaking. It became clear that the only way to proceed further was a complete rewrite of the system from the ground up. The new system would provide support for device localization and also provide a sustainable framework for extending its functionality.

2.2 GRAIL version 3: Owl Platform

What had begun as a testing framework to evaluate indoor wireless localization algorithms had become a real-time system supporting modular localization Solvers, multiple user interfaces, and even some sensor data. The new design of the system, Owl Platform, would attempt to balance the needs of many different applications - many of which had not been written or even thought of. The first set of applications written for the new architecture would be a mix of old and new - the same localization and interface applications used in version 2, and also new applications for collecting and analyzing wireless sensor data. During the same period that the newly-redesigned system was being developed and tested, a laboratory intern was available and would provide feedback from the perspective of a novice programmer.

2.3 General Design Principles

Driving the design of Owl Platform were three goals: focus on Wireless Sensor Networks (WSNs) and the data they produce, support much larger-scale deployments, and keep the individual components as simple as possible. The system would be structured such that each layer would depend on the one below and provide services to the one above. As in many layered architectures, this provided the opportunity to isolate components, provide well-defined interfaces, and make implementations interchangeable.

When considering the scale of such a system, it was decided that it should support something the size of a large university campus. This was both a practical consideration (envisioning an eventual campus-wide deployment), and also because it seemed likely that anything at larger scales would require significantly different design choices. We estimated that an average building might have approximately 1000 sensors, and the entire system might support 200 such buildings. Any application requiring a larger system than this would need to interact with multiple "instances" of the system, and coordination between the systems would take place at this level.

The range of planned applications started with those that had been previously developed: wireless device localization, device-free motion localization, and simple event reporting. Some new ideas would be explored by the student intern: RSSI-based refrigerator opening detection, moved-chair detection, Twitter integration, and a "tea time" context detector. Given the scope of the proposed system, it was clear each layer/component needed to remain as simple as possible to avoid the same problems encountered in version 2. The layers were designed such that each provided a very limited number of services, and the data structures were kept as flexible and generic as possible. Two primary components were also designed: an **Aggregator** to combine all Wireless Sensor Network (WSN) data streams into a single point of contact, and a **Distributor** storage engine that would provide a virtual representation of the world based on information produced by Solvers.

2.4 Layering, Responsibilities, Interfaces

It is not possible to know exactly which applications will be desired from a system like this, nor is it likely those deemed most important at the start will remain important over time. Modularity, well-defined interfaces, and data-agnostic processes all become critically important to ensure flexibility. With the focus of the system is on Wireless Sensor Networks, a few basic assumptions can be made to guide the design.

The first assumption is that Wireless Sensor Networks are wireless. This is a seemingly obvious assumption, but one that will factor greatly into the design of the lowest layers of the system -

where the physical properties of the underlying networks are most closely modeled. One defining factor of nearly all radio equipment today is that the receiver is capable of producing a value representative of the power of the received signal. Most equipment provides either an RSSI value or an equivalent measurement. This value is by-and-large a scalar value, and justifies its inclusion in the representation of a Sample.

The second assumption is that WSNs produce "data". That they produce data is equally obvious and even the name "Wireless *Sensor* Network" demands that they *sense* something. Some sensors may be configurable, queryable, or perform basic processing. Cell phones are an excellent example of devices that may be part of a WSN that can be configured, respond to queries, and perform complex analysis on sensed data before sending it out to the rest of the network. On the other hand, devices like the Pipsqueak sensor blindly sense their environment and report the results periodically until their power source is exhausted. So the system should likewise represent the sensed "data" in some form, though that form cannot be predicted ahead of time, and a generic format should be adopted.

The third and final assumption is that some data will be useful only within a short period of time after it is produced, whereas others will need to be stored for long periods of time. A simple example can be found in a typical building security system that has sensors on various doors and windows to detect when they are open or closed. Imagine such a system reports the state of a door to be *Open* or *Closed* every 100 milliseconds. Even if this data is stored as only a single bit, the operator of a system does not actually care to interrogate each individual point of the door's status. The user only wants to know when the door's state is *Open* and when it is *Closed.* Instead of storing every single piece of data, the system could combine a series of scalar data points into a single vector value indicating the starting and ending times of a specific state of the door. This will result in a significantly smaller storage requirement for the system, and no loss of information that is important to the user. From this example, we can see that the system should be able to distinguish between raw *data* and semantically-useful *information.* Furthermore, it should distinguish between **On-Demand** and **Permanent** information.

With these ideas in mind, the initial design for Owl Platform was a layered system with three major components: an **Aggregator** to consolidate WSN traffic to a single point, a **Distributor** to hold data that was dynamically generated by system components, and a **World Server** that contained static or rarely-changing state about the physical environment. Providing data to the Aggregator would be the **Sensing Layer**, which formatted WSN data into a standardized stream of data values called **Samples**. Connecting to the Aggregator to receive the WSN data would be analysis components, called **Solvers**, residing in an **Analysis Layer**. These Solvers would process the raw WSN data and produce data elements called **Solutions**.

2.5 Initial Design

The initial design of Owl Platform, shown in Figure 2.2, divided the system into 5 layers: Sensing, Aggregation, Analysis, Distribution, and Presentation. Somewhat independent from this architecture was the World Server that would interact with both Solvers and applications in the Presentation layer. It was intended that the Distribution Layer would be defined by a set of distributed Distributor components - these would communicate and share data between each other allowing Solvers and Clients to access any available node.

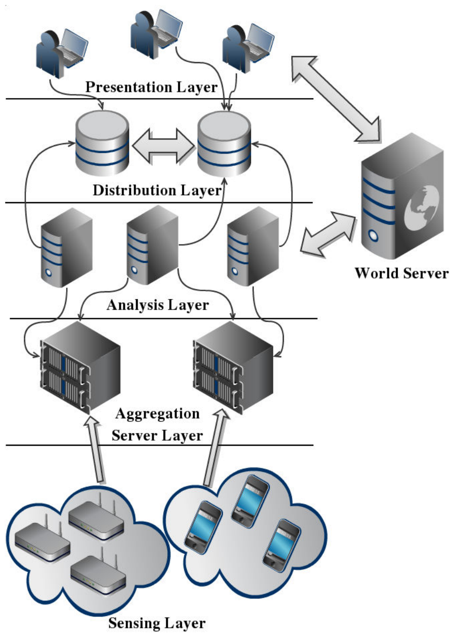


Figure 2.2: Initial architecture diagram of Owl Platform (December 2010).

2.5.1 Sensing Layer

At the lowest level, WSNs convert data into a stream of Samples. The Samples would contain the physical layer (PHY) identifier, transmitter and receiver identifiers, RSSI value observed by the receiver, and any recorded data sent by the sensor. This very simple abstraction of WSN data meant that nearly any type of sensor or sensor network would be able to interact with Owl Platform. The only limit placed on the Sample was that the length of the sensed data was limited to approximately 64 kB. This was not a technical limitation, but a practical one. Any data larger than 64 kB was unlikely to have been sent in a single wireless frame, and was more appropriate to be incorporated at higher layers of the system.

2.5.2 Aggregation Layer

The Aggregator combines streams of Samples from WSNs, or even other Aggregators, and provides an interface for Solvers to receive this data in real time. It does not perform any storage of the Samples, other than what is necessary to buffer incoming data before being sent to the Solvers. Excluding storage of Samples greatly simplifies both the design of the Aggregator, and the protocol between the Aggregator and the Solvers. If storage of low-level Sample data is desired, a Solver can be written that performs the same function - storing the data either in the Distribution Layer or in a data base or file. For Solvers requesting WSN data, the Aggregator provides a simple subscription-publication protocol that defines two simple filtering mechanisms: selecting on physical layer and ID, and enforcing a minimum time between forwarded Samples for each transmitter-receiver pair. Selection of individual PHY and transmitter ID values allows a Solver to receive Samples only from those transmitters it actually needs. Enforcing a minimum time between forwarded samples can be useful when a device or WSN produces data at a rate greater than needed for a particular Solver. Both of these techniques further serve to reduce network traffic between the Aggregator and Solvers, with minimal impact on throughput and delay.

2.5.3 Analysis Layer

In the Analysis Layer, Solvers perform all of the computation necessary to generate the information needed by applications in the Presentation layer. The simplest function of a Solver is to transform WSN sensor data into a more usable form. For example, temperature data may come from a variety of sensors, in a variety of formats. Instead of an Application requiring knowledge of all the varied types of sensors and formats, which may change significantly over time, one or more Solvers can convert those low-level data formats into a single standardized data format usable by the Application. Solvers may also build on one another to produce more complex forms of data. The output of the temperature Solvers may be used by another Solver to produce histograms over time, physical temperature maps, or perform correlations with weather data from external data sources. This design promotes both data reuse and also modularity of Solvers. And because the interaction between them is entirely through the Distribution Layer, other components need not change as a result.

The Solvers can produce two general types of data: Permanent data that is stored indefinitely, and On-Demand data that is only produced when a request for it is made, and is never stored by the system. On-Demand data is only sent to the Distributor once it has been requested from another connection, eliminating unnecessary computation and reducing workload on the Distributor. Any

Permanent data sent to the Distributor is consequently stored in a persistent storage implementation for later retrieval via requests.

2.5.4 Distribution Layer

The Distribution Layer is a network of nodes that stores and exchanges information among one another in response to requests from the Presentation Layer. The requests can be made from either traditional applications, which present the data to users or external systems, or from Solvers that will use to the data to generate additional Solutions.

The original vision for Distributors was to have each communicate with another, in a Peer-to-Peer (P2P) topology and exchange data with one another. It was envisioned that this would enable a more scalable system for large enterprise installations, would allow data to be replicated or cached, and support more complex future interaction between Distributors and Solvers. One example is that of a large university with multiple physical campuses. Each department or unit could host its own Distributor (or have it hosted by IT staff), keeping some data "private" and allow other data to be "public". Private data would only be accessible through a direct connection to the Distributor via an authenticated session, while public data would be accessible to any connection. With this approach, a hierarchical model could be developed of a university-wide system that enabled some data to be made available publicly (e.g., gross attendance rates for classes), while keeping other data private for policy reasons (e.g., individual student attendance).

Entries in a Distributor would be uniquely addressable by a Uniform Resource Identifier (URI). Each URI would have two major components: a Domain and an Identifier. The Domain would be linked to access restrictions (public/private, authorized roles, lifetime), and the identifier would be unique to some object, action, or context (e.g., a door or meeting). If the same Identifier existed in two Domains, then it would represent the same object/event. This would enable different levels of detail or content to be associated with the Identifiers.

The World Server exists independently of the main software stack with the intention that it is accessed from both the Analysis and Presentation layers. It utilizes the same URI format to store long-term, perhaps even unchanging, information about the entities in the Distributor. For instance, it may store the physical properties of a meeting room, the layout of an office building, or the model number of a coffee maker. Unlike the Distributor, which stores programmatically-generated information, it was expected that much of this information would be entered manually by users. Utilizing the same URI structure (Domain/Identifier) would enable a loose binding between the two components, but still enabling each to focus on their specialized purposes.

URI	**Solution**	**Creation**	**Expiration**	**Data Type**	**Origin**	**Data**
staff.chairperson	In Office	07/01 07:55	*null*	boolean	localize	True
staff.chairperson	In Office	06/30 18:01	07/01 07:55	boolean	localize	False
staff.chairperson	In Office	06/30 07:53	06/30 18:01	boolean	localize	True

Table 2.1: Example listing Distributor entries for a single URI value. Creation/Expiration dates exclude years for brevity.

A final vision of the World Server was to store bindings between objects in the Distributor, or between the World Server and Distributor. For instance, if the URI Identifier or Domain of an object was not known, it would be possible to search the World Server values for objects that matched the desired object. Once found, the URI could be retrieved and used to request data from the Distributor. This would be similar to how DNS binds human-readable domain names to IP addresses or to other host names.

URI	**Field Description**	**Field Data**
edu.rutgers.cs.staff.chairperson	Office	H305
edu.rutgers.cs.staff.chairperson	Name	Jane Q. Johnson
edu.rutgers.facilities.buildings.3752	Code	HLL
edu.rutgers.facilities.buildings.3752	Campus	Busch
edu.rutgers.facilities.buildings.3752	Erected	1971

Table 2.2: Example listing of World Server entries.

The distinction between the Distributor and the World Server, as well as the networked design of the Distributors, would eventually be removed. As development of the first Solvers and Applications progressed, it became increasingly apparent that having two separate components was the wrong choice. Logically, the two were closely connected: they both utilized the URI object format to track the same entities, the both used very similar data formats for storing the contents of the entities, and the software that needed to interact with both was much more complex than equivalent software that could connect to only a single component. The result was a merger of the two that created the World Model.

2.5.5 Presentation Layer

The Presentation Layer is the highest layer defined in the system. It is where end-user applications and external systems would interact with the Owl Platform architecture. The focus was on search and retrieval of data from the Distributor and World Server. Search was implemented using Regular Expression (RegEx) matching on URI strings and also on Attribute names, in both the World Server and Distributor. This permitted efficient look-up of values in both components, as long as they were named using a semi-hierarchical pattern. For example, the URI Identifier "edu.rutgers.cs.staff.chairperson" might identify information about the Computer Science department chairperson, while "com.google.scholar.site.access.counter" might contain information about the number of times Google Scholar was used. This naming convention is not enforced by either component, and it was left to the administrators and users of the system to design and adhere to any naming conventions. By not enforcing a naming convention, the system did not limit itself to a strictly higerarchical interpretation of objects, and developers could use application-specific naming systems where desired.

Interactions with the World Server were primarily single-transaction lookups of data. An application would connect to the World Server, perform a search on URI Identifiers, make a request for the matching data, and disconnect. Periodically, an application might reconnect and check for new entries or the rare data update. Because latency would be a factor with a polling-based protocol, it was decided that the World Server would support "standing queries" where any new or updated value that matched an initial query would be pushed to an open connection. This allowed applications or Solvers that needed the newest or latest data would get it with minimal latency.

Once the data values were retrieved from the World Server, the application would connect to the Distributor to request the desired data based on the URI values found in the World Server. Aside from URI search requests, which also utilized Regular Expression matching, the Distributor supported 3 primary request types: Snapshot Request, (Time) Range Request, and Standing Query. The Snapshot Request specified a unique time instance and would retrieve with the "current" state of the requested URI values at that point in time. Time Range Requests specified starting and ending timestamp values, and starting with a Snapshot at the starting timestamp, all updated values were sent in time-order until the ending timestamp. The Standing request began with a Snapshot of the values at a specified time included in the Request, and continued with any updates that arrived afterward.

With this design in mind, interfaces were specified, protocols were written, and the details of each component were fleshed-out. As work progressed on the prototype versions of each component, however, it became clear that the Distributor and World Server would need to be combined into a single unit. For both components, the network protocols and data structures used were very similar, and sometimes identical. Software development resulted in a lot of copied code (very little could be shared due to minor inconsistencies between the two), and was very slow as a result. In the end, these two would be merged into a single component called the **World Model**.

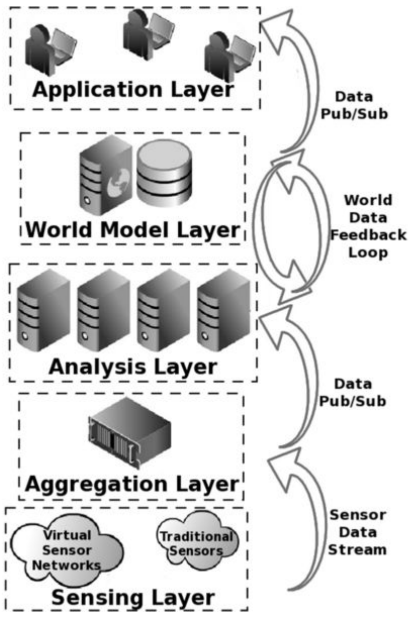


Figure 2.3: Final architecture diagram of Owl Platform (October 2011).

2.6 Final Design: World Model

The new World Model would contain the core components of both the Distributor and the World Server. Part of this merger, however, meant significant changes. Unlike the Distributor, World Models would not explicitly form a network with one another. The Presentation Layer was renamed to the Application Layer to more accurately express its purpose. Information could still be exchanged, but would come in the form of regular protocol connections - new data inserted via the Analysis Layer interface and retrieved via the Applicaiton Layer interface. The entity representations of the World Server and Distributor would be merged and simplified [12].

ID	**Attr. Name**	**Creation**	**Expiration**	**Origin**	**Data**
staff.chairperson	In Office	07/01 07:55	*null*	localize	True
staff.chairperson	In Office	06/30 18:01	07/01 07:55	localize	False
staff.chairperson	In Office	06/30 07:53	06/30 18:01	localize	True

Table 2.3: Example listing World Model entries for a single URI value. Creation/Expiration dates exclude years for brevity.

The World Model was now responsible for the management and dissemination of all data sent to it by connections from the Analysis Layer. Most of this data would be generated by Solvers, but some would be manually entered by users or sourced from external services. It would model the application environment as a collection of entities with characteristics that would change over time. Each entity was uniquely identified by an Identifier, which would no longer be required to conform to the URI standards set by IETF. It could be anything from a name to a hash value to a hierarchical listing. Third-parties would be required to adhere to whatever naming convention they wished. This flexibility simplified the design of the World Model, but put additional burden on developers to ensure that they communicated data types and Identifier values in some way.

Each entity was represented by a collection of values called **Attributes** which provided time-sequenced data about its state. Attributes are composed of a 6-tuple: **{Name, Origin, Creation, Expiration, OnDemand, Value}**. **Name** is the unique name, or identifier, of the Attribute. Examples of names might be "location.xoffset", "GPS Lat.", or "img/src". The **Origin** is the identifier of which process or user produced that specific piece of data for the Attribute. This might be something as simple as a version number for software, or might even be a cryptographic public key used to sign the data. **Creation** and **Expiration** are timestamps, specified as milliseconds since Jan 1, 1970 00:00.000 UTC (the Unix Epoch). Creation marks when the data is first "valid" and Expiration identifies its invalidation. Because values like temperature or location might change frequently, these two points help to establish a chronological record of changes in state. **OnDemand** is a flag to indicate whether data is On-Demand or not. On-Demand data should not be permanently stored by the World Model, and is only produced in response to its request. The last element, **Value** is the binary representation of the Attribute data. Its encoding is entirely specified by the Origin, and so must be documented externally from the system.

2.7 Protocols, Features, Communication

The overall design of the system is a set of interacting layers, each providing a limited interface between those layers above and below it. In order to support independence between layers and components within those layers, it was decided that each layer interact would with another via a network interface. A prototype implementation has been written in TCP/IPv4, which means the system can be supported by a wide range of computers in use today from smart watches to servers. The choice of TCP also reduces the need to implement reliable data transfer, and protocols like SSL and TLS can also be used to provide end-to-end security for data in transit across public networks. Beginning with the Sensing Layer, the WSN gateway to the system establishes a TCP/IP socket to the Aggregator and begins pushing data as it is generated by the sensors in the network. Solvers also connect to the Aggregator via TCP to receive the sensor data in real-time. The World Model provides TCP/IP connections for both the Solver and Application interfaces and these remain distinct to prevent the "blurring" of the roles of each layer.

2.8 Aggregator

The Aggregator serves to produce a single point of conctact for all data coming from the many disparate Wireless Sensor Networks that are connected to the GRAIL system. It provides this stream of data to Solvers which request it, filtering only by Physical layer, Transmitter Identifier, and relative timing between data arrivals. Focusing solely on aggregation and simple filtering allows the Aggregator to provide these functions with low-latency and at significant scale.

WSN systems vary significantly in structure, protocols, data representation, and goals. If each Solver needed to know how to interpret each different format, or how to retrieve it from the WSN, the complexity of a large-scale multi-WSN system like GRAIL would quickly make application development nearly impossible. To address this problem, a simple format was developed which would unify the most common parts of the data and leave the proprietary details to be worked-out at the Analysis layer.

Each data point, or **Sample** is a 6-tuple **{Timestamp, PHY, Transmitter ID, Receiver ID, RSSI, Data}**. **Timestamp** is the time at which the Sample data was observed by the receiver, specified as milliseconds since Jan 1, 1970 00:00.000 UTC (the Unix Epoch). The physical layer is identified by **PHY**, which is an 8-bit integer with a unique value for each PHY type supported by the system. At present, only a few PHY layers are supported. **Transmitter ID** and **Receiver ID** are the PHY-dependent identifiers of the transmitter and receiver, respectively, of the Sample. Both Transmitter ID and Receiver ID values are stored as 128-bit values, which is sufficient for all physical layers in common use today. The **RSSI** value is a floating point reprentation of the PHY-dependent RSSI value, reported by the receiving radio. As an alternative to the RSSI, some PHY layers may provide the actual amplitude of the received frame (typically the preamble sequence), and this may be substituted instead. The last component, **Data**, is the PHY-dependent sensor payload. Data may include data points recorded by the sensor, framing information for the packet, or a combination of both. The data is limited to 64 kB in size, which makes it sufficient for small amounts of sensed data and radio framing headers and trailers. For more substantial data, such as high-resolution images, audio, or video recordings, a Solver should be written to process these data and insert them into the World Model as appropriate.

The filtering mechanism in the Aggregator is very simple. When a connection is established to the Aggregator's Analysis Layer interface ("Solver connection"), the connecting Solver is expected to send a request for a specific data Stream. This request is called a **Subscription Request** is composed of a set of **Subscription Rules** specifying which physical layers and transmitters are desired. The Aggregator then responds with a similar message, the **Subscription Response**, which specifies what request will be honored. In most cases, the Response will be identical to the Request, but in the case of security or policy decisions, or even due to significant load on the Aggregator, a more restricted Response may be sent. The Solver can then decide either to accept the Response as sent, or terminate the connection. Immediately following the Response message, any Sample received by the Aggregator which matches the Rules in the Response will be forwarded to the Solver.

The Subscription Request message contains a set of Rules specifying which devices are desired, and at what rate data should be forwarded from the Aggregator to the Solver. Each rule is a 3-tuple containing **{PHY, {Transmitter Base,Transmitter Mask}*, Update Interval}**. The PHY value identifies which physical layer the Rule applies to, and matches the same PHY values from the Sample message format. A value of “0” for the PHY is used as a wild-card match on all PHY layers. The Update Interval is the minimum time (in milliseconds) between Sample messages forwarded to the Solver, for each Transmitter/Receiver pairing. For instance, if a WSN forwards Samples from 3 Receivers that each observe a Transmitter broadcasting every 100 ms, then 3 Samples would be sent each 100 ms. If the Solver wanted data only once every 250 ms, then the Aggregator would ensure at least 250 ms has elapsed between the previous Sample sent for each Transmitter/Receiver pair before forwarding the next-received Sample. Each Rule has zero or more **Transmitter Base** and **Transmitter Mask** values, which combine to form a matching framework for Transmitter ID values. The Aggregator will perform a bitwise AND operation to a Sample’s Transmitter ID and the Transmitter Mask. If the result of the operation is equivalent to the Transmitter Base value, then that Transmitter ID value is said to “match” the Rule, and the Sample is forwarded to the Solver (if the Update Interval permits it). If the result of the operation and the Transmitter Base values do not match, then the Sample is not sent to the Solver. Additionally, if no Transmitter Base/Transmitter Mask values are provided in the Rule, then all Transmitter ID values are considered to match the Rule.

2.9 World Model

As outlined in Chapter 2.6, the World Model is a component which receives, forwards, and (usually) stores the state of the system for use by Solvers and Applications. The World Model supports two main functions: search and data subscription/publication. Search is supported by using Regular Expression matching on Identifier values and Attribute names. Subscription/publication of data in the World Model is used for a combination of one-time data fetching and continuous push-based updates of values. These two main actions, provided across both Persistent and On-Demand data support a broad range of applications, discussed in greater detail in Chapter 4.3.

2.9.1 Identifiers vs. UUIDs: Searching or Hashing

One of the early decisions made when designing both the World Server/Distributor and the subsequent World Model was to support searching of Identifier values. Specifically, it was assumed that Solvers and Applications, written generally to support a certain type of WSN or data transformation, would not know which entities in the World Model either contained the data required, or which should be updated with newly-generated information. In order for these entities to be programmatically identified without site-specific configuration for each installation, some type of searching or matching functionality had to be implemented.

The first approach to solving this problem was the World Server. The idea was that entries in the World Server would have some form of human-readable (or at least parseable) URI values, which would then map to values in the Distributor. For example, the World Server would contains URI values which were more human-oriented, like "smith.house.refrigerator/temperature", and these values would map to more machine-centric values, perhaps UUID-based URI values or similar. As noted earlier, however, this separation created additional complexity and overhead, without gaining much in terms of efficiency or expressive power. In the end, the decision was made to merge the two components and enable searching directly over the Identifier values for each entity in the World Model. To support search, Regular Expressions were chosen because they are a powerful tool for matching against structured string values (URIs) and standard library implementations are available in many languages, obviating the need for reimplementation.

The decision to support Regular Expression searching is not without its drawbacks, however. The most obvious candidate for World Model persistent storage is a relational database management system (RDBMS), also chosen as the first two prototype implementations. Unfortunately, one of the more powerful features of most such databases is the use of Structured Query Language (SQL) and index-based searches on column values. With Regular Expressions, because each needs to be compiled and executed independently across all data, many of the most common string-based indexing techniques are rendered useless without careful design of the schema used for persistent storage. A more thorough discussion of both the advantages and drawbacks of using Regular Expression matching is provided in Chapter 6.5.

2.9.2 Snapshots, Ranges, and Standing Queries

Once a search has been performed and a set of Identifiers has been selected, the next step is ususally a query to retrieve some or all of the data for a subset of the matched Identifiers. The World Model provides support for three basic types of queries: Snapshot Requests which provide entity state at a single moment in time, Range Requests which provide a stream of sequential updates between two times, and Standing Queries which provide a real-time push-based update of values. Each is focused on a specific scneario for data retrieval, and in some cases all three are used in a single application.

A Snapshot query retrieves the state of some set of entities at a specific moment in time. One of the most common uses of this type of query is to retrieve the "current" status of the entities in the World Model. This can be used to quickly retrieve state for presentation to users, even while a range query is retrieving more data simultaneously. It is also helpful for systems that cannot support a standing query where data is pushed back to the requesting application. Traditional web-based systems often rely on data polling to simulate a continous connection, and mobile devices which cannot afford the energy costs of a continuous connection back to servers can also utilize this technique. In both cases, a very simple interface between an HTTP server and the World Model can be used to bridge the two systems. An example implementation is described in Chapter 4.3.

Range queries provide a stream of entity Attribute updates between a specified pair of timestamps, enabling an application to view the state of entities over a period of interest. One of the more common use cases for this type of query is for report generation, search result presentation, and data aggregation or summary. It can also be useful for systems that do not have continuous access to the World Model, and so can perform "batch" updates of data when a connection is established.

Standing queries provide a stream of Attribute updates starting from a specified timestamp and continuing until the query is cancelled by the application. As each Attribute is updated by Solvers, the World Model matches the Identifier and Attribute name values to the query, and forwards the data to the requesting Application. This is one of the most commonly-used queries for Solvers, as it enables them to perform asynchronous data-driven analysis and updates. As a result, a single initial entry into the World Model, perhaps based on raw sensor data, can trigger a number of subsequent updates in related Attributes as each Solver receives the input data it needs to produce its output. In applications, this type of query can be used to provide a type of "cache" of up-to-date values, including a limited history, to a large number of clients. For example, this type of caching behavior can be very useful for more complex web-based applications by using an application server to handle load balancing when a large number of clients are connected and requesting data.

Chapter 3
Related Systems

The challenges of collecting, organizing, processing, and presenting data about the physical world have resulted in the proposal and development of many differing systems. These include proof-of-concept academic demonstrations, well-established commercial products, and ad-hoc assemblies of different commercial products by hobbyists and each has viewed these problems from slightly different angles [15, 25, 24, 35]. Personalized home helpers, enterprise task automation, and global collaboration all have motivated the development of systems designed to collect, interpret, and react to data in the real world. Most, however, are focused on a specific challenge or domain that they attempt to optimize: from synthesis of inputs from all over a single home to aggregation of weather and seismic data to assist researchers.

3.1 Vertically Integrated Systems

Most of the earliest systems built around wireless sensor networks were fully integrated with the hardware they used for sensing. As with any technology, it is reasonable that a "vertical silo" is the first modality used in developing it, especially with research prototypes and proof-of-concept commercial systems. Until the challenges facing such systems are revealed by the pioneering technologies, it is not easy to partition them into distinct components and produce modular systems with specialized components functioning together.

One of the earliest related works in this field is the Active Badge system, developed in 1990 by Xerox Parc and Olivetti Research Ltd [38]. It focused on a system of infra-red (IR) "badges" worn by staff that would periodically beacon an ID to a network of receivers embedded in the environment. A central computer then processed these beacons and produced location information for each badge, and consequently the staff member who wore the badge. The system was used by receptionists for routing calls, colleagues for finding one another, and to keep track of visitors. The system was entirely focused on determining the location of staff and using that data to make day-to-day operations more efficient. It was even integrated with the telephone system to automatically route some telephone calls directly to the handset nearest a person.

Several years later, the Active Bat indoor location system was also developed in Cambridge in collaboration with one of the original authors of Active Badge [1]. This system was also focused on indoor location, and utilized ultrasonic beacons and sensors rather than IR to provide a more precise position estimation. The result was a system that could do everything found in its predecessor and more, including automated workstation configuration, telephone routing within a dense network of cubicles, and tracking of objects and supplies like personal digital assistants (PDAs). Like its predecessor, it employed a nearly monolithic design focused almost entirely on location data. It supported many of the same applications as Active Badge including staff location tracking and automatic telephone call routing. It also added new features like audible notification of emails and voicemails, and repurposing the badge itself to act as a computer mouse on large display screens.

Of particular note with Active Bat was its use of a "world model" and use of "...sensors to update a model of the real world." This concept, while not original or unique to the Active Bat systems, was very important, even if it focused almost entirely on location data. The combination of the high-precision badge location data and functional state and location of the office equipment, made the system much more flexible than its predecessor. Use of Common Object Request Broker Architecture (CORBA) enabled generalized modeling and interaction with a number of devices in the system, making it flexible and extensible for future expansion.

The early 2000's saw an increase in the number of low-cost wireless sensor devices and an increased interest in associated operating systems, networking algorithms, power management techniques, and administration and data collection systems [20, 2]. Among these systems, two dominant techniques for data analysis can be observed: those that task the network with performing data analysis, and those that analyze the data externally. The choice of where data analysis is performed often depends on the processing, memory, communication resources available to the network [34, 41].

3.2 Tightly-Coupled Systems

Systems that tasked the network itself with the job of efficiently collecting, organizing, and analyzing sensor data assumed that the sensors had relatively sophisticated software and hardware resources. The nodes in these networks could perform relatively complex operations like resource budgeting, query optimization, and dynamic network routing. For instance, Intanagonwiwat *et al.* proposed a technique for query and data dissemination in networks where nodes had bidirectional communication and local storage available [19]. Likewise, Madden *et al.* describes a software platform running on an embedded operating system where nodes work together to execute database-like queries for data [26]. Using the same operating system, Woo *et al.* designed a spreadsheet interface that enabled users to reprogram nodes, issue queries, and process data from sensors [39].

The authors of Active Bat acknowledged that their system would not be commercially feasible because of the high costs of both the badges and the large number of sensors needed to be installed for accurate positioning. The researchers behind MicroSoft's SixthSense [33], however, attempted to solve the same problem by using passive RFID badges and readers spaced throughout an office environment. Although the cost of RFID readers is relatively high, at several thousand US dollars per unit, the cost of the tags themselves is well under one dollar. With this in mind, the authors proposed and demonstrated a proof-of-concept system that enabled tracking of individuals and objects, ownership inference, identity inference, and automatic reservation of conference rooms. The system is a collection of tightly-coupled components and provides an API for applications to perform lookups and register callbacks for targeted events.

With this system, RFID readers are deployed in a non-overlapping pattern throughout an area of interest. Each time an RFID tag is read by a reader, the reading is entered in a "raw database" with the tag's identifier, the reader and antenna, and the timestamp. Other enterprise data, including user schedules, keyboard logins, and keycard swipe logs are also entered into this raw database. A set of "inference engines" then process this raw data to create information including where a user is, who they are with, and what devices they are carrying, and place it into a "processed database". For instance, a series of RFID tag reads over a period of an hour in the raw database might be transformed into a single entry in the processed database consisting of a starting and ending timestamp to indicate when the tag was first and last in the area covered by the same reader. For outside application access, an API engine proxies access to the processed database, performing lookups, caching, and invoking callbacks in applications based on data produced by the inference engine and requested by the applications.

3.3 Loosely-Coupled Systems

Other systems have also interacted with the sensor nodes, but in a more abstract and generalized way. These systems viewed the sensors, or even the entire network, as having a set of properties and controllers which could be generalized. Applications could be built on top of these generalized interfaces which were capable of exploiting highly heterogeneous networks and collections of networks.

One system, SenseWeb [16, 21] utilized the emerging ubiquity of the World Wide Web, specifically web services, to build a coordinated system for integrating sensor data. The system has a simple layered design centered around a "coordinator" that performs a number of critical tasks including data indexing and archiving, access control, and coordinated and optimized access to sensors with data needed for multiple independent applications. The system is designed to operate in such a way that sensor owners can enforce access policies to both their sensors and sensor data. This enables SenseWeb to interact across multiple administrative domains without a strong trust in the operator of the common components, including the Coordinator.

Providing data to the coordinator are a number of "sensor gateways", which transform a device-independent interface, exposing common capabilites. The sensor gateway in turn converts the instructions from the coordinator into device-specific commands, possibly exercising any appropriate policies set by its owners or operators. For example, a weather station installed atop an office building may provide its operator with data every 15 minutes, while only permitting data to be sent to the public Coordinator every 4 hours. Likewise, images captured by a camera may be time-delayed to allow its operators to redact sensitive images before being made available via the sensor gateway.

With an application programming interface (API) based on web services, application developers can rapidly develop applications that interact with SenseWeb by re-using existing HTTP libraries. This, combined with standardized set of operations exposed by the Coordinator enables rapid development of many different applications. Unlike Active Badge and Active Bat, however, SenseWeb is focused on very large geographic areas: cities, mountain ranges, countries, or the Earth. Every sensor is associated with geospatial coordinates, either manually entered for fixed devices or dynamically generated from GPS satellite data. Unfortunately, GPS performs consistently poorly when satellite signals are weak, particularly inside buildings where humans spend the majority of their time. This type of system, which supports a global range of position values does does not translate well into indoor spaces like offices and homes.

With similar automation-enhancing goals as SenseWeb, but focused entirely on the concept of a "smart home", a home where sensors and computer systems are used to enhance and improve the experience of its residents, HomeOS provides an "operating system" abstraction to interface with a network of heterogeneous devices [9]. The system is designed to provide non-technical homeowners and residents the ability to manage sensors, cameras, audio-video systems, and security systems, combining them with various applicaitons to produce new functionality. Examples include light control based on presence detection and location, music that follows people from room to room, and automatic notifications to email or mobile telephones. Composed of 3 main layers (in addition to the "application layer"), the system is focused primarily on management of devices and exposing an abstracted API to application developers.

Interacting with the applications is the "Management Layer", which is responsible for access control, application installation and configuration, device installation and configuration, and the user interface. The Management layer interacts with a "Device Functionality Layer" which uses abstracted interfaces for devices based on their functionality. For instance, a table lamp may implement a "light source" role which has 3 functions: *on()*, *off()*, and *isOn()* $\rightarrow$ *Boolean*. Similarly, a color-changing LED lamp might also implement a "colored light source" role, which has 2 additional functions: *setColor(Color)* and *getColor()* $\rightarrow$ *Color*. In effect, each device becomes the composition of the set of roles that it provides to the system. Because each device may have a different set of interfaces or instructions that it accepts, the Device Functionality Layer interacts with the Device Connectivity Layer which implements any device-specific protocols, instructions, and functionality.

The result is a centralized system that, so long as drivers are available for the desired devices, enables users to configure and control their home in novel ways. Application developers are also able to rapidly develop applications based on functional roles rather than specific lists of devices or model numbers. This means that applications don't need to be updated because new devices are available, or because a device provides additional roles that were not previously implemented.

For access management, HomeOS utilizes user accounts and permissions as the two primary security concepts. Users are arranged hierarchically into a set of groups, and each group or user is provided with a set of permissions for an individual device or group of devices. Users are arranged in a logical tree hierarchy, while devices are arranged in a hierarchy by "rooms". A special high-security device group is used to allow easy grouping and control of devices considered particularly security-sensitive like cameras or door locks. When new devices are installed, the system prompts the user to configure access to the device, for both users and applications, and when new applications are installed it similarly asks the user to configure access.

3.4 Decoupled Systems

In contrast, others designed systems that agnostically used WSNs as a source of relatively simple sensor data, and addressed the problem of data collection, processing, and storage. These systems rarely made assumptions about the capabilities of the sensors, and instead assumed that the networks would produce data over time. Owl Platform is one such system, and similarly makes very few assumptions about how the networks will perform. All of these types of systems refrain from considering how commands are issued to the network or how data exits the network, and focus almost entirely on how to process the data once it is available outside the WSN.

Most examples of this type assume that data processing is handled by a collection of distributed servers which are primarily tasked with transforming data from one form or another. For example, pressure sensor readings on a roadway might be transformed into information about the number and type of vehicles passing through an intersection. Cherniack *et al.* compare two proposed systems, Aurora* and Medusa, which are primarily concerned with how to distribute data processing tasks among a set of distributed processing nodes [7]. Cao *et al.* present and evaluate an algorithm called RfInfer for distributed processing of radio frequency ID (RFID) tag location and monitoring [5]. Their goal was to enable computation to be distributed among different processing units while minimizing the loss of data fidelity caused by incomplete information.

Like these systems, Owl Platform is focused on the management and transformation of WSN data once it has left the network. However, instead of defining a specific application or problem and then setting out to optimize a solution for that problem, it attempts to define a framework and architecture that permits many different applications to operate cooperatively on the same data. Given the architectural design choices of Owl Platform, it is possible that systems like Medusa* or RfInfer could operate like Solvers which focus on specific types of WSN data or application queries.

Chapter 4
Generality

4.1 Sensors and Sensor Networks

Sensors produce data; wireless sensors produce data wirelessly. With no other assumption about what a wireless sensor can do, Owl Platform represents data produced by wireless sensors in an extremely simple format called the Sample. A Sample is a simple tuple that identifies *Data* sent by a *Transmitter* and received by a *Receiver* across a shared textitPhysical Layer at a specific *Timestamp* with a specific *RSSI* value. Although there are many different types of sensors that produce different types of data in different formats and at different time scales, all of them will, at some point, transmit that data and it will be received by one or more receivers.

The format of the Sample overall and its components were chosen for two reasons: it can handle nearly all current types of sensor networks and data and is short enough to normally fit into a single IP datagram. With these features in mind, we can investigate the format of the Sample as presented in Table 4.1.

The **PHY ID** value is an unsigned 32-bit integer type which identifies the physical layer that the sensor employs. Given the ability to uniquely identify 254 possible PHY values (the 0 value is reserved), it can readily handle a large number of heterogeneous wireless sensor networks. In the event that a greater number of sensor networks are required, the size of the value may be expanded to 16 or 32 bits, providing 64 thousand to 4 billion possible values. Doing so would retain backward compatibility with the current format, with only the upper bits being extended, and would have a minor impact on the size of the remaining data able to be sent by the sensor. The PHY ID value is important to higher layers of the system, because it provides critical information about how to decode the remainder of the sample. It provides necessary context about how the RSSI and Timestamp values are generated, and how to parse and process the Transmitter ID, Receiver ID, and Data fields.

The **Timestamp** value is a signed 64-bit integer type, the format is expected to be a count of milliseconds since some event, and the value is generated by the receiver. For systems that have accurate "wall time" values, this value can be used to represent the current time in milliseconds since an epoch. In systems without accurate clocks, or where even releative timing is imprecise, this value can be used primarily to guage the inter-arrival time of sensor data to a specific receiver. For example, the low-energy Pipsqueak sensors use a digital oscillator to track time between transmissions, resulting in timing variations up to 20 or 30 percent between units [11]. However, even in this case, device-dependent variations are very low, typically around 2–3% and this information can be used to detect wireless collisions or even identify when sensors stop broadcasting due to failures.

The **Transmitter ID** is a 128-bit binary value that, together with the PHY ID value, uniquely identifies a transmitting device. This size was chosen because although a number of currently-available systems use 48-, 64-, or 96-bit identifiers, 128-bits is sufficient to address every atom in the known universe, and should be sufficient for WSNs anticipated in the next several decades. In addition, it enables Owl Platform to view the Internet of Things (IoT) using IPv6 addresses as a WSN. In general, the Transmitter ID value should uniquely identify a sensor, though it may instead uniquely identify a *radio* on a sensor. In the event that a sensor has multiple radios, for instance to assist in self-localization of a WSN, the Transmitter ID of that radio should be sent instead. This allows exploitation of radio propagation effects at different angles, polarities, or power levels to be analyzed and exploited in the Analysis layer. Likewise, the **Receiver ID** is a 128-bit binary values representing a unique radio or sensor of a specific physical layer. Just like Transmitter IDs, if a receiving device has more than one radio, then a unique Receiver ID should be used for each.

The **RSSI**, or Received Signal Strength Indicator field is a 32-bit floating point value that is intended to capture the RSSI value produced by the radio hardware. If the RSSI value is not made available by the hardware, another proxy for the power level of the received radio transmission may be used instead. If the physical layer provides no indicator for the received signal power, nor any type of link quality indicator, the value may be set to *NaN* (Not a Number) so that Solvers in the Analysis layer do not incorrectly interpret the value in this field. RSSI values are frequently used to assess the likelihood of a corrupt packet, and also to perform active and passive localization, motion detection, and mobility detection. In general, because the RSSI value produced by a radio is strongly associated with the way in which it is computed, RSSI values cannot be compared across different physical layers. For example, the RSSI values of a physical layer where all transmissions are made at the same power level should not be compared to the RSSI values of a physical layer where power optimizations are made by adjusting the power used in transmission.

The final field in the Sample, the **Data** value, is the physical-layer specific data transmitted by the sensor. For physical layers that only beacon an Identifier, this field may be empty while other networks may process data from several sensors within the network, and then transmit in a combined or compressed form to a sink node. One of the critical functions of Solvers in the Analysis Layer is to decode the data stored in this field and trasnform it into a format usable by other Solvers or by Applications. Data may include temperature or humidity readings, output from seismometers or geiger counters, or even error-checking values like CRCs or checksums.

PHY ID	Physical layer identifier	8-bit unsigned integer
Timestamp	Receiver-generated timestamp	64-bit signed integer
Transmitter ID	PHY-specific transmitter identifier	128-bit unsigned value
Receiver ID	PHY-specific receiver identifier	128-bit unsigned value
RSSI	Hardware-reported RSSI or equivalent	32-bit signed floating point
Data	PHY-specific binary data	Unstructured, max. length $2^{32} - 45$ bytes

Table 4.1: Description of WSN Sample format

4.2 Solvers and Data Transformation

Solvers are independent components within the Analysis Layer that transform data either by converting it from one format to another (e.g., Fahrenheit to Celsius temperature transformation), combining it with other data source to produce new information (e.g., combining security access logs and passive motion data to track individuals), or a combination of the two. Within the design of the Owl Platform, Solvers are expected to perform a limited set of data transformations with the expectation that by combining multiple independent simple Solvers, complex analyses can be performed.

One of the first transformations to take place is decoding the raw data produced by a wireless sensor. This type of solver might connect to one or more Aggregators and subscribe to an entire PHY layer of sensors. As each Sample is received, the Solver decodes the Data segment and inserts Attribute value updates into a World Model, as appropriate. In the most limited form, this type of Solver would have no additional context about the sensors producing the data, and so might assign the Attribute values to an intermediate Identifier value in the World Model. For instance, an RSSI averaging solver for Wi-Fi devices might use an Identifier structure such as *PHY_ID/BAND/CHANNEL/TX_ID/RX_ID*, where *PHY_ID* is the identifier for the Wi-Fi physical layer, *BAND* is the frequency band of the device (e.g., 2.4 or 5 GHz), *CHANNEL* is the logical channel of the transmission (e.g., 1–14), and *TX_ID* and *RX_ID* are the Transmitter ID and Receiver ID respectively. This format would allow other solvers to easily search for devices and retrieve the values produced by the solver, so long as the pattern was well-known. Other types of Solvers that operate on raw data produced by the sensors might focus on RSSI statistics, produce a topological map of the sensor network, or analyze delay and jitter.

The second type of Solver would not require a connection to Aggregators, instead using Attribute values in the World Model to produce new types of data. The simplest of these solvers might be to bind the sensor data to a more useful context for applications or other solvers. As an example, a very simple Solver of this type might take manually-entered values for the position of sensors attached to doors, windows, vehicles, or personnel, and automatically copy Attribute values from the sensors to those Identifiers. This type of Solver may also make use of external data sources like business databases or account information to automatically bind sensors to context. This Solver performs integration of multiple Attribute values to produce new information. For example, if personnel carry ID badges with temperature sensors outside their clothing, then this temperature information could be combined with localization data to provide a temperature proxy for rooms without dedicated sensors. The data would no doubt be less accurate than a dedicated temperature sensor (because it may be affected by body heat), but may be better than no data for many applications.

Independent, single-focus Solvers combined with the network-based interfaces of the Owl Platform produce a system that can scale easily as more sensors or sensor networks are incorporated into it. So long as there is network connectivity between them, each component (Aggregator, Solvers, World Model) can reside on independent hosts. Components that require more resources, for instance localization algorithms requiring large numbers of trials, may starve one another for resources if hosted on the same physical machine [22]. A drawback of this decoupled approach, however, is an increase in the amount of "out of channel" information needed to communicate what data is available and how it is generated. In general, it is not expected that this requirement will place a restrictive burden on solver or application development for the Owl Platform. Much like the GNU philosophy of Unix-like utility applications, having a large number of single-purpose applications allows them to be well-designed, maintainable, and able to focus on the selected problem each addresses.

4.3 Application Types and Requirements

Unlike purpose-built systems design to focus on a single application or application domain (e.g., localization, weather monitoring, building security), Owl Platform does not assume a specific application goal *a priori*. Instead, the World Model exposes a limited set of queries that correspond to frequent application data requirements. In general, the World Model exposes two basic data queries: a point query and a range query. The point query returns information about the system at a single point in time. Any valid Attribute value, one with a creation timestamp before the query timestamp and an expiration timestamp after the query time (or unspecified), is matched and returned. This type of query can be used to initialize a user application on start-up, use polling to simulate a continuous stream of updates, or to determine the state of the system at some point that matches an external event. The Range query returns a set of matched Attribute value updates, ordered by the Creation timestamp values, for all values that are valid after the beginning and before the ending timestamp values defining the range. A range query can be used to provide "playback" of values over time, to fill an application-specific cache and reduce load on the server, or to produce an aggregated data point (mean, min, max, etc.). A special case of the Range query is the *Standing* query, which is a range query with an unspecified ending time value. A Standing query begins like a Range query by returning the matched values starting with the beginning timestamp, but never closes the returned set and instead sends any updated Attribute values matching the other query parameters as they arrive in the World Model. Standing queries provide true "push" mechanisms so that the requesting Solvers or applications can be written asynchronously to respond to data as it arrives.

The World Model does neither enforces any data types for Attribute values, nor does it include any documentation about the data type (e.g., MIME types). Applications that use Attribute values from the World Model need to manually decode the data from a raw binary form. This approach enables the World Model to support a broad range of data types from simple scalars to polygons to multi-dimensional vectors. While this can put additional burdens on the application developers, it is relatively easy for Solver developers to document the data types and formats produced, along with the Origin values used as "signatures" for these Solvers, so that consumers can easily decode the values. For instance, a Solver that produced device-free localization results divided the monitored space into a set of rectangles, and the output of the Solver was one or more of these rectangles that indicated a likely area of mobility [30]. When an application was written to display these values, it required reading the documentation for the Solver to determine what the binary values represented, and so how to display the results. While this approach does work well for a limited number of Attribute values and data formats, in general it does not scale well. For instance, if a developer fails to maintain accurate documentation after changes to the values, or if the server hosting the documentation has a failure, then at some point it may no longer be possible to know how to interpret the data. This limitation, although significant, is not much different from many existing programming languages and toolkits available today.

A relatively simple solution to this problem might be to encourage Solver developers to produce a "special" Attribute value that contains human-readable documentation about the data type the solver produces. The Identifier could follow a standard pattern, something like *Origin ID/Attribute* *name*, which would allow other developers which see the value in a World Model to be able to easily determine its format. The solution previously attempted in the GRAIL Distributor was to include a field that documented the data type with a "standard" name for each type. This approach is not unlike MIME types for content sent over HTTP connections, but it became increasingly difficult to maintain because some Solvers (like the passive localization one above) produced complex data and it was not possible to easily format this data in a machine-readable way. Consequently, the World Model does not enforce the requirement of documented data types, and instead relies on developers to agree on a standard externally. In practice, this approach has worked relatively well, as long as the Solvers and Applications are a combination of well-documented open-source tools and proprietary tools used within a single administrative domain (like a company or organization).

Chapter 5
Aggregator Performance Evaluation

The Aggregator is the first potential bottleneck limiting the overall performance of Owl Platform. It acts as a focal point for all data arriving from Wireless Sensor Networks, and the effects of its limitations will be felt throughout the rest of the system. Chapter 2.1 described a large University campus as being the driving example of the scale at which the system should perform efficiently. Such a campus might include 100 buildings, averaging 30 rooms each, and 5-10 sensors per room. Across the campus, 30,000 sensors updating every 10 seconds would generate 3,000 Samples per second. Add to that another 50,000 students and staff with 3–4 sensors each, and the rate can grow to 10,000 Samples per second. Other examples could include hotels, sports stadia, municipal complexes, and shopping malls. For example, the Izmailovo Hotel in Moscow has nearly 7,500 rooms in four buildings [36]. If each room is a 3-part suite (bedroom, wash room, entrance), and each part has two sensors (one on the wall to measure temperature and light, one on the floor to detect leaks), the hotel would have around 22,500 sensors. With each sensor reporting every 30 seconds, an Aggregator would typically receive 1,500 Samples per second. During an emergency, this might increase to 45,000 per second. In New York's Pennsylvania Station, more than 600,000 passengers pass through the halls every day [32], around 7 every second. If each person were carrying 3–4 sensors each, and these reported frequently (once per second), then an average of 28 Samples would be generated each second. Add to that another 10,000 sensors spread throughout the facility, and a reasonable load of 350–3,500 Samples per second could be expected. In each of these cases, bursts of higher rates are possible, but the average rates provide a reference point when determining what level of performance might be acceptable.

To measure the performance of the Aggregator under different load types, and in different system configurations, a series of experiments were designed and conducted to determine its capabilities. The experiments began by measuring the maximum number of Samples that the Aggregator can handle in a short period of time, providing an upper bound on its abilities. The number of Sensor and Solver connections were scaled up to determine what impact they have on Aggregator performance. Finally, the number of Subscription Requests was increased to provide a realistic workload for an

Aggregator in one of the scenarios described above. The ultimate goal of these experiments was to measure the rate of processing, latency introduced, and areas needing improvement in the future.

5.1 Summary of Results

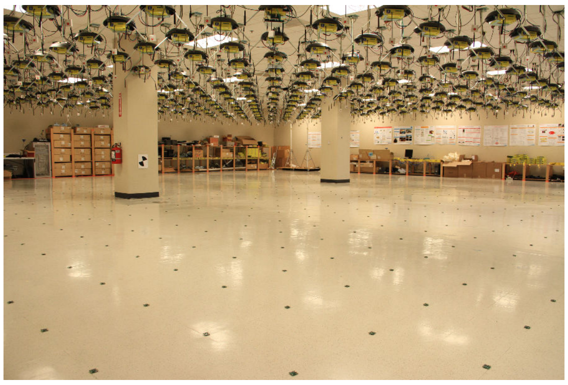
Table 5.1 provides a summary of the performance evaluation of the prototype Aggregator for the different experiments performed. For each row, the maximum average service rate in Samples per second was determined and the average processing time and Sample lifetime for that data point were selected. Although it does not expose some of the more interesting details explored in this chapter, it shows the general trend of decreasing throughput as the amount of per-Sample work increases.

		Avg. Service Rate (Samples per Second)	**Avg. Processing Time** (Per-Sample μs)	**Avg. Sample Lifetime** (Per-Sample μs)
Single Sensor		174,142	0.286	3,585
Multiple Sensors	3	401,731	0.383	5,613
	6	658,196	0.233	418
	9	515,069	0.577	14,039
“All Data” Solvers	1	488,652	2.036	7,862
	3	477,618	0.452	6,120
	6	467,903	0.442	6,129
	9	466,619	0.457	6,192
“Selection” Solvers	1	222,587	20.040	732
	3	166,233	36.465	2,222
	6	83,500	77.963	1,199
	9	69,763	168.627	3,378

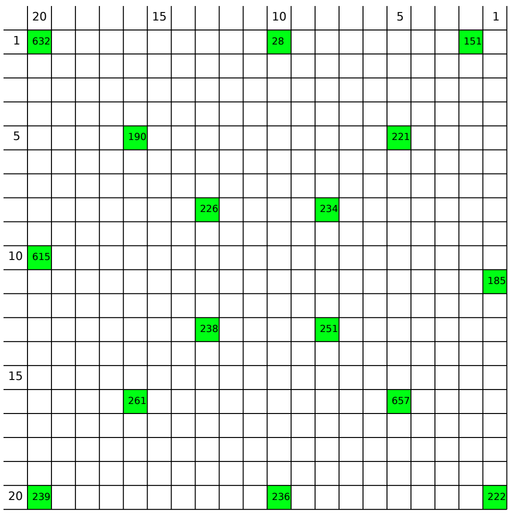
Table 5.1: Summary of Aggregator performance results for selected service rates. Maximum service rates for each class of experiments were selected and the corresponding processing time and Sample lifetime values are shown.

5.2 Experimental Methodology

In order to determine the performance limits of Owl Platform, a testbed system was set-up using the ORBIT testbed. 576 Pipsqueak sensors were arranged on the floor of the ORBIT testbed facility in a 24 by 24 grid, with two feet of spacing between each sensor (Figure 5.1a). 16 Pipsqueak receivers were installed in the ceiling-mounted ORBIT nodes, arranged in the pattern shown in Figure 5.1b. Each transmitter was configured to broadcast a minimal packet containing a 4-byte preamble, 4-byte sync, 1-byte length, and 3-byte ID at 250 kbps (384 μsec transmisson time).



(a) Sensors at the ORBIT testbed facility arranged in a 24x24 regular grid on the floor. The computers overhead are ORBIT nodes used for the experiments.



(b) Arrangement of Receivers in the testbed.

Figure 5.1: Arrangement of sensors and receivers during the ORBIT testbed experiments. Sensors are arranged on the floor in a grid pattern with a 2 foot spacing (Figure 5.1a). Receivers are installed in the overhead nodes to capture wireless packets, with the arrangement shown in Figure 5.1b. Green squares indicate a node with an installed receiver and the numbers are the PHY layer identifiers for each receiver.

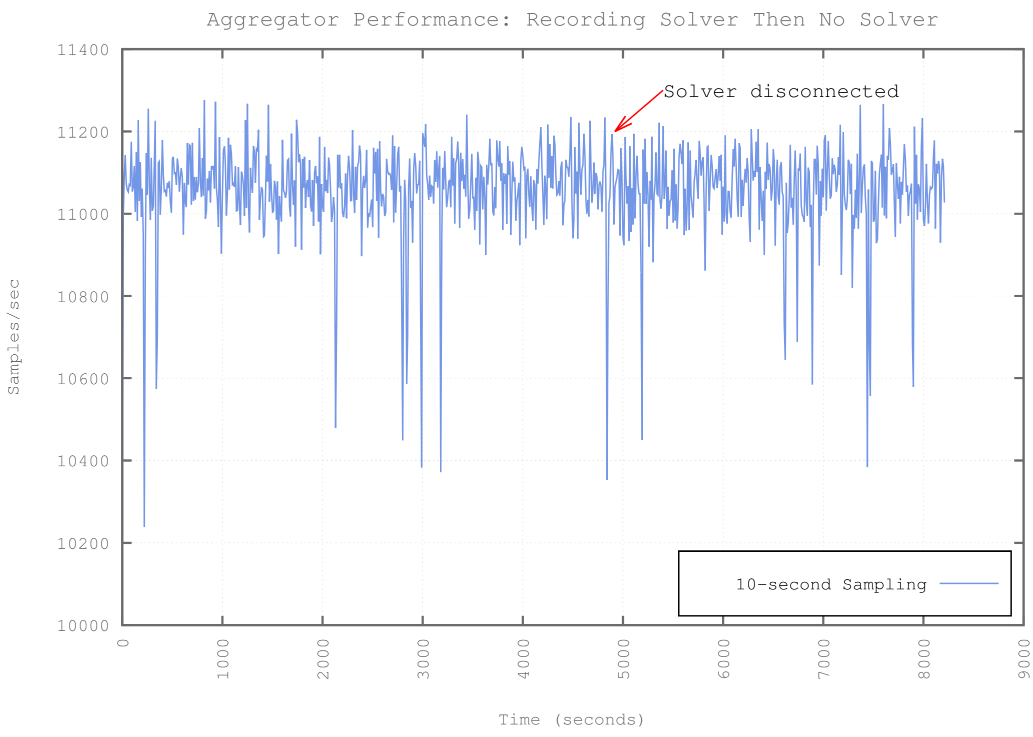


Figure 5.2: Sample throughput in Aggregator from 16 receivers and 576 sensors, each broadcasting 5 times per second.

The ORBIT testbed contains 400 nodes connected by a network of 48-port 1Gbps ethernet switches. Every receiver was configured to forward all received sensor frames to a single aggregator over the network as Samples. The aggregator had a single Solver connected to it, which recorded all Samples to disk. The system was allowed to run for approximately 80 minutes, after which the Solver was disconnected and the Aggregator and Sensors ran for another 55 minutes. The performance of the Aggregator over these two periods is reproduced in Figures 5.2 and 5.3. Assuming that each Transmitter's packets could be received by each Receiver, the estimated number of Samples per second should be $576 \times 16 \times 5 = 46,080$. However, the observed rate of approximately 11,000 Samples per second indicates a reception loss rate of approximately 76%. This level of packet loss is not abnormal given the high density and broadcast rates of the sensors [13].

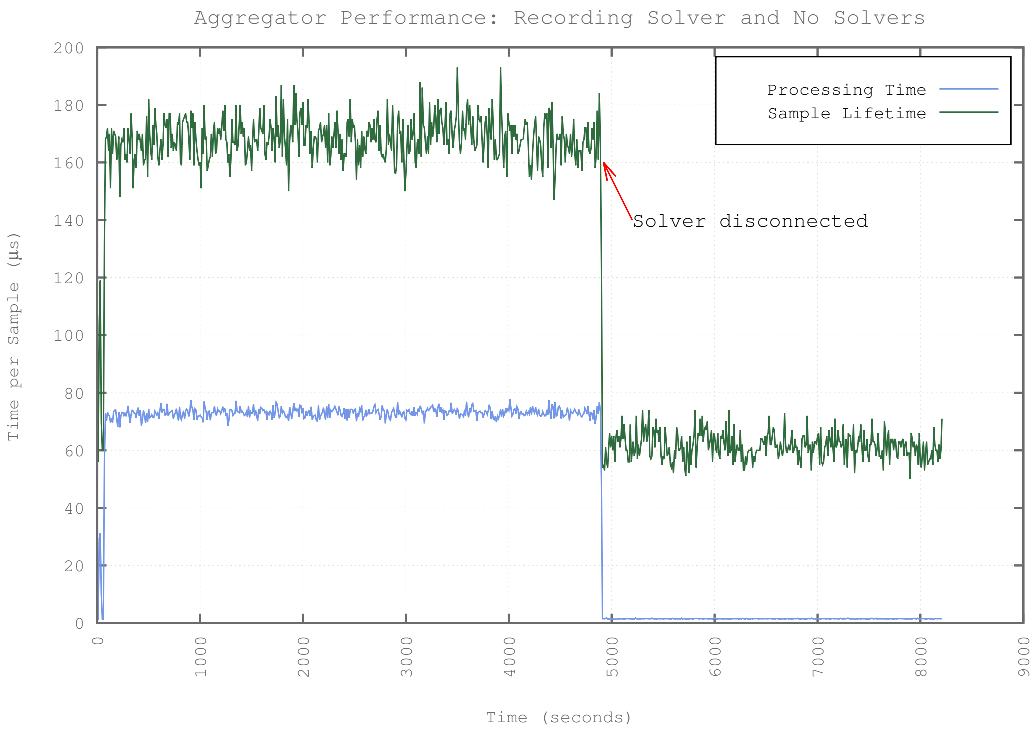


Figure 5.3: Processing time and Sample lifetime in Aggregator from 16 receivers and 576 sensors, each broadcasting 5 times per second.

5.3 Simulated Workloads

This collected data was used to drive a number of simulated workloads for the Aggregator, to provide an estimation of how effectively the Owl Platform system can perform at different scales. Simulations were also performed on the ORBIT testbed, with a single Aggregator, 1–9 Sensors, and 1–9 Solvers, each hosted on a separate physical host. Each host had an Intel i7-3770 CPU @ 3.40 GHz, 4 GB available RAM (16 GB installed with a 32-bit operating system), hard disk drive with 3 Gbps serial ATA (SATA) interface, and 1 Gbps ethernet card for experimental use. Each host ran a 32-bit version of Ubuntu 12.04 LTS with the CPU scaling governor set to “performance”, locking the CPU at 3.4 GHz, and all swapping (HDD-based memory) disabled. Sensors would load and send the 80-minute recorded trace file, and Solvers would request the data from the Aggregator. The number of Sensors, rate of Sample transmission (replay rate) from each Sensor, number of Solvers, and Subscription Requests from each Solver were varied.

The Aggregator reported statistics about the average service rate (Figures 5.4, 5.7, 5.12, 5.18), average Sample object lifetime (Figures 5.6, 5.9, 5.14, 5.20), and average Sample processing time

(Figures 5.5, 5.8, 5.13, 5.19), each computed over a 10-second period. Average service rate was computed by incrementing a counter each time a worker thread selected a Sample object from a queue, and this counter was recorded and reset every 10 seconds. Average Sample lifetime was computed by timestamping each Sample object when it is created by the Apache MINA library decoders, recording the timestamp when a worker thread finished processing the Sample, and taking the difference of the two. The sum of these differences was divided by the total number of Samples processed to produce an averaged value. The Sample Lifetime does not include the time a Sample waits in a Solver output buffer, as this buffer is limited to 200 messages before writes are dropped. The cost to instrument the Aggregator to include this delay in computations required the inclusion of reference counters and multiple callbacks for each Sample to be written, which would result in additional computational resources without a significant gain in lifetime accuracy. Average processing time was computed by recording the timestamp when a worker thread selected a Sample from the a queue, using the same finishing timestamp as the Sample Lifetime computation, and computing the difference. The sum of these differences was divided by the number of serviced Samples to produce the average processing time.

In addition to the statistics produced by the Aggregator, an informal measurement of bandwidth used during experiments was collected using the *jnettop* tool to measure per-connection and aggregate bandwidth observable within the operating system. The bandwidth was recorded in megabits per second (Mbps) and kilopackets per second (kPps) during relatively stable points in the experiments (Tables 5.2, 5.3, 5.4, 5.5). For some experiments where the transmission rates were less stable (e.g., Figure 5.4), an approximate value is provided. These values are provided for a qualitative explanation of how the Aggregator impacts the network and should not be considered suitable for benchmarking purposes.

5.4 Testbed Characterization

Prior to the simulated experiments described above, the network between a pair of ORBIT nodes was tested to determine its capabilities. Two tests were performed: a bandwidth test and a ping test. The bandwidth test was performed using the *Iperf* tool. Iperf requires a server process to be running on one host, and a client process run on a second host. It will establish a TCP connection between the two hosts and attempt to transmit a large amount of data between the two hosts. The result of this test between ORBIT nodes "node1-1.grid.orbit-lab.org" and "node1-2.grid.orbit-lab.org", which reside on the same ethernet switch, was a measured 935 Mbps (935,000,000 bits per second). This confirms the capability of the hosts to transmit over gigabit links and the network to sustain gigabit speeds, at least for a single connection. The second test used the *hping3* packet analyzer tool to test how many packets could be successfully transmitted between the same pair of hosts. The tool generates a sequence of ICMP echo requests and measures how many are successfully acknowledged. The test involved reducing the number of microseconds between each ICMP packet until packet loss was 100%. With a 4 microsecond spacing between packets, the tool achieved 864,633 packets over 7.294 seconds (118,540 packets per second) and only 1–2% packet loss. With a 3 microsecond spacing, 100% packet loss was observed and faster rates produced the same failed result. This means that the network limit lies between 118,000 packets per second and 333,000 packets per second (3 microsecond gap). Table 5.2 demonstrates that while the sending rate of a single process seems to be limited to approximately 125,000 packets per second, Table 5.3 demonstrates that the Aggregator host is capable of handling up to 316,000 packets per second. The difference between these two values may lie in the operating system's network stack implementation. In fact, much of Cisco's literature about the capabilities of their enterprise-level network switches indicates a hardware limit in the millions of packets per second, and so these measured limits may be caused by application software, operating system drivers, or network interface cards.

5.5 Aggregator Input Limits

The Aggregator receives Sample messages from multiple wireless sensor networks as they are generated. This may be the result of a poll request to the wireless sensor network, as in the cases of Z-Wave networks (devices that do not support "Instant Status/Hail") and Microsoft's SenseWeb, or if they are automatically generated by the sensor as in Roll-Call and Z-Wave "Hail" devices. In either case, the Aggregator sees the data as being "pushed" rather than requesting it from sensors. The first limiting factor for the Owl Platform will be how many Samples it can handle without performing any action on the Samples. Each Sample that arrives over the network is decoded, allocated as a Java heap object, assigned to a queue, and eventually handled by a worker thread. If there are no Solvers connected to the Aggregator, then the worker thread does nothing other than a small amount of bookkeeping to collect statistics and then the Sample is discarded.

The first experiment was to measure how fast a "Replay Sensor" could send Samples to the Aggregator, and then measure how well the Aggregator performed. The Replay Sensor is a debugging/testing program which loads the output of the "Recording Solver", and sends it to an Aggregator. The rate at which the Replay Sensor sends Samples is configurable with a command-line parameter at start-up. To conduct this experiment, and all the others to follow, an Orbit Management Language (OML) script was created that would issue instructions to the Orbit Management

Framework (OMF). OMF is a set of servers that control power, disk image storage and loading, and process execution for the ORBIT testbed. The script configured two nodes to share an IP subnet, one would host an Aggregator and the other would host a Replay Sensor. Each host was loaded with an identical disk image, on identical hardware (described in Section 5.3). The aggregator was started first, and the Replay Sensor was started 10 seconds later. The system then waited for 360 seconds (6 minutes) while the Replay Sensor sent Samples from the trace file, and the Aggregator periodically logged performance statistics. After 360 seconds elapsed, all processes were shut-down and the Aggregator log was saved. This process was repeated with different replay speeds for the Replay Sensor, and the Aggregator statistics were recorded for each.

Sensor Rate	**Agg. Rx Mbps**	**Agg. Tx Mbps**	**Agg. Rx kPps**	**Agg. Tx kPps**
1X (11 kS/s)	8.0	0.4	11.0	.6
2X (22 kS/s)	20.0	0.6	21.1	1.1
3X (33 kS/s)	30.0	1.0	32.0	2.0
4X (44 kS/s)	40.0	1.8	43.0	2.8
5X (55 kS/s)	49.7	1.5	54.0	3.5
6X (66 kS/s)	60.0	1.9	64.0	4.0
7X (77 kS/s)	69.0	2.1	75.0	5.0
8X (88 kS/s)	79.5	2.8	86.5	5.5
9X (99 kS/s)	89.0	3.0	97.0	5.8
10X (110 kS/s)	100.0	3.2	107.0	6.0
12X (132 kS/s)	118.0	3.1	124.0	6.0
14X (154 kS/s)	100.0	0.5	125.0	2.0
16X (151 kS/s)	80.0	3.0	50.0	5.0
18X (165 kS/s)	68.0	0.1	1.6	0.1
20X (165 kS/s)	68.0	0.1	1.0	0.1

Table 5.2: Aggregator statistics for a single sensor pushing Samples into an Aggregator at various rates. Replay rate is calculated based on 10-second averaged service rate, bandwidth (bps) and packets per second (Pps) recorded manually using the “jnettop” tool.

Table 5.2 details the approximate bandwidth consumed by the Aggregator, in bits per second (bps) and packets per second (Pps), for a range of replay speeds. In addition, the “Sensor Rate” column includes the median Aggregator service rate, in Samples per second, for each replay speed.

The Replay Sensor makes use of the Java-based Owl Platform library to handle the network protocol details. One the network settings used in the library disables Nagle’s TCP algorithm, which would normally cause IP datagrams to be delayed for a short period so that larger payloads could be sent within each datagram. Instead, the library network code disables this setting causing datagrams to be sent as soon as an application layer message has been sent to the trasport layer buffers. The goal is to trade an increase in network bandwidth for a decrease in potential network delays. The effect of this setting becomes most apparent in Table 5.2 where as the Sensor Rate increases from 12X to 16X, the overall bandwidth and number of packets transmitted decreases.

Although it is not possible from this data to determine the exact cause of the decrease, the most likely explanation is that as the sending rate increases, the TCP/IP implementation begins to pack multiple Sample messages into the same TCP segment and IP datagram. If this were the case, we would expect to see an overall decrease in bandwidth as the sending rate increases because the amount of per-Sample overhead will decrease. By the time the sending rate has reached its maximum (18–20X based on Figure 5.4), both the bandwidth and the packet rate have also reached a stable rate. It appears that the maximum rate a single Replay Sensor on an ORBIT node can transmit is approximately 170,000 Samples per second, or 1 every 6 microseconds.

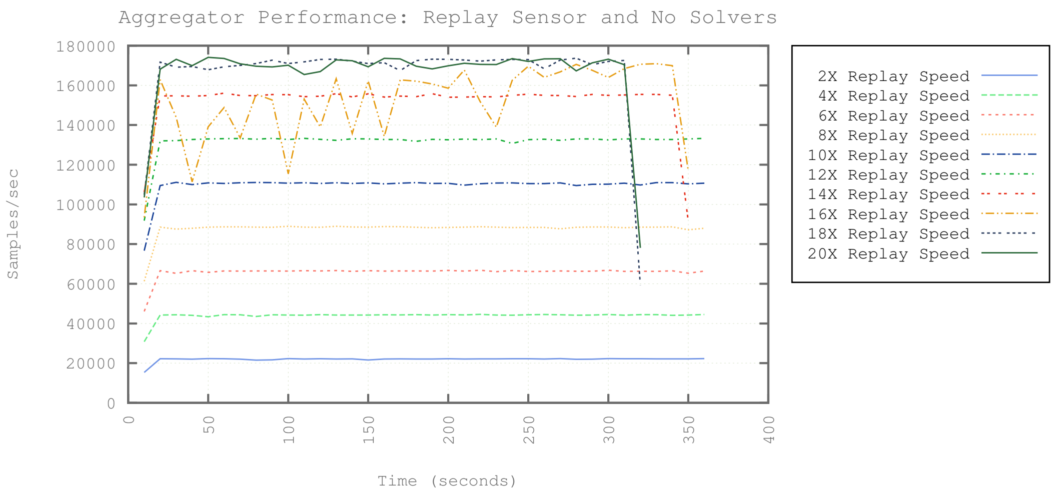


Figure 5.4: Plot of Aggregator service rate over time for a single Sensor sending Samples. Sensor replay speed is a multiple of the recorded rate.

Figure 5.4 details the average service rate for the Aggregator over the duration of the experiment. In general, the service rate increases linearly with the replay speed with two notable exceptions: 18X and 20X show nearly the same performance, and 16X shows a very unstable rate throughout the experiment. The nearly equal performance at 18X and 20X is an indication that the Replay Sensor has reached the limit at which it can send Samples to the Aggregator. Based on the earlier profiling of the network links between the two hosts used for this experiment, it is reasonable that this limitation is a part of the Replay Sensor software and not imposed by the infrastructure used. The Replay Sensor is a single-threaded process that reads Samples from a file and transmits them across the network. Although the file is read-ahead buffered in memory, the time to decode/deserialize the Samples and send them may be a bottleneck. An updated software design that divorces the two activities into separate threads may achieve an increase in performance. However, this limit is not as significant as it may initially appear. Based on the results in Section 5.8, it becomes clear that the Solvers further restrict Aggregator performance. The highly variable performance demonstrated at 16X is more difficult to explain and the cause is not evident, though appears to be an exaggerated version of similar variances seen in the other traces and additional investigation is left as future work. During the last 90 seconds of the experiment, the jitter does appear to reduce in magnitude. With reduced jitter, the stable sending rate looks to be very close to the 18X and 20X rates, which are likely the upper bound on the Replay Sensor's capabilities.

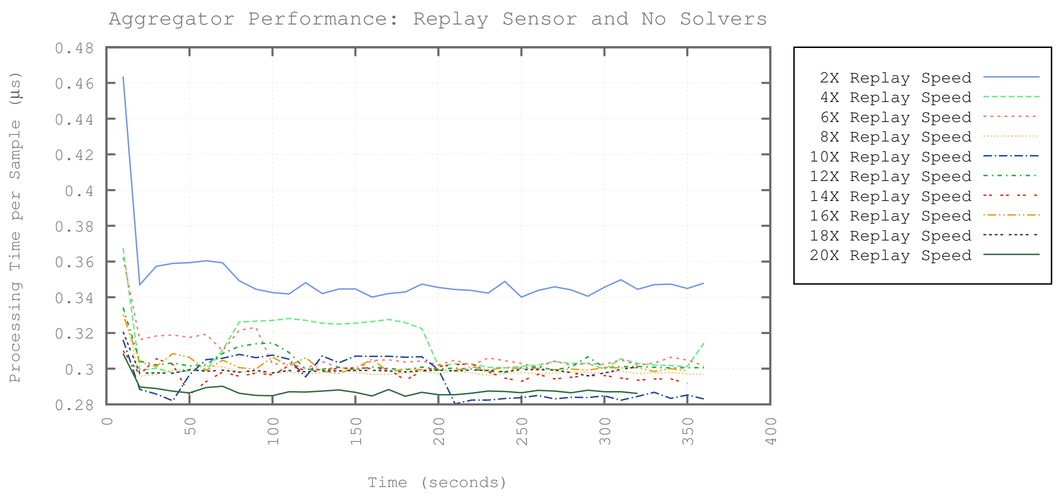


Figure 5.5: Aggregator Sample processing time over the experiment duration.

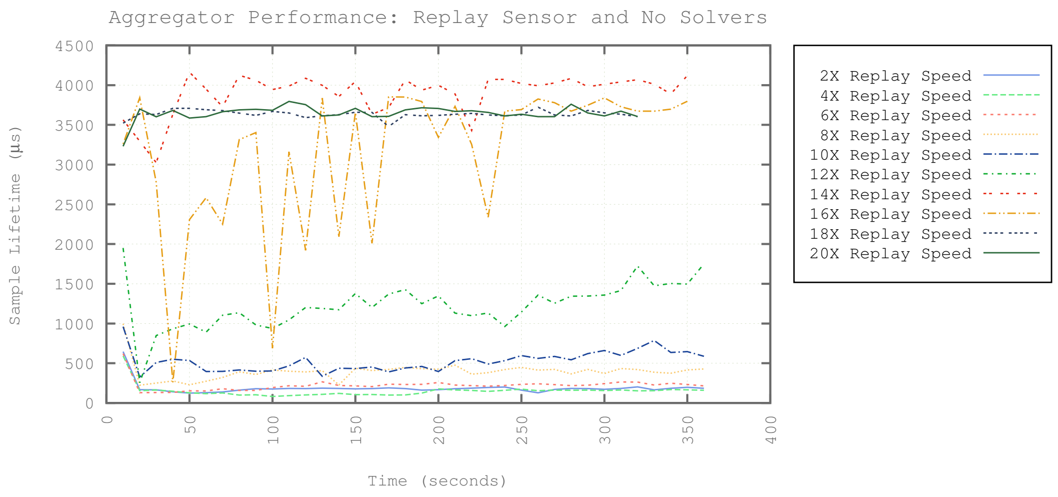


Figure 5.6: Aggregator Sample lifetime over the experiment duration.

Figure 5.5 shows the average time required to process each Sample by one of several worker threads within the Aggregator. Of note is the trend toward lower processing times (faster processing) as the rate of Samples arriving at the Aggregator (service rate) increases. This effect is likely caused by a number of factors including less frequent context switching as the Aggregator demands more CPU time over other processes in the OS and consequently a higher hit rate in the CPU caches. For the 16X replay speed, the processing time differs by approximately 10 nanoseconds, or around 3% even while the Sample arrival rate fluctuates by as much as 30% in the same experiment. Figure 5.6 shows the average "lifetime" for each Sample that is processed by the Aggregator. The most striking feature of this figure is that the 16X replay speed demonstrates variability similar to Figure 5.4, and in fact the lifetime values track closely with the arrival rates. This would be expected given the relatively stable processing time demonstrated by the Aggregator.

5.6 Multiple Data Sources

To further test the limits of the Aggregator under high input loads, a second set of experiments used multiple Replay Sensors to send Samples to the Aggregator at varying speeds. These experiments would enable profiling the Aggregator at higher work loads than possible with a single Replay Sensor is capable of generating, and also to see what impact multiple network connections has on the Aggregator. For each experiment, the Aggregator was started on an ORBIT node, and then the Replay Sensors were started one at a time, on separate ORBIT nodes. The result is a gradual increase in the number of Samples the Aggregator needs to process, which should minimize transient effects caused by a sudden increase in traffic.

No. Sensors	Sensor Rate	Agg. Rx (Mbps)	Agg. Tx (Mbps)	Agg. Rx (kPps)	Agg. Tx (kPps)
3	1X	30.0	1.5	32.0	3.3
3	5X	148.0	7.0	173.0	17.0
3	10X	285.0	8.0	300.0	16.0
3	15X	160.0	0.2	4.0	0.3
6	1X	60.0	3.6	65.0	6.8
6	5X	288.0	13.5	316.0	25.0
6	10X	211.0	0.2	5.5	0.4
6	15X	210.0	0.2	6.0	0.4
9	1X	90.0	6.5	95.0	12.0
9	5X	195.0	0.2	4.5	0.4
9	10X	200.0	0.2	3.5	0.4
9	15X	196.0	0.2	4.5	0.4

Table 5.3: Network statistics for multiple Sensors pushing Samples at varying rates into a single Aggregator.

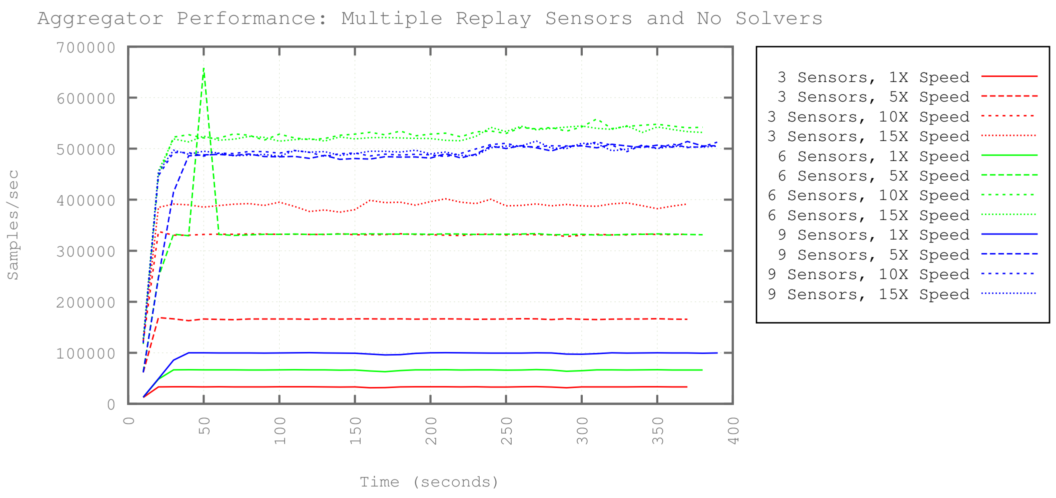


Figure 5.7: Aggregator service rate with multiple Replay Sensors connected and forwarding Samples at various speeds.

Similar to the experiments involving a single Replay Sensor, the network bandwidth and packet traffic exhibited increases until reaching a saturation point and then decreasing as multiple Sample messages are combined in a single IP datagram. Table 5.3 shows that the tipping point in this case appeared to be around 300,000 packets per second (3 Sensors/10X and 6 Sensors/5X), which is in line with the *hping3* measurements described in Section 5.4. Figure 5.7 shows that the Aggregator appears limited to approximately 525,000 Samples per second. This limit occurs at roughly the same point for both the 6-Sensor and 9-Sensor cases, and so does not appear to be strongly impacted by the number of connections. However, the 9-Sensor experiments do have a slightly lower overall service rate. This small but significant correlation between larger number of Sensor network connections and lower Aggregator performance implies a need to avoid excessive numbers of connections in practice.

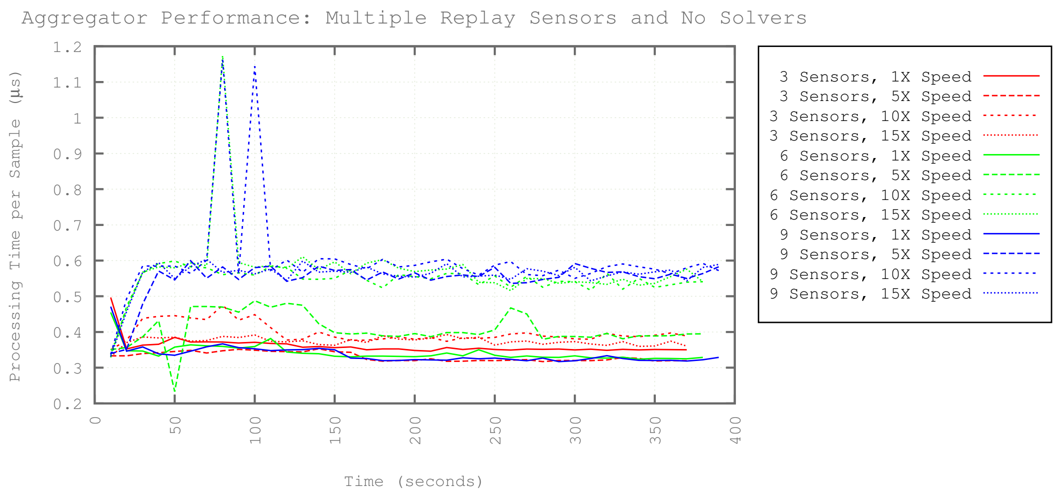


Figure 5.8: Aggregator per-Sample processing time with multiple Replay Sensors connected.

With multiple Sensor connections to the Aggregator, the per-Sample processing time is still relatively stable. In Figure 5.8, it is clear that for many workloads, the performance is very close to the single-connection experiment (Figure 5.5), averaging between 300 and 400 nanoseconds per Sample. For several of the higher workloads, the average processing time increases to upwards of 600 nanoseconds per Sample. Again, this increased processing time may be attributable to the increased workload caused by multiple connections to the Aggregator.

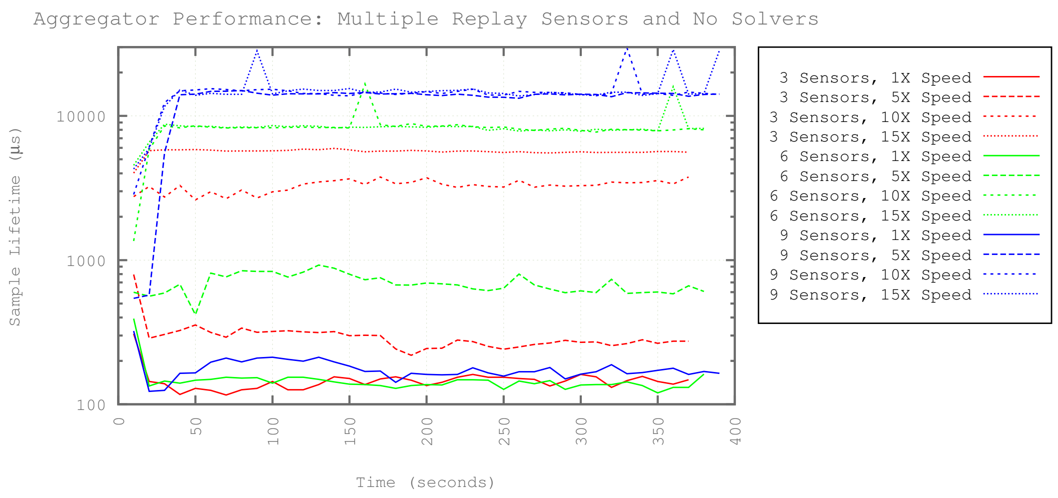


Figure 5.9: Aggregator Sample lifetime with multiple Replay Sensors connected.

Finally, looking at the Sample lifetime graph in Figure 5.9, it becomes clear that although all workloads achieve a stable state within the Aggregator the higher workloads produce considerably longer queueing times. The difference between the 1X and 5X cases for 9 Sensor connections is nearly 2 orders of magnitude, jumping from 200 microseconds to around 15 milliseconds. A delay of even several milliseconds for raw sensor may be sufficiently high as to cause problems in many applications (e.g., alarms or feedback controllers), and at 15 milliseconds many humans will notice a delay. For instance, many video displays for video games or virtual reality systems are run at 60 Hz (16.6 milliseconds per frame) because many users feel motion is more fluid and realistic than at lower framerates. It is important, therefore, to keep these limits in mind when architecting a system that relies on timely information being delivered to various components.

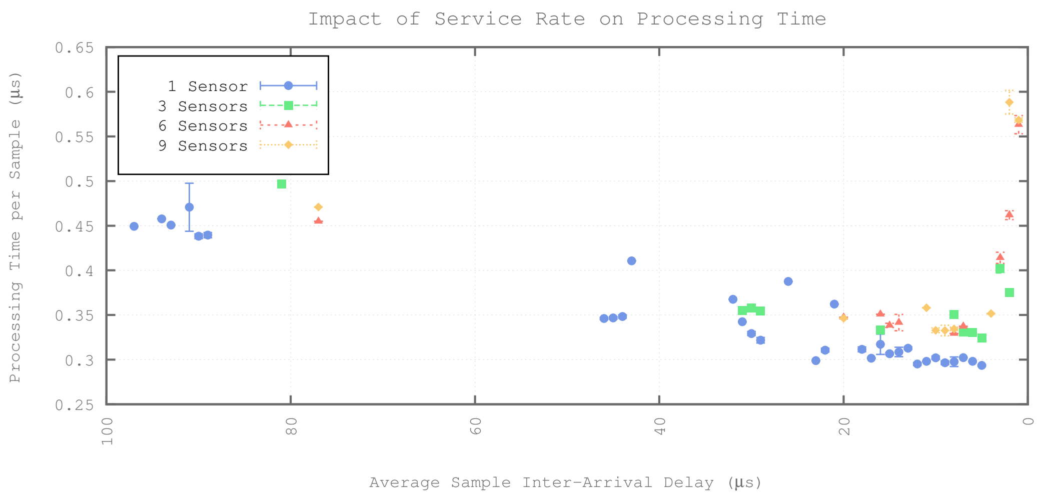


Figure 5.10: Impact of service rate on Aggregator Sample processing time for 1, 3, 6, and 9 sensor connections and no solvers. The sensors replayed trace data at varying rates.

Figures 5.10 and 5.11 plot the relationship between the Sample inter-arrival delay and the processing time and Sample lifetime, respectively. More clearly than in the previous figures, it is possible to see that as the inter-arrival delay decreases, the processing time decreases and Sample lifetime (queueing delay) increases. As expected in a queueing system like this, both processing time and Sample lifetime begin to grow exponentially as the input rate approaches the limit of the system. For example, to maintain sub-millisecond delays, it is necessary to limit the inter-arrival delay to no fewer than 15 microseconds. This is equivalent to approximately 65,000 Samples per second across all connections.

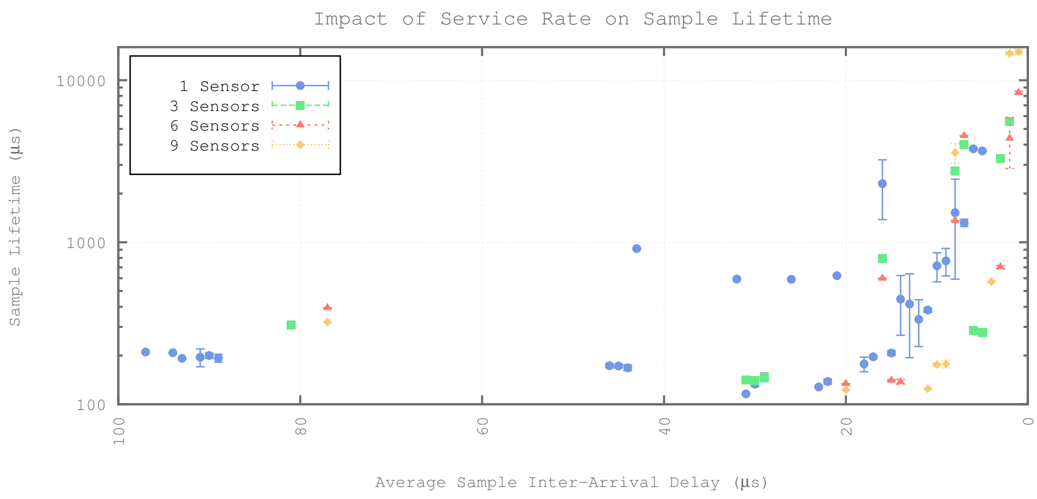


Figure 5.11: Impact of service rate on Aggregator Sample lifetime for 1, 3, 6, and 9 sensor connections and no solvers. The sensors replayed trace data at varying rates.

5.7 Impact of Solver Connections

The next set of experiments evaluated how the Aggregator performance was impacted by the number of Solver connections. For each experiment, 5 Replay Sensors were connected to the Aggregator, and different numbers of Solvers were connected and configured to request all Samples from the Aggregator. The rate at which the Replay Sensors sent Samples to the Aggregator was varied to provide more detailed information about how the performance was affected at different service loads. As in the previous experiments, the system was allowed to run for a period of 360 seconds while statistics were gathered from the Aggregator.

No. Solvers	**Sensor Rate**	**Agg. Rx (Mbps)**	**Agg. Tx (Mbps)**	**Agg. Rx (kPps)**	**Agg. Tx (kPps)**
1	1X	50.0	25.0	55.0	8.0
1	5X	240.0	44.0	265.0	22.0
1	10X	181.0	26.0	7.7	3.8
1	15X	178.0	25.0	7.8	3.7
3	1X	51.0	59.0	57.5	12.9
3	5X	230.0	55.0	230.0	16.0
3	10X	128.0	60.0	6.4	6.4
3	15X	135.0	60.0	11.0	8.5
6	1X	53.0	118.0	60.0	17.0
6	5X	100.0	83.0	14.0	13.0
6	10X	99.0	81.0	13.7	12.6
6	15X	99.0	83.0	13.0	12.0
9	1X	55.0	132.0	63.0	20.0
9	5X	81.0	97.0	16.0	15.5
9	10X	78.0	94.0	15.4	14.7
9	15X	82.0	92.0	15.5	14.5

Table 5.4: Aggregator statistics for 5 sensors pushing Samples at varying rates, with different numbers of Solvers connected. Each Solver is requesting "all" Samples to be forwarded.

Table 5.4 shows the approximate network bandwidth used by the Aggregator over the 16 experiments conducted. As before, Nagle's algorithm is disabled in the Replay Sensors, resulting in higher bandwidth and packet rates for lower playback speeds. Unlike previous experiments, however, each of the connected Solvers serves to multiply the number of Samples transmitted over the network. For each Sample received from the Replay Sensors, the Aggregator needs to transmit a Sample to each Solver. This results in a ninefold increase in Sample rate between the receiving and sending components of the Aggregator. However, because Nagle's algorithm is enabled between the Aggregator and Solvers, the number of transmitted packets (and the overall bandwidth consumed) is markedly lower than the Sensor-Aggregator connections.

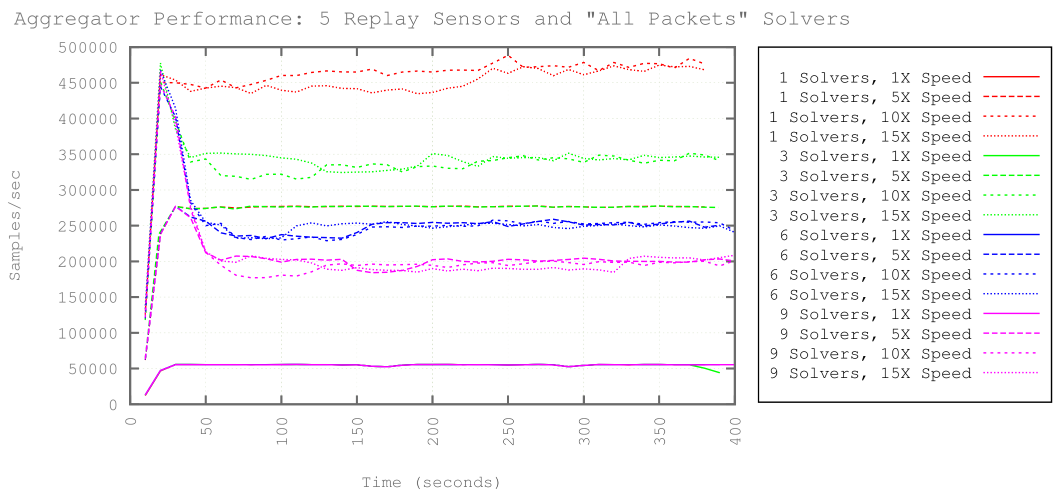


Figure 5.12: Aggregator service rate with multiple Replay Sensors and Solvers connected. Solvers requested all Samples from the Aggregator.

Figure 5.12 shows that, like the Sensor-only experiments in Section 5.6, the Aggregator appears to have a well-defined service rate limitation. With a single Solver connection, the limit appears to be around 475,000 Samples per second, approximately 75,000 Samples per second fewer than without Solver connections. As additional Solver connections are established, this limit decreases rapidly until with 9 Solver connections, the maximum stable rate achieved is close to 200,000 Samples per second. For 1- and 3-Solver connection experiments, equivalent performance was achieved at the 1X and 5X rates (55,000 and 275,000 Samples/per second), but beyond this performance begins to suffer noticably for the 3-Solver experiments. With 6 and 9 Solver connections, performance is even more significantly impacted for anything above the 1X rate.

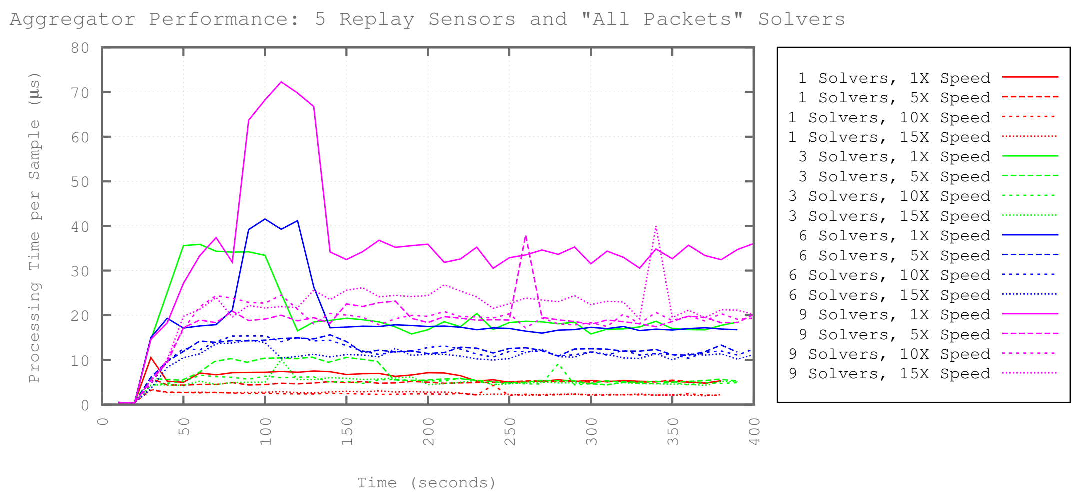


Figure 5.13: Aggregator per-Sample processing time with multiple Replay Sensors and Solvers. Each Solver requested all Samples.

Per-Sample processing time is significantly higher than for the experiments with no Solver connections. Figure 5.13 shows that for the 1-Solver experiments the processing time is an order of magnitude higher, up from 400 nanoseconds to nearly 8 microseconds. An interesting trend in the same figure is that for each of the experiments, the 1X rate has a higher processing time than the other, faster, rates. In every case except the 9-Solver experiments, the per-Sample processing time decreased as the rate increased. For the 9-Solver experiments, this trend was observed except at the 15X rate. While the exact cause of this apparent efficiency gain is not readily observable, it is likely to be same effect seen in Figure 5.5.

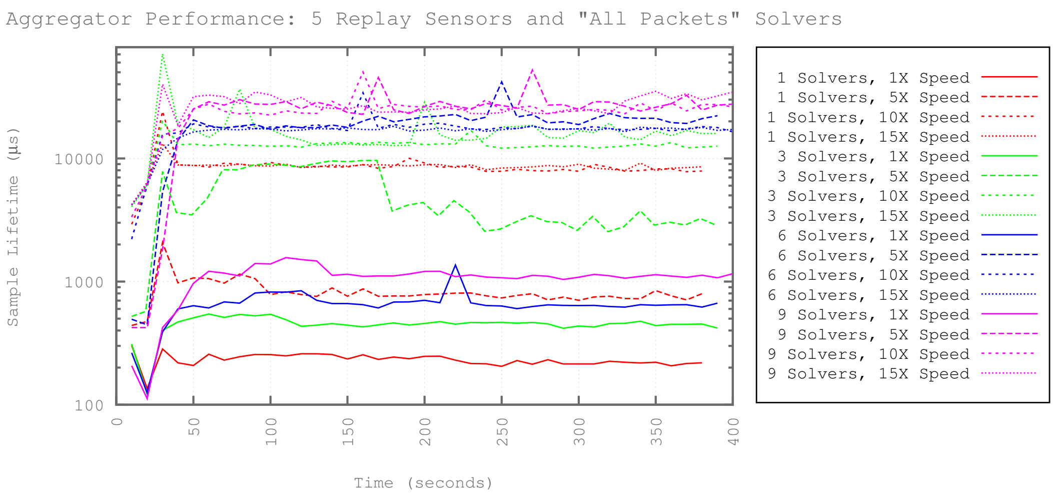


Figure 5.14: Aggregator Sample lifetime with multiple Replay Sensors and Solvers. Each Solver requested all Samples.

The Sample lifetimes for these experiments (Figure 5.14) are similar to those of the no-Solver experiments, with a near doubling of the delay in the highest-rate experiments. In cases where significant delays are intolerable to the applications desired, it appears necessary to limit either the number of Solver connections, or the Sample rate, or both. In situations where a large amount of data is generated by a wireless sensor network, but where a large number of Solvers are also needed to analyze the raw data, it is possible to build a hierarchy of Aggregators with Solvers acting as both relays and filters between them. For instance, instead of 10 Solvers subscribing to a single Aggregator, the primary Aggregator (to which all WSN data is sent) has a single Solver which connects to it and applies in-line filtering to reduce the data stream based on application requirements. This Solver can in turn forward the data to 2 other Aggregators, which accept 5 Solver connections each. This same technique can also be used to combine the WSN data streams from multiple Aggregators into a single Aggregator for processing.

Figure 5.14 shows a number of “spikes” in Sample lifetime at 160, 220, and 250 seconds for different traces. Very similar increases in Sample lifetime at approximately the same time within the traces for the 6- and 9-Solver experiments. Figure 5.12 shows that these traces have approximately the same Sample processing rate, and therefore similar object allocation rates in the JVM heap. It stands to reason that the garbage collector would need to execute at approximately the same time during these experiments, leading to an increase in processing times and Sample lifetimes. Similar results are observed in Figures 5.8 and 5.9, also likely caused by garbage collection.

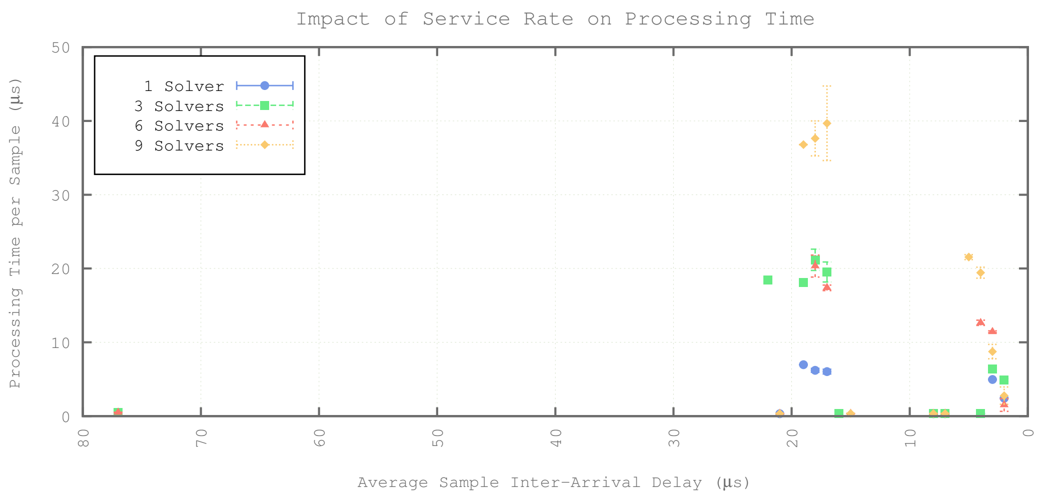


Figure 5.15: Impact of service rate on Aggregator Sample processing time for 5 sensor connections and 1,3,6, and 9 solvers. The sensors replayed trace data at varying rates and the Solvers requested all Samples.

Figures 5.15 and 5.16 plot the per-Sample processing time and average Sample lifetimes for various Sample inter-arrival delay values. As in the previous section, limiting the inter-arrival delay to around 15 microseconds will keep the delay to under a millisecond, except in the 9-Solver cases. With a large number of connected Solvers, limiting to 20 microseconds between Samples (50,000 Samples per second) should ensure Sample delays are consistently below 1 millisecond.

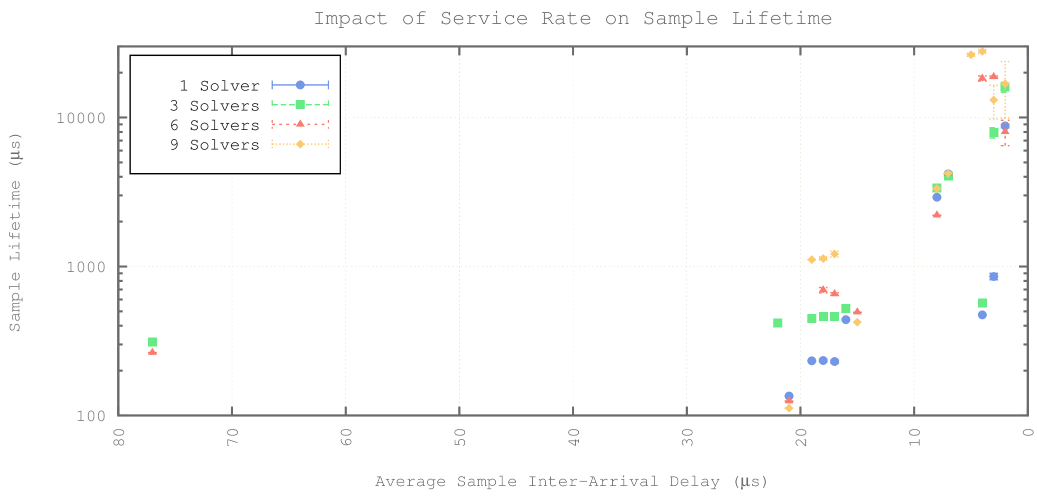


Figure 5.16: Impact of service rate on Aggregator Sample lifetime for 5 sensor connections and 1,3,6, and 9 solvers. The sensors replayed trace data at varying rates and the Solvers requested all Samples.

5.8 Filtering Transmitters

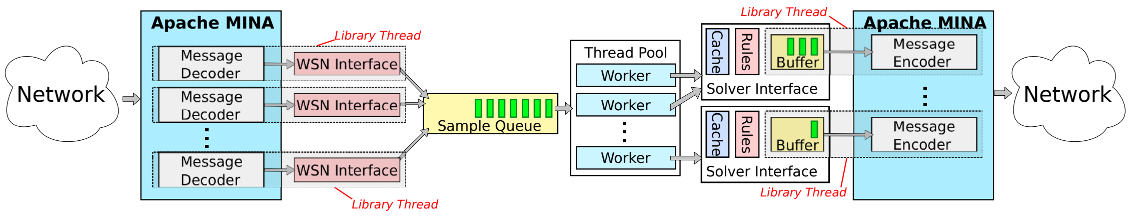


Figure 5.17: Component diagram of the Aggregator showing the flow of data (Samples) through each of the internal components (arrows).

The last set of experiments to test the Aggregator's performance configured the system to start the Aggregator and 5 Replay Sensors on separate hosts, and then start a varied number of Solvers and configure them to request 5 specific Transmitter ID values that are known to be in the trace files. Although the experiments in Section 5.7 examined the performance of Solver connections, those Solvers requested all Samples and so the rules to be evaluated were extremely simple. In contrast, by requesting a specific set of Transmitters, the Aggregator worker threads will be forced to perform relatively complex bitmasking on each Sample that passes through the Aggregator. Figure 5.17 has a detailed view of how the Aggregator is structured internally. These experiments evaluated the bottleneck imposed on the Aggregator by the "Solver Interface" component, specifically its "Rules" and "Cache" data structures.

For each Sample that is about to be sent to a Solver, the Worker thread must first access the Solver Interface's cache to determine if a previous evaluation of the Subscription Request has been performed and stored. The cache is a fixed-size LRU cache implemented using Java's Linked-HashMap class, and modified to eject the eldest entry when the map's size exceeds a threshold. This allows a reduction in the amount of per-Sample processing required for each Solver dependent on the size of the cache. For these experiments the default value in the Aggregator's code base, 200 entries, was used. Although this is only 35% of the total number of Sensors in the experiments, it seemed an acceptable choice since it was currently in use at multiple installations of the Owl Platform at the time.

If there is no entry in the Solver Interface's cache for a Sample's Transmitter ID value, either because it was ejected or never inserted, each Rule in the Subscription Request is evaluated until one is found that permits sending the Sample. If none is found, then the Sample is discarded for that Solver. Evaluating a Rule involves two steps performed in series. The first is to apply the Transmitter ID mask to the Sample's Transmitter ID value and verify that it matches the Transmitter ID base value in the Rule. Because Java does not allow direct access to vector instructions in the host CPU, the 16-byte Transmitter ID mask is applied 1 byte at a time. In a 32-bit system, a 4-byte application would be possible, but the internal representation of a Transmitter ID is a byte array of length 16. The resulting loss of performance is not measured in these experiments and is left as future work. Once the Sample's Transmitter ID value has matched one of the Rules, the second step is performed. Based on the Update Interval provided in the Subscription Rule, the Aggregator determines if the elapsed time between the previous Sample from the same Transmitter-Receiver pair and the current time is greater than or equal to the Update Interval. If it is, then the Sample is sent to the Solver, else it is discarded. When the Sample has been sent to the Solver, the timestamp is recorded and used for comparison with future Samples.

No. Solvers	**Sensor Rate**	**Agg. Rx (Mbps)**	**Agg. Tx (Mbps)**	**Agg. Rx (kPps)**	**Agg. Tx (kPps)**
1	1.0X	49.0	2.5	54.0	5.0
1	2.0X	100.0	6.3	108.0	11.4
1	3.0X	149.0	9.0	162.0	18.0
1	4.0X	198.0	11.2	212.0	20.2
3	1.0X	50.0	3.5	55.0	5.5
3	2.0X	101.0	7.0	107.0	11.4
3	2.5X	126.0	8.5	136.0	13.2
3	3.0X	—	—	—	—
6	1.0X	50.0	4.5	55.0	7.0
6	1.5X	76.0	7.1	83.0	10.5
6	2.0X	—	—	—	—
9	1.0X	51.0	5.6	56.0	7.5
9	1.25X	64.0	6.8	70.0	8.8
9	1.5X	—	—	—	—

Table 5.5: Aggregator statistics for 5 Sensors pushing Samples at varying rates, with different numbers of Solvers connected. Each Solver is subscribed to 5 Sensors using 5 Subscription Rules. The marking "—" indicates that the Aggregator could not maintain service and eventually crashed.

Table 5.5 details the bandwidth used by the Aggregator for each of the experiments performed. In contrast to the previous experiments, where the Solvers requested all Samples from the Aggregator, the bandwidth transmitted by the Aggregator is significantly reduced because only 5 Transmitter IDs (fewer than 1% of all Transmitters) are requested by the Solvers. However, the significantly increased workload borne by the Aggregator resulted in much lower Sample rates compared to the previous experiments. For each of the experimental configurations, as the sending rate from Replay Sensors was increased, the Aggregator eventually crashed. The rate that caused a crash is indicated by the "—" marks within the table.

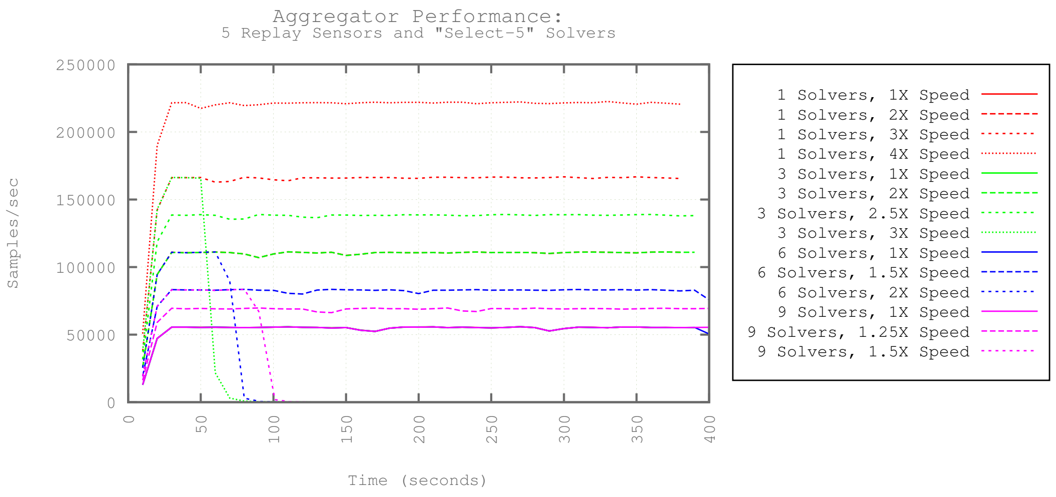


Figure 5.18: Aggregator service rate with multiple Replay Sensors and Solvers connected. Solvers requested 5 specific Transmitter ID values from the Aggregator.

Figure 5.18 shows the maximum service rate sustained by the Aggregator for different numbers of Solvers and Sensor sending rates. When the Aggregator was unable to maintain stable service, the service rate rapidly dropped to a few hundred Samples per second. Unlike predictions made by queueing theory models, where the service rate would be expected to decrease exponentially as it approached 100% capacity, the Aggregator rapidly went from stable performance to complete failure. The overall service rate is also significantly lower when Transmitter ID values need to be evaluated for each Sample. In Figure 5.12, the highest throughput of the Aggregator was approximately 475,000 Samples per second whereas in these experiments the highest rate is only 225,000 Samples per second — more than a 50% decrease. As the number of Solvers grows, the service rate is limited even further. With 9 Solvers connected, the highest sustainable throughput was limited to around 70,000 Samples per second.

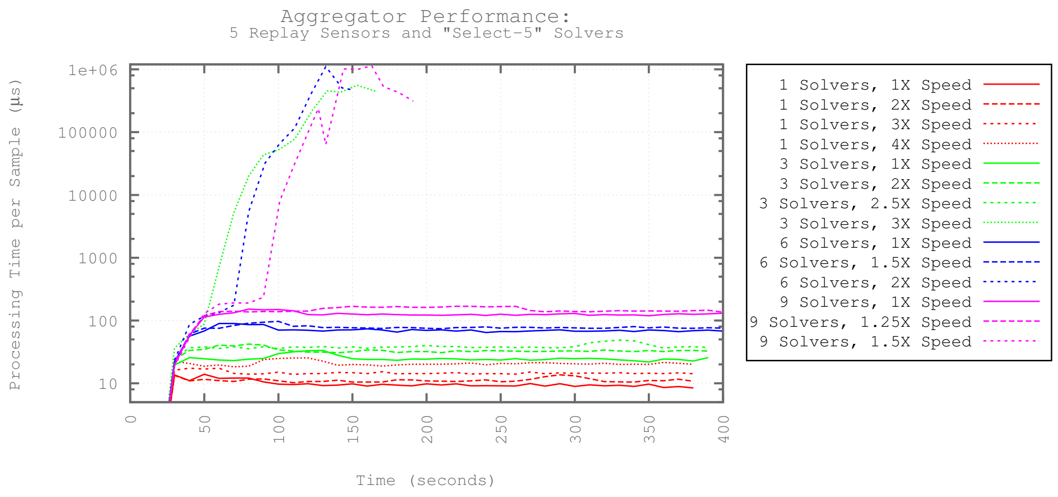


Figure 5.19: Aggregator per-Sample processing time with multiple Replay Sensors and Solvers. Each Solver requested 5 specific Transmitter ID values.

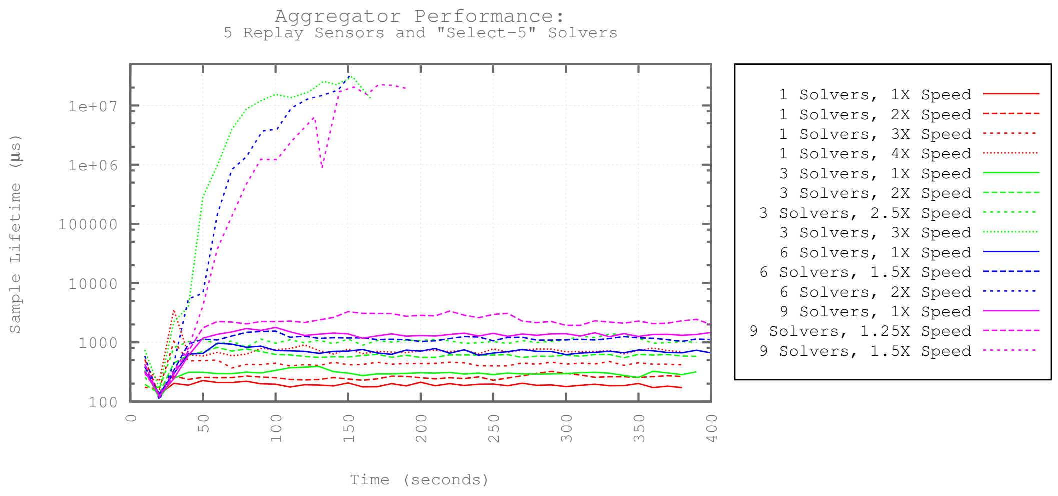


Figure 5.20: Aggregator Sample lifetime with multiple Replay Sesnors and Solvers. Each Solver requested 5 specific Transmitter ID values.

Looking at the per-Sample processing time and the Sample lifetime (queueing delay) values in Figures 5.19 and 5.20, respectively, it is clear that the amount of work required for each Sample has increased appreciably. For a single Solver with the Sensors sending at 1X (11,000 Samples per second), the processing time has doubled from around 5 microseconds to nearly 10. At higher rates, the processing time climbs to 25 microseconds. With 9 Solvers connected, each Sample takes over 100 microseconds to process. The Sample lifetime values remained mostly the same, when compared to the previous experiments at the same rates. Once again, all but the 9-Solver experiments remained below 1 millisecond at 1X indicating that the 20 microsecond inter-arrival delay is still a good choice.

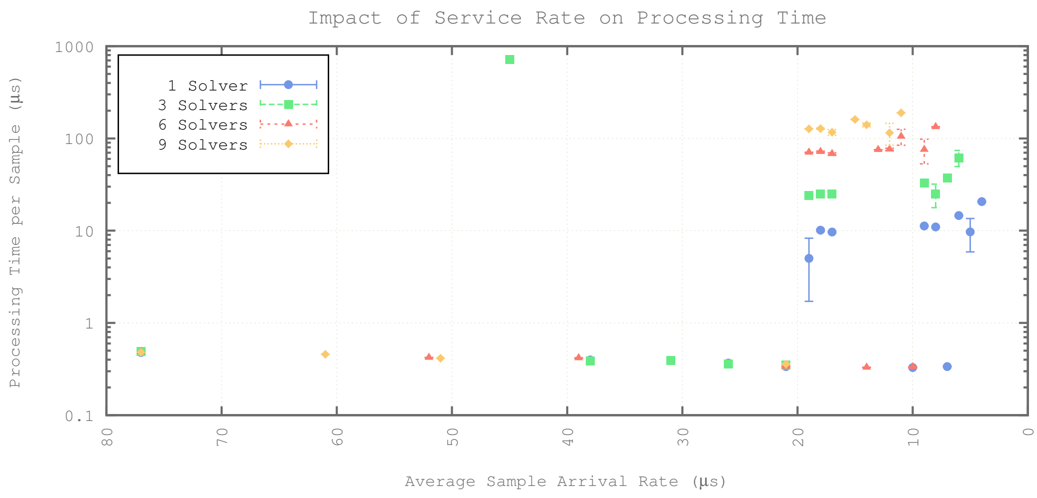


Figure 5.21: Impact of service rate on Aggregator Sample processing time for 5 sensor connections and 1,3,6, and 9 solvers. The sensors replayed trace data at varying rates and the Solvers request 5 specific Transmitters.

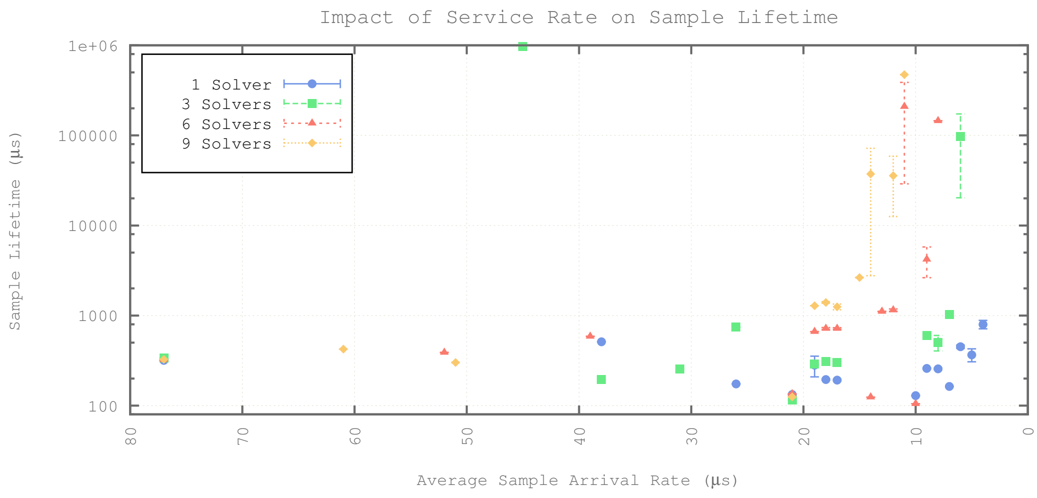


Figure 5.22: Impact of service rate on Aggregator Sample lifetime for 5 sensor connections and 1,3,6, and 9 solvers. The sensors replayed trace data at varying rates and the Solvers request 5 specific Transmitters.

Figures 5.21 and 5.22 provide additional details about the impact on inter-arrival delays on processing time and queueing delay. Figure 5.21 provides a particularly clear picture of the effect that the number of Solver connections has on the performance of the Aggregator.

5.9 Improving Performance

Overall, the relatively low rates at which the Aggregator failed to perform sufficiently were particularly troubling, and an attempt was made to determine the cause or causes contributing to the failures. A particularly troublesome observation was that the total CPU usage of the Aggregator would drop from 600–700%, indicating the use of 6–7 cores, to around 100%. Two likely explanations could account for this drop in CPU utilization by the Aggregator. Either the Aggregator had too many active threads being swapped in and out too frequently (CPU churn) such that no one thread operated efficiently, or the only active thread was the garbage collector, which was running nearly continuously trying to free memory in the JVM heap.

Hoping to gain at least a cursory understanding of what specifically limited the performance of the Aggregator, the code was adjusted in several ways, and the impact on the performance was tested using the 9-Solver experiments and comparing the results at different rates to what is provided in this section. The testing indicated that a number of different components each contributed to the Aggregator failures. Specifically,

- increasing the cache size to 2,000 entries reduced processing times,
- modifying several synchronized data structures to reduce or eliminate contention further improved performance, and
- reducing the overall number of threads allocated by the Aggregator improved performance.

While the first two items are relatively obvious and straightforward improvements that can be made to nearly any similar piece of software, reducing the number of threads provides a less intuitive improvement in performance. The computer on which the Aggrgator ran has an 8-core CPU capable of executing 8 different sets of instructions simultaneously. The Aggregator, however, originally had 30 separate threads running simultaneously: 16 worker threads, 5 threads for the Sensor connections, and 9 threads of the Solver connections. In general, the Apache MINA library used for the network input/output allocates a new thread for each connection to the Aggregator. This works well for connections that have relatively low activity levels, such as HTTP or SSh connections, but for these experiments the number of messages transmitted was very high and the threads were frequently active. The worker pool for the Aggregator is configured manually at compile time, and was originally set to twice the number of available processors in the system. This worked well for lower service rates, but with the addition of rule checking each thread became active for much longer periods of time. The result of all of these threads running with such high workloads is a lot of CPU churn, where the thread scheduler needs to rapidly switch among threads to give each a fair share of the CPU resources available.

In order to validate this last theory, two changes were made to the Aggregator: the number of worker threads was varied and the Solver Interface was further instrumented to record the time required to perform small critical sections of code. The Aggregator would log any Sample that took longer than a 50 microseconds to fully process a Sample. The number of threads was then decreased from twice the CPU count to exactly the CPU count, and then to half the CPU count. Decreasing the number of worker threads generally increased the service rate that the Aggregator could sustain before crashing. When the Aggregator did begin to crash, the time required to execute different sections of the code would randomly increase by large amounts. For instance, a simple conditional check might normally take 600 nanoseconds, but under very high loads it would fluctuate between 600 and 800,000 nanoseconds. The same large jumps would occur randomly in other parts of the code. This is a strong indicator that the worker thread was swapped-out for another thread for the 800 microseconds and then swapped back in.

One of the good and bad parts about the Aggregator being written in Java is that memory management is handled automatically by the Java Virtual Machine (JVM). So long as no strong references remain to an object, it will be removed from the heap by the garbage collector. In general, this makes programming applications easier than in languages like C/C++ which do not provide garbage collection. Unfortunately, it also means that high-performance applications like the Aggregator are particularly susceptible to garbage collection failures. One of the most common error messages when the Aggregator crashed was that there was no longer enough free heap space for the garbage collector to function. The use of an Object pool, like that provided by the Apache Commons Project, may serve to reduce the number of objects needing garbage collection during each pass [14]. This may not be sufficient, however, when the incoming Sample rate begins to exceed the sustainable service rate in the Aggregator. In this case there are two major options, to provide additional resources to the Aggregator or to reduce the incoming Sample rate. The former solution may be appropriate in some situations where the Aggregator was previously running on older or slower hardware to save costs, and the WSN size has grown to the point where it can no longer maintain stable functionality. In general, though, this would not scale beyond a few orders of magnitude due to the physical limits of electronic circuit design.

A more generalized solution would be to communicate with the WSN gateways about the Aggregator performance, and request that the WSN send fewer Samples to the Aggregator during periods of high load. Alternatively, the Aggregator itself could perform load shedding by dropping Samples according to some set of rules (random, time interval, etc.) and notifying the the WSN about its action. Whether the WSN makes decisions about reducing Sample rates or the Aggregator does is a matter of policy. The Application and Solvers utilizing the Samples sent from the Aggregator may have different priorities concerning Samples than the WSN. As a specific example, the WSN may prioritize data from a large number of independent Transmitting sensors, and drop Samples that contain the same transmitted data, but recorded by different Receivers. However, if the desired application is something that relies on multiple RSSI values from the WSN, then the best policy may be to perform round-robin like selection of Transmitters such that a single transmission recorded by multiple Receivers is transmitted during some time period. However, this type of communication between the Application or Analysis layers and the Sensing layer violates one of the fundamental goals of the system, avoiding the need to know "how" other layers perform their tasks. Evaluation of these alternative approaches, and the exploration of others, is outside the scope of this thesis, and will be considered as future work.

Overall, the changes described above enabled the Aggregator to support 9 Solvers, each requesting the 5 Transmitter ID values, at approximately double the rate it could previously, up to 135,000 Samples per second. Although the experiments were not repeated with these changes applied to the Aggregator due to time constraints, these initial results show promise that further careful improvements to the Aggregator may be able to further improve performance.

Chapter 6
World Model Performance Evaluation

The World Model is the second major component in the Owl Platform architecture. It acts as a central repository for data generated by Solvers and accessed by Solvers and Applications. As data from WSNs and external data sources is pushed into the system, Solvers produce new types of information and push these into the World Model. At the same time, other Solvers are actively awaiting those new Attribute values to generate their own updates. The World Model frequently has to manage multiple sources and destinations for the Attributes flowing through it, while responding to queries for previously-generated data residing in its persistent storage. Rather than require a specific choice of storage engine or set of detailed characteristics, the World Model only has to correctly store, retrieve, and search according to the messages transmitted to it. In the following sections, two proof-of-concept implementations are evaluated and compared: MySQL 5.5 and SQLite 3.7.

MySQL is a "traditional" relational database management system (RDBMS), similar to other popular products like Microsoft SQL, Sybase Server, or PostgreSQL. It is a server process running on a host that handles in-memory and on-disk storage, concurrency guarantees, provides network and local (socket) interfaces, and supports a number of standardized Structured Query Language (SQL) dialects. The World Model server implementation using MySQL accesses the RDBMS over a TCP/IP socket, enabling the World Model server process and the DB process to exist on separate hosts. In the experiments described below, the World Model server process and the MySQL server resided on the same host.

In contrast, SQLite is an embedded relational database (RDB) which resides entirely within the World Model process. SQLite provides many of the same guarantees as MySQL (concurrency, atomicity, isolation, durability), but does not implement as much or as many of the SQL standards available. It stores all of its data within a single file on the host system, which limits how much it can store, and so is most appropriate for relatively small data sets. The World Model server relies on the linked SQLite library to manage reading and writing of the data to and from the hard disk.

A complete and thorough evaluation of the World Model implementations is a significant undertaking, involving a long list of variables including number of Solver connections, number of Client connections, number of Standing Queries per connection, frequency of Attribute updates, frequency of Attribute expirations, frequency of Attribute deletion, regular expression complexity, storage engine configuration, and concurrency of different operations. To attempt to evaluate every one of these values, and their interactions, is beyond the scope of this thesis. Instead, a number of simpler experiments will be used to provide a set of benchmarks to determine at what scale the implementations are capable of performing.

One of the motivating design goals of the Owl Platform is the realization that the amount of raw data potentially produced by a wireless sensor network is orders of magnitude greater than the actual information needed by end-user applications. As an example, imagine a collection of security sensors that monitor the state of the doors and windows in a building, performing a check several times per second to determine if the door or window has been opened and trigger an alarm. For the vast majority of the sensors (doors, windows) over most of the time, the state will never change. If each sensor produces a piece of data 5 times per second, with 20 such sensors in a building, that WSN produces over 8 million pieces of data per day. In contrast, if the doors and windows open once or twice for a few minutes or hours per day, the useful amount of data might be only a few hundred updates indicating the start and end of each door or window state (open/closed). Although an extreme example, it illustrates the point that the raw *data* can be overwhelmingly more voluminous than the amount of data necessary to persist the same *information*. In other words, if the user only cares about the state of the environment to the degree at which the sensors can record it, then the raw sensor data can be viewed as an uncompressed, even inefficient representation of that information. By transforming the raw sensor data into a more compact form, the user does not lose any valuable knowledge about what was recorded, and can store it much more efficiently.

The World Model will not handle data on the scale of the Aggregator, and so need not perform at the same scale. Instead of completing operations at the nanosecond or microsecond scale, the World Model can complete tasks within a few milliseconds and still provide acceptable performance to applications and users interacting with it. To determine whether the current implementations of the World Model achieve this basic level of service, a limited but diverse set of experiments will be used to evaluate its performance. These experiments will serve to illuminate areas needing improvement and provide a point of reference for future implementations or updates.

For each of the experiments, the size of the data set used (or eventually created) will be varied to determine how much server performance depends on data set size. Testing the Solver interface of the World Model will include determining how rapidly a set of Attribute values can be inserted (updated), expired, and deleted for each World Model server implementation. The Client (Application) interface will use fully-populated World Model data set and test how long it takes to perform Snapshot queries, Range queries, and ID searches using regular expressions. For all experiments, the same Origin value will be used throughout for simplicity. In practice, operations that modify Attribute values (Update, Delete, Expire) only apply to the same Origin value, a restriction that is meant to permit multiple sources for the same data and to prevent accidental corruption or destruction of data sourced from multiple Origins.

6.1 Summary of Results

Overall, the SQLite implementation of the World Model outperformed the MySQL implementation by several orders of magnitude for most operations. However, for regular expression Identifier searches and Range Queries, the performance was each was nearly identical. Table 6.1 contains a sampling of results. The SQLite implementation has been heavily used for over 3 years in both research and commercial settings. The MySQL implementation suffers from under-optimized queries, database procedures (functions) which implement significant amounts of application logic, and a failure to exploit the multi-threaded capabilities of the MySQL server. A detailed analysis of the performance of both World Model implementations is provided in the remainder of this chapter.

	MySQL 10k	**SQLite 10k**	**MySQL 100k**	**SQLite 100k**
Update	17,165	284	17,206	294
Expiration	5,192	162	5,576	168
Deletion	5,266	81	5,587	87
RegEx Search 3	26,596	23,486	241,052	239,823
Range 5% Query	8,883	4,415	97,899	47,947
Snapshot 22.5% Query	15,952	284	177,330	294

Table 6.1: Summary of World Model implementation results. Cell values are 95 percentile time measured by the World Model in microseconds.

6.2 Experimental Design

Experiments were performed on the ORBIT testbed facility at WINLAB. The same nodes described in Chapter 5 were used in order to maintain a comparable testing environment. For each experiment, up to 3 nodes were used and hosted either a World Model server, a “Profiler” program, or a “receiver” program. The World Model server was either the MySQL or SQLite implementation. The Profiler is a Java program that performs a customizable set of non-interactive tasks and computes the “wall” time taken to perform each [28]. The receiver program is a simple World Model Client that requests all Identifiers and Attributes and records the relative time elapsed between each value being received.

The receiver is used to determine if Standing Queries result in a significant impact in performance on the World Model, and also as a secondary method of measuring World Model performance.

Experiments were executed in sets, where each set employed groups of 3 nodes, using the same parameters for the World Model and Profiler program, except that the number of attributes to be updated was varied. In general, 5 groups of 3 nodes each were used for each set of experiments. In some circumstances, failure of the node management software meant that specific experiments were repeated with fewer than 5 groups. Before each experiment was started, any previous state of the World Model was cleared by either removing the database files from the file system (SQLite) or by issuing delete statements to the database server directly (MySQL). Although it may be theoretically possible for different groups of nodes to interfere with one another this is not predicted to have a noticable impact on the experimental outcomes. Each node has a 1 Gbps network interface and each network switch can process up to 10 Gbps of traffic. The highest observed network utilization on the World Model hosts was below 1 Mbps when a single group of nodes was used.

The Profiler application was able to be configured at run-time for one of several workloads. All workloads began by generating the desired number of Identifier/Attribute values, which would be inserted into the World Model (Attribute Update). The simplest experiment was to insert each Attribute individually in the World Model, await completion, and then exit. All other workloads would perform bulk Attribute updates, where multiple values are included in a single network message, to reduce the total time required. Each Attribute had its Creation Timestamp set to exactly 1 minute from any other, resulting in a regular distribution of Creation Timestamp values between the start of the experiment and n minutes in the past, where n is the number of Attributes generated for the experiment. For all experiments, the Identifier value was unique for each Attribute value, while the Attribute name was identical. This is equivalent to having distinct objects, all with a "temperature" Attribute value available.

After insertion, Attributes may be expired by setting the Expiration Timestamp equal to the next Attribute value's Creation Timestamp. After expiration, the Profiler performed a Range Request for 100% of the data, then 10 requests for 10% of the data, 20 requests for 5%, and 100 requests for 1%. After the Range Requests, a series of 20 Snapshot requests were made at 5% intervals starting at 2.5% of the data set. For instance, if the data set contained a total of 1000 Attribute values, the first request would be at 25 minutes after the Creation Timestamp of the earliest Attribute value, the next at 75 minutes, and so on. After the Snapshot Requests, four regular expression searches were performed on Identifier values. Finally, each Attribute value was deleted individually. For all client interface requests (Range, Snapshot, Search), timing information was recorded by the World

Model and the Profiler. For solver interface requests, only timing at the World Model server is available because the network protocols lack acknowledgements for individual messages.

It also important to note that throughout the following sections, the experiments involving the MySQL and SQLite implementations used overlapping but different ranges of data set sizes. The MySQL implementation data sets ranged from 1,000 to 200,000 Attribute values while SQLite evaluated 10,000 to 2,000,000 values. All of the experiments were limited to approximately 1 hour in duration, which included all steps described above aside from the individual Attribute Update experiments, which were performed separately. This limitation prevented the slower-performing MySQL implementation from being tested on the same large scales as the SQLite version. When comparing the two versions, careful attention should be paid to the independent (horizontal) axes of the graphs.

6.3 Updating Attributes

One of the most common actions that a World Model performs is to accept a newly-generated value for an Attribute and insert it into persistent storage, forward it to Clients with Standing queries, or both. The World Model needs to be able to perform this basic task quickly. A slow update/insert introduces undesirable latency between components, particularly if data undergoes several "rounds" of the analysis-update cycle before being presented to the final Application. Updates are write operations and are consequently critical sections for maintaining consistency among multiple threads of operation. While multiple read operations can be performed concurrently without a need for synchronization, writes must be performed in a consistent way to ensure integrity. Maintaining low time costs for writes is a top priority when choosing a World Model implementation.

Figures 6.1, 6.2, 6.3, and 6.4 show the results of experiments where a number of Attribute values were updated sequentially in the World Model from a single Solver connection. One of the most striking features of these graphs is the apparent disparity between the MySQL and SQLite implementations. In both sets of graphs, and in nearly all subsequent results described in this chapter, the MySQL implementation is noticeably slower than the SQLite implementation. A thorough discussion of this trend is provided in Section 6.8. In brief, the SQLite implementation has been the most actively used, and so is the most "tuned" for performance at present.

Figures 6.1 and 6.2 both show the 95^{th} percentile times for Attribute updates over the full range of data set sizes tested. For each data set size, each experiment was tested both with and without a Client application that has a Standing query for all newly-updated Attributes to be forwarded to it. For the SQLite implementation (Figure 6.2), it is clear that the Receiver does not have a

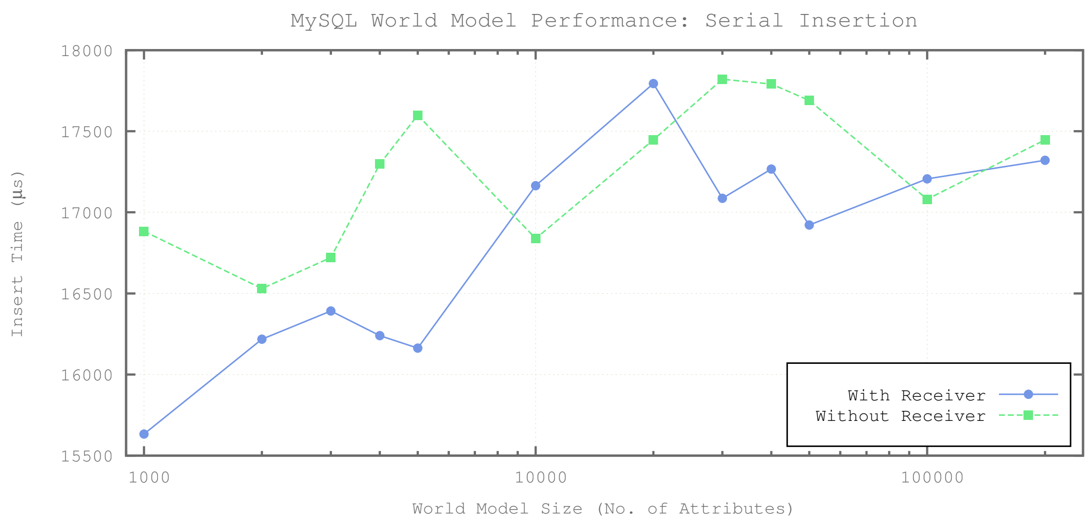


Figure 6.1: MySQL World Model Attribute insert time (95 percentile) versus size of resulting data set.

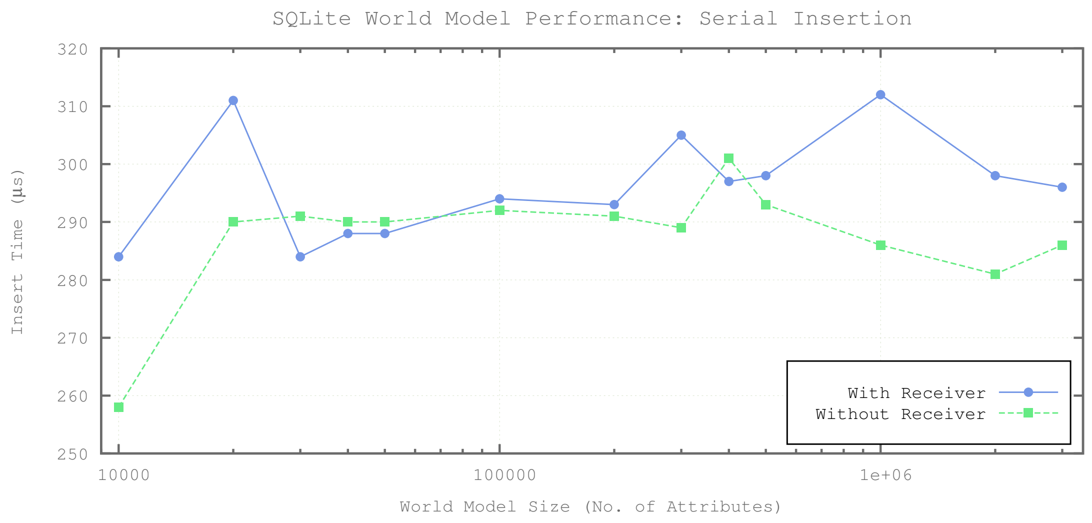


Figure 6.2: SQLite World Model Attribute insert time (95 percentile) versus size of resulting data set.

negative impact on performance. In most of the data points, it appears that it may actually *improve* performance, although this is very unlikely. The performance variation within the *with-Receiver* and *without-Receiver* data sets indicates that this trend is most likely a statistical aberration. The MySQL implementation in Figure 6.1 shows that the *with-Receiver* performance is slightly slower than without. Overall, both implementations demonstrate relatively stable Update performance as the data set changes by two orders of magnitude. The results of each experiment are within around 20% across data set sizes, with the MySQL implementation showing the slightest upward trend as the data set increases in size.

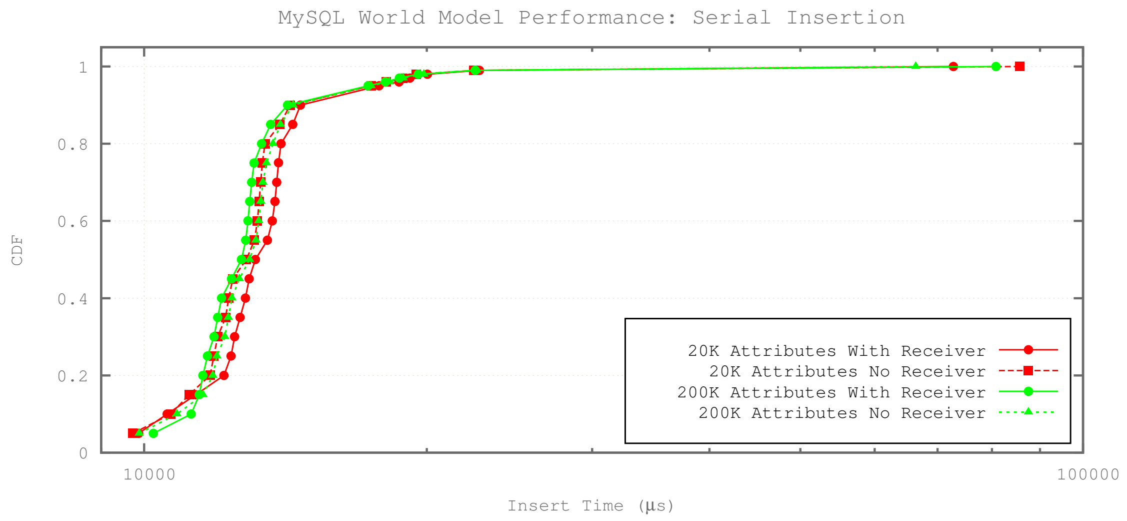


Figure 6.3: MySQL World Model Attribute insert time CDF for 20,000- and 200,000-Attribute data sets.

Figures 6.3 and 6.4 provide additional detail about the Update performance of the two versions. Two data sets available for both implementations were chosen for a closer comparison. Notable in both is the significant “long tail” in the last 1% of Updates. For both SQLite and MySQL, these may be the result of disk write-throughs, context switching, or cache misses. In particular, the first Update operation was frequently much slower than subsequent operations, although small “spikes” in the overall performance were noted in the full traces. The SQLite implementation demonstrates a very tight distribution, staying below 200 microseconds for 90% of the Updates. The MySQL version has a wider distribution, from 10–20 milliseconds most of the time, with a maximum around 90 milliseconds. In both versions the maximum time is at tens of milliseconds, indicating a shared bottleneck for the slowest operations.

Overall, the SQLite version clearly outperforms the MySQL implementation in these experiments. The MySQL version, however, still performs relatively well and takes fewer than 20 milliseconds to

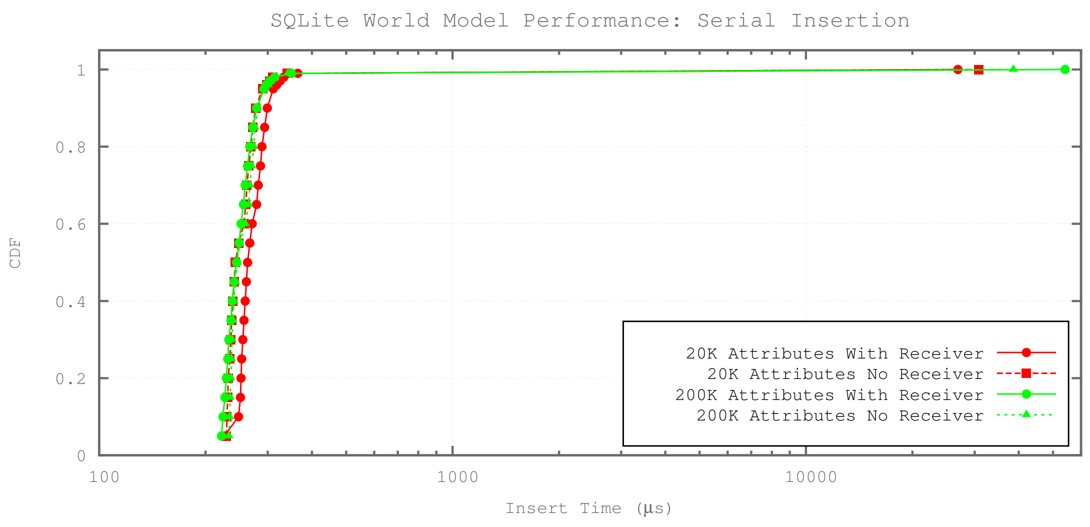


Figure 6.4: SQLite World Model Attribute insert time CDF for 20,000- and 200,000-Attribute data sets.

perform the vast majority of Update operations. Aside from the most time-critical applications, this may be sufficient to satisfy clients.

6.4 Expiring and Deleting Attributes

When a single Attribute has multiple values from the same Origin in the World Model, each value is valid for a limited period of time. Specifically, each Attribute value is only valid between its Creation Timestamp and the Creation Timestamp of the value that follows it chronologically. When Attributes are updated in the World Model, previous values are automatically invalidated, or Expired, with the value of the incoming value's Creation Timestamp. This approach works well in the general case where Solvers produce a value indicating the most current state of some measured value. In other situations it is more appropriate for a Solver to explicitly expire an Attribute value without generating a new value. As an example, this may occur when a Solver has not received data from a sensor over a period of time.

While expiring an Attribute value is appropriate when it is no longer considered valid after a specific point in time, deleting an Attribute value (or entire Identifier entry) results in all data for that Attribute being completely removed from the World Model. A destructive operation, deleting an Attribute value is usually considered an administrative operation. For instance, a Solver may have produced incorrect values due to a coding error or corrupt input data, or a policy changes and certain values need to be removed. The World Model is not currently designed to support deleting a subset of Attribute values from a specific origin, and so deleting an Attribute value results in the complete removal of all values for that Attribute from that Origin.

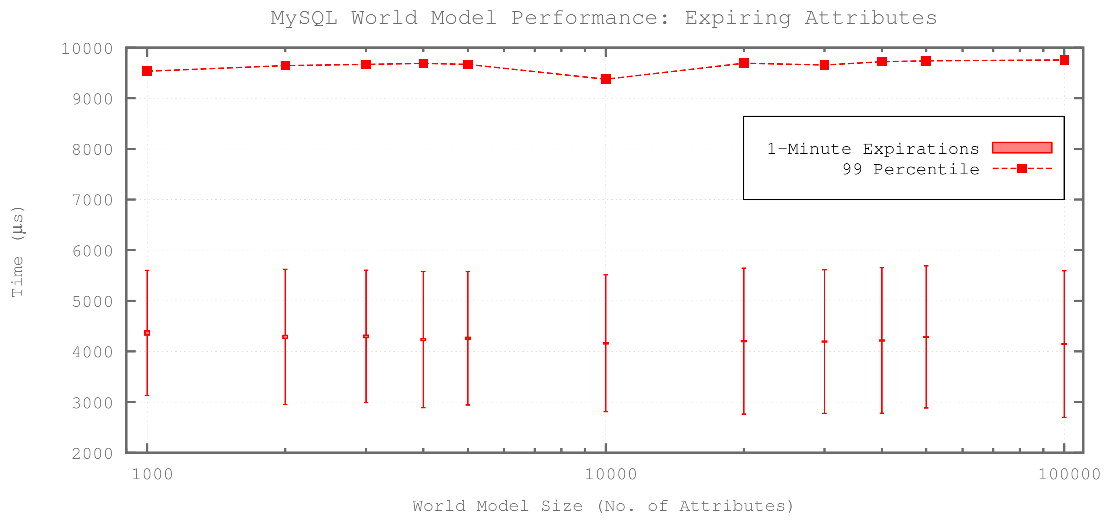


Figure 6.5: MySQL World Model Attribute expiration time versus data set size. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99^{th} percentile is plotted alongside.

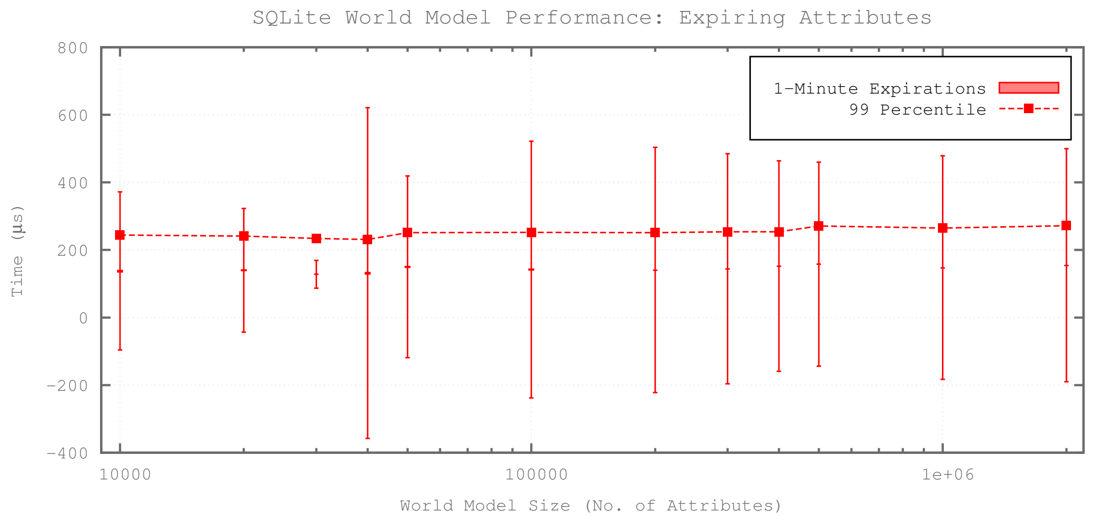


Figure 6.6: SQLite World Model Attribute expiration time versus data set size. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99^{th} percentile is plotted alongside.

Figures 6.5, 6.6 provide timing statistics for expiration operations taking place in the MySQL and SQLite implementations of the World Model. In both graphs, the vertical lines consist of boxes centered on the mean with upper and lower bounds at one standard error of the mean, SE_M. The lines above and below each box are bounded at one standard deviation, σ. The lines crossing the data graphs show the 99^{th} percentile values for the expiration operations. The MySQL version in Figure 6.5 shows consistent performance numbers across all data set sizes. Notably, the 99^{th} percentile timing is almost double the mean. In contrast, the SQLite version (Figure 6.6) shows a very tight distribution around the mean with all but the top 1% of operations constrained to within one standard deviation in nearly all experiments.

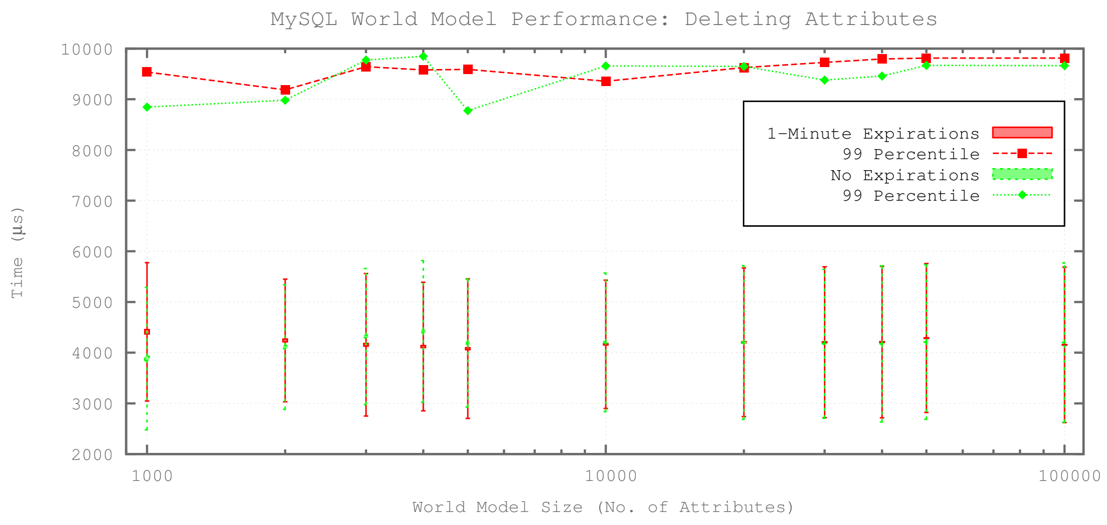


Figure 6.7: MySQL World Model Attribute delete time versus data set size. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99^{th} percentile is plotted alongside.

Deleting Attribute values takes nearly the same amount of time as performing Expire operations. Figure 6.7 shows the time taken to delete individual Attribute values for the MySQL implementation. Experiments with expired and unexpired values, where the value is valid indefinitely beyond the Creation Timestamp, exhibit nearly equivalent performance. Again, the 99^{th} percentile operations require nearly double the mean time to execute. The SQLite implementation (Figure 6.8) is slightly different from the expiration results, with a wider distribution overall.

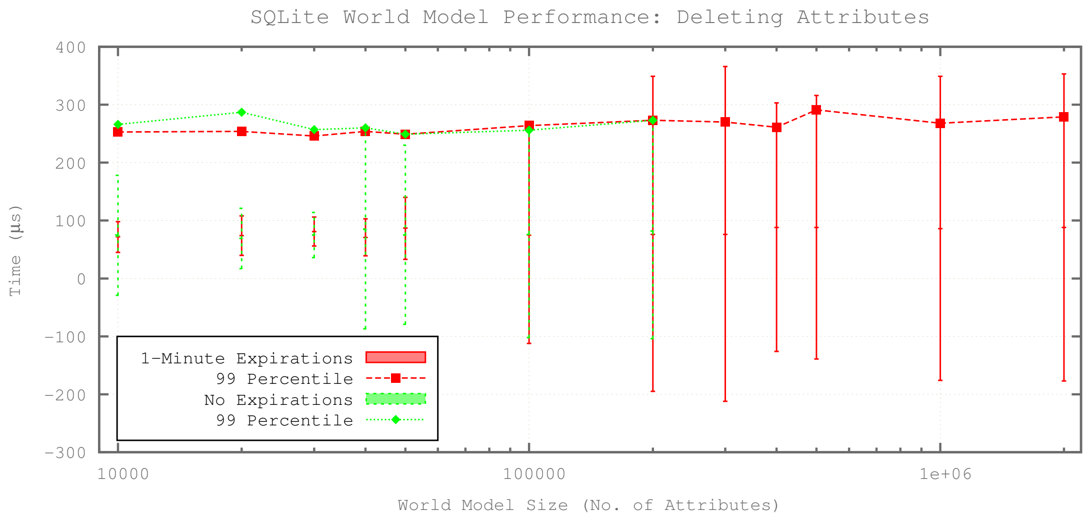


Figure 6.8: SQLite World Model Attribute delete time versus data set size. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99^{th} percentile is plotted alongside.

6.5 Regular Expression Search

The World Model client interface is focused on search and retrieval of data produced by Solvers. One of the most basic operations for clients to perform is a search over the Identifier space to determine which values are present in the World Model. Although Owl Platform does not enforce naming conventions or structure to the values used for Identifiers and Attribute names, it provides facilities to support a wide range of such conventions. The next series of experiments was performed to determine how the complexity of regular expressions submitted by clients to perform searches impacted the time required to perform the search. The World Model also allows regular expression matching on Attribute names, enabling clients to request data for the same properties but without knowing which Identifiers contain those Attributes.

Each World Model server utilizes a different regular expression engine. The MySQL implementation relies on the regular expression library provided in its distribution. The SQLite library does not provide a regular expression engine, but does provide the capability to link to a user-provided library. The SQLite World Model implementation uses the C++11 “regex” library to provide the capability to the SQLite database engine. Performance of the regular expression implementation impacts a large number of operations in the World Model because it is used to match both Identifiers and Attribute Names. For instance, each Standing Query by a Client connection results in an execution of regular expression matching whenever a new Attribute value arrives. Although it would be possible to cache the results of these queries, trading memory for processing time, it is not clear that such an approach would result in appreciable gains in performance.

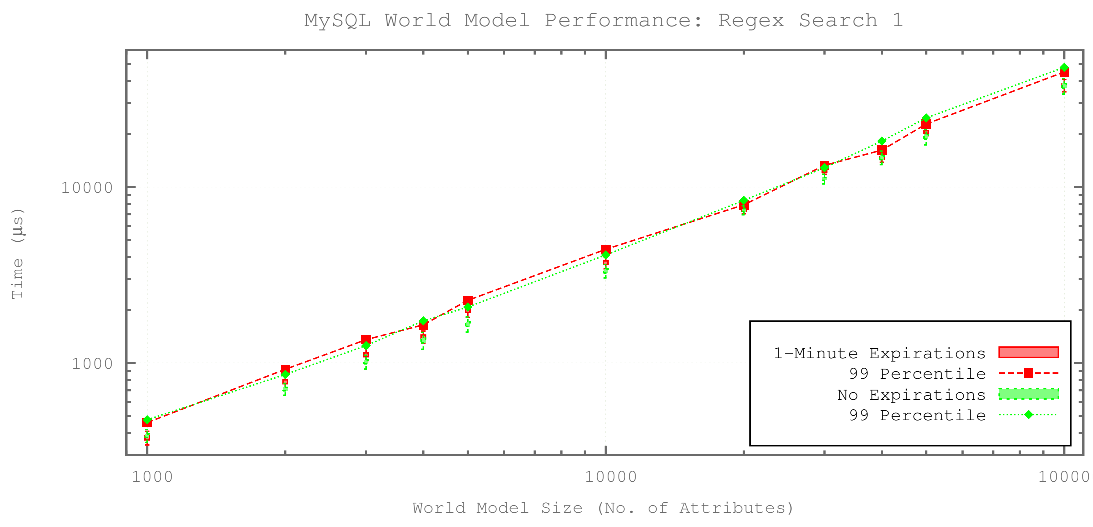


Figure 6.9: MySQL World Model regular expression search time for query ".*", which matches the entire data set. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99th percentile is plotted alongside.

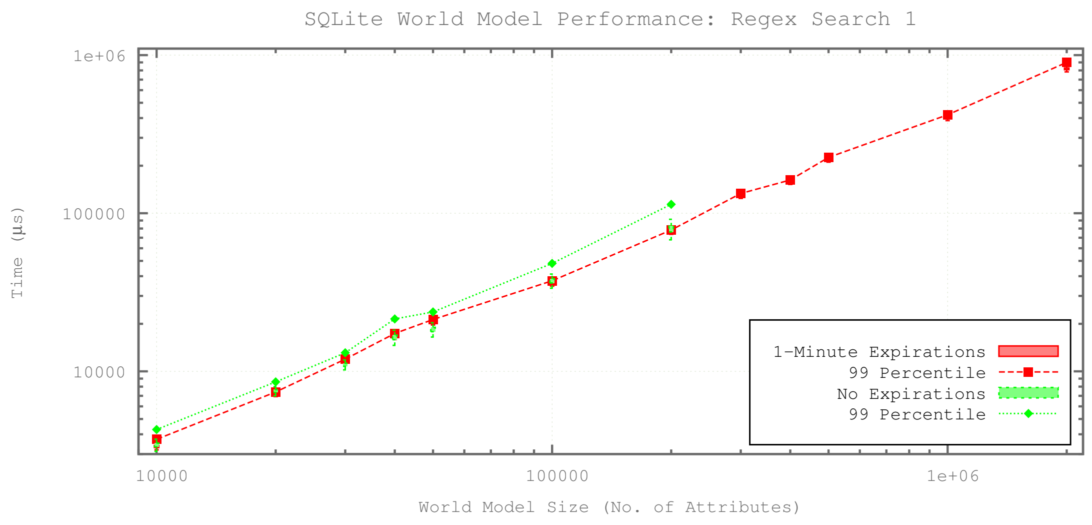


Figure 6.10: SQLite World Model regular expression search time for query ".*", which matches the entire data set. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99th percentile is plotted alongside.

Figures 6.9 and 6.10 detail search times in the MySQL and SQLite implementations of the World Model, respectively, for the regular expression ".*". The "." matches any single character, and the "*" allows matching zero or more times. This particular pattern is convenient when a Client connection wishes to select all Identifiers available, perhaps narrowing the requested data by using a more discriminating regular expression for Attributes. In both figures, results are plotted for the 99^{th} percentile as well as box plots centered on the mean, bounded at one standard error, and with one standard deviation marked as vertical lines extending above and below. This pattern is very commonly used, as it matches all possible values. The MySQL implementation is slightly slower than SQLite, taking around 30% longer at 100,000 entries, though both perform well enough to support a large range of potential applications.

Important to note when interpreting these results, is that the search is over tens or hundreds of thousands of individual values. Identifiers represent discrete objects, events, or contexts, and Attributes represent their measurable characteristics. A more typical scenario for a World Model would be to have a few thousand, even ten or twenty thousand unique entries (Identifiers/Attributes), and those entries would have longitudinal data values over long periods of time. For instance, a system for building management might have an Identifier for every door, window, and room in the building. A 20-story tower might have a hundred rooms, a hundred windows (not all rooms have windows, some have more than one), and a few hundred doors. If there are 500 Identifiers for each floor and 20 floors, there are approximately 10,000 entries. Both MySQL and SQLite were able to perform this search in fewer than 5 milliseconds, which should be sufficient for many applications.

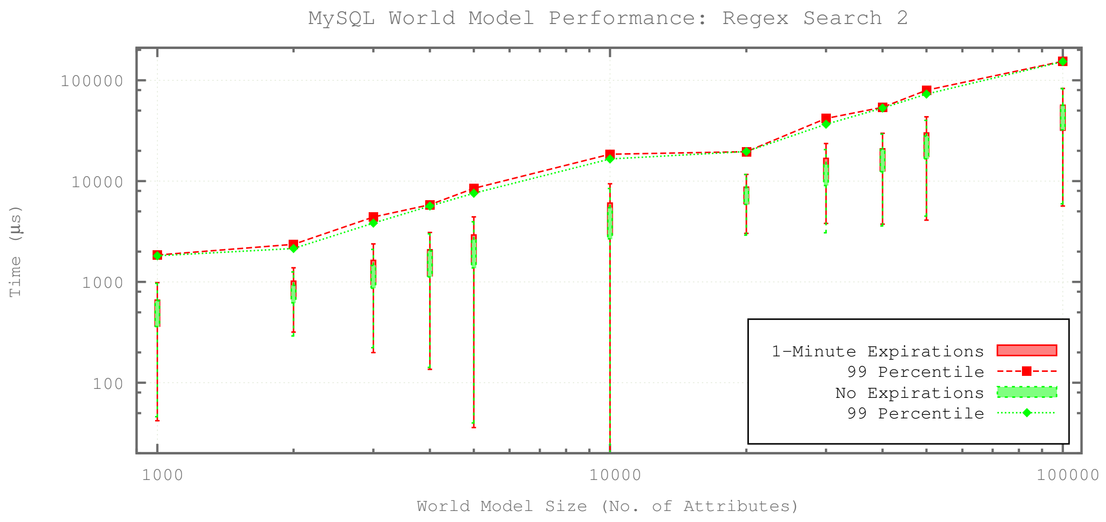


Figure 6.11: MySQL World Model regular expression search time for query "ˆ(.*?[a-z])(.*[0-9]){8}0", which matches 10% of the data set. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99th percentile is plotted alongside.

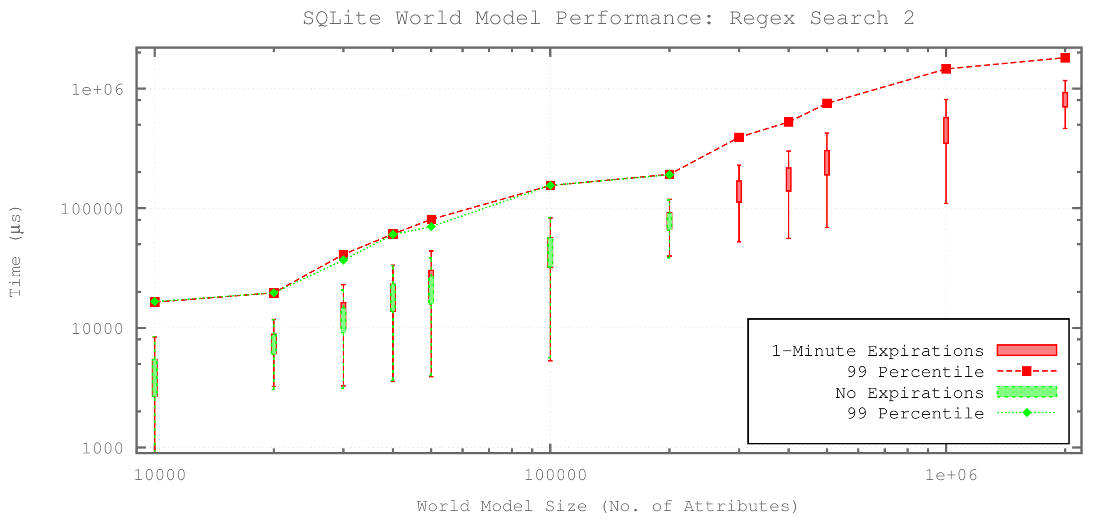


Figure 6.12: SQLite World Model regular expression search time for query "ˆ(.*?[a-z])(.*[0-9]){8}0", which matches 10% of the data set. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99th percentile is plotted alongside.

Figures 6.11 and 6.12 show the results for a slightly more complex regular expression that matches 10% of the data set. This regular expression represents an search of moderate complexity that might be used to select out a specific subset of Identifiers that utilize a regular naming scheme. For both MySQL and SQLite, the time required to execute is longer than for the first regular expression. At

100,000 entries, MySQL timing increases from 45 milliseconds to around 150, and SQLite rises from around 35 milliseconds to 150 milliseconds.

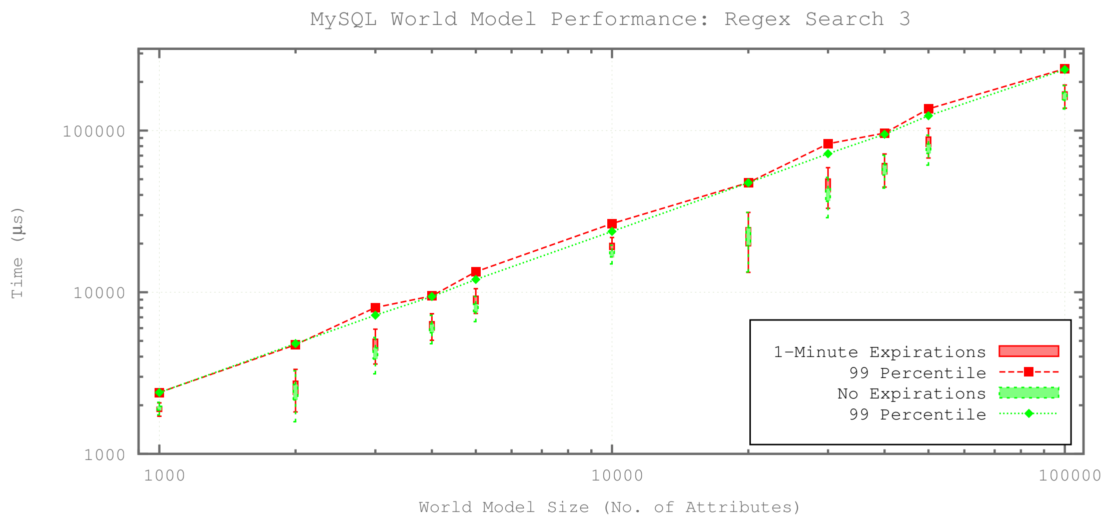


Figure 6.13: MySQL World Model regular expression search time for query "(([^\.]*))*1", which matches 10% of the data set. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99^{th} percentile is plotted alongside.

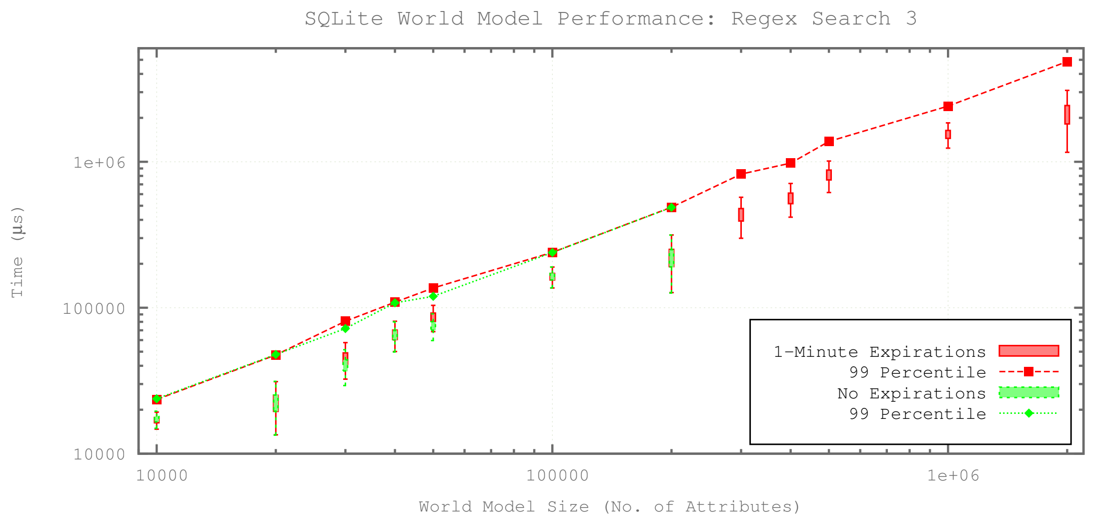


Figure 6.14: SQLite World Model regular expression search time for query "(([^\.]*))*1", which matches 10% of the data set. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99^{th} percentile is plotted alongside.

The final two regular expressions evaluated involve "backtracking" during evaluation, and serve as upper bounds on the complexity of requests made during normal operation. The expressions selected were intentially designed to require significant computation and to be highly inefficient. Regular expression 3, “(([ˆ\.]*))*1” has a nested conditional clause and requires that the matched value end with a “1”. A much more efficient regular expression would simply match a “1” at the end of the string (e.g., “1$”) without needless backtracking and reevaluation of the input string. The fourth regular expression is much like the third, except that it will not match any of the Identifiers used in the experiments. This causes it to take even longer to execute than the previous expression, demonstrated in Figures 6.15 and 6.14, where a search over 100,000 Identifiers takes 250 milliseconds for both MySQL and SQLite. In normal practice, it is common for programmers to work hard to design efficient regular expressions and eliminate unnecessary delays and overhead, though this is not always the case. Both MySQL and SQLite versions perform reasonably well when searching modest numbers of Identifiers.

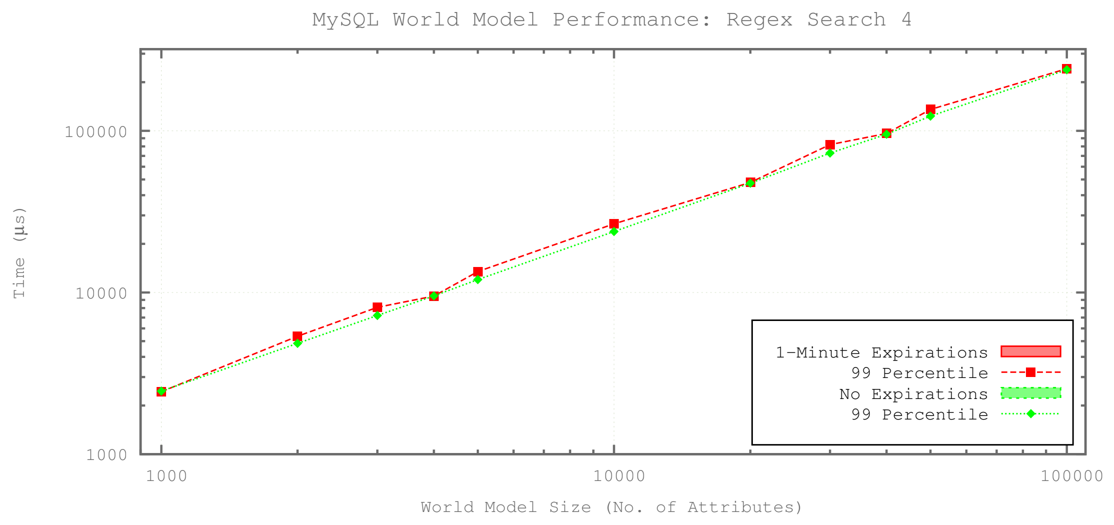


Figure 6.15: MySQL World Model regular expression search time for query “(([ˆ\.]*))*p”, which does not match any IDs and requires expensive backtracking to perform. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99^{th} percentile is plotted alongside.

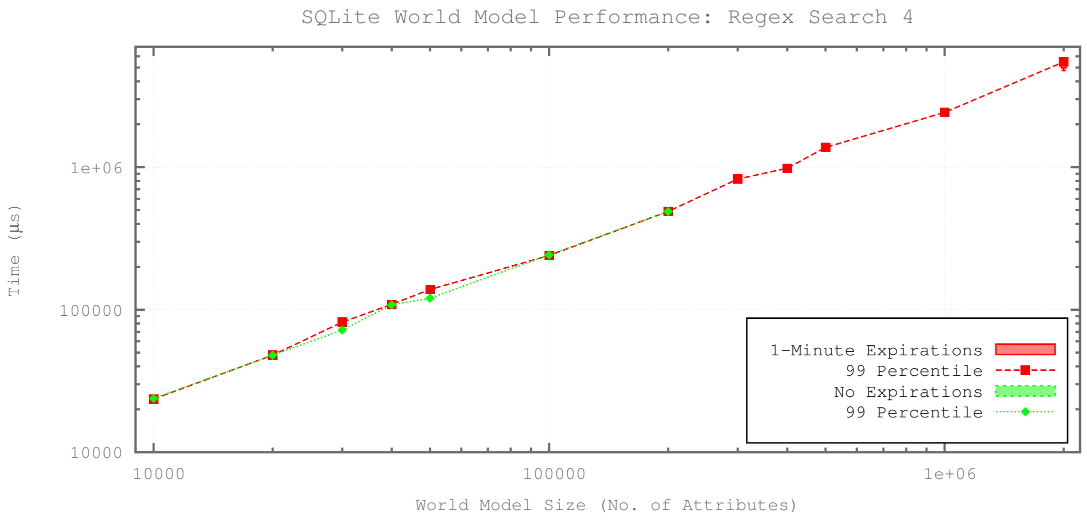


Figure 6.16: SQLite World Model regular expression search time for query “(([^\.]*))*p”, which does not match any IDs and requires expensive backtracking to perform. Boxes are centered on the mean and bounded at $\pm SE_M$ with whisker bars at $\pm\sigma$, and the 99th percentile is plotted alongside.

6.6 Range Queries

A Range query retrieves a block of time-contiguous data for a selected set of Identifiers and Attribute names. It can be useful when finding minima and maxima over a period of time, calculate average values, or other aggregate computations. Range queries perform selections on both Identifier and Attribute names, and explicitly state the start and end timestamps for the values desired. The query start and end timestamp values limit the returned data such that each Attribute value returned must have a Creation Timestamp after or equal to the query's start timestamp and before or equal to the query's ending timestamp. The experiments each performed a series of 3 different range queries, for 10%, 5%, and 1% of the data set each. The queries overlapped in time, setting the next start timestamp to the previous end timestamp, and matched all Identifiers and Attribute names using the ".*" regular expression.

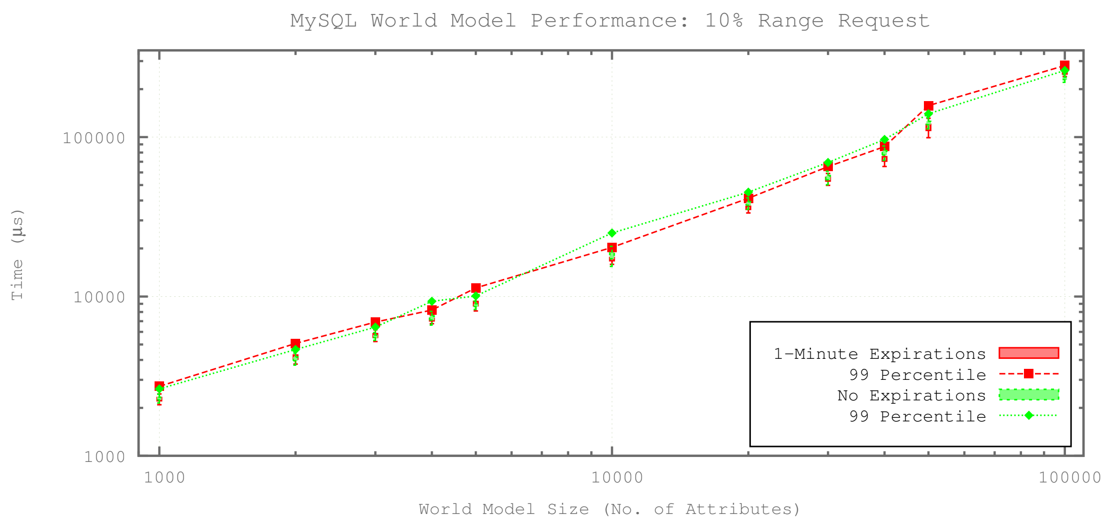


Figure 6.17: MySQL World Model range query time when requesting 10% of the total data set size. Timing is measured at the World Model and excludes any network delays.

Figures 6.17 and 6.18 show how long it took to perform each of the ten "10%" queries over data sets of different sizes. Attribute values were either expired after 1 minute from their Creation Timestamp values or not expired at all. In both the expired and non-expired experiments, the returned Attribute values will be the same for each Range request, since it only matches those Attributes with a Creation Timestamp after or equal to the query start timestamp, and does not return Attributes valid before that time even if they are valid within the time range.

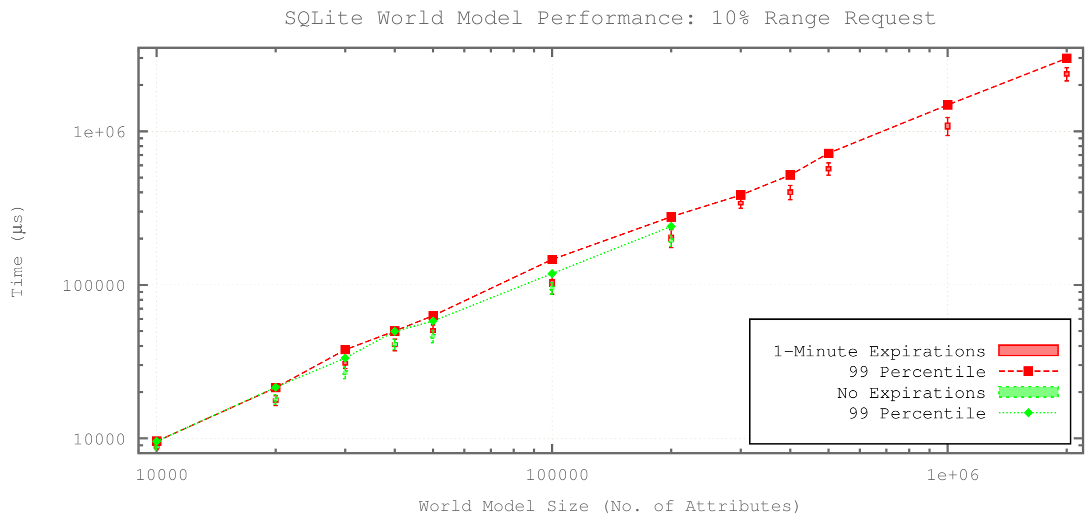


Figure 6.18: SQLite World Model range query time when requesting 10% of the total data set size. Timing is measured at the World Model and excludes any network delays.

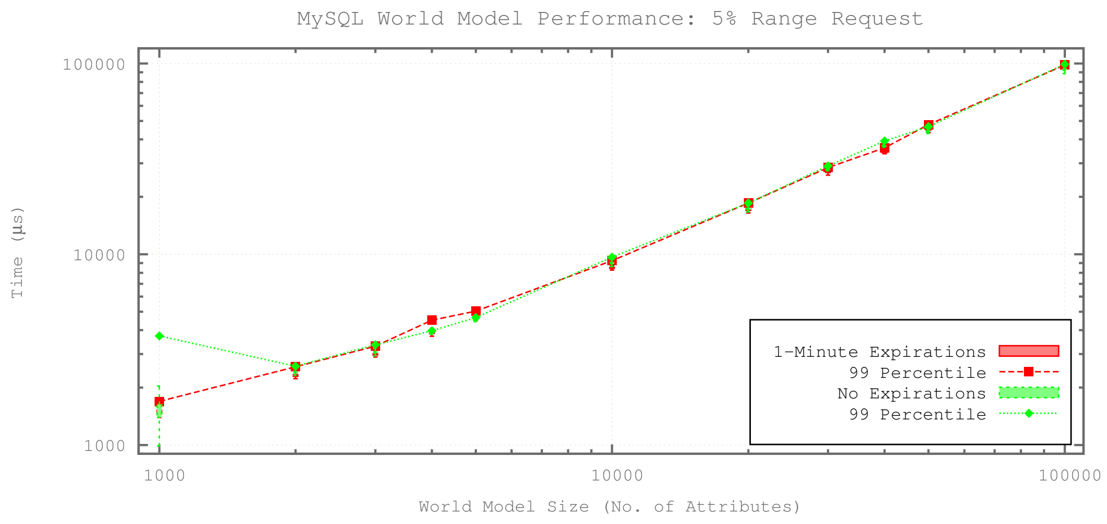


Figure 6.19: MySQL World Model range query time when requesting 5% of the total data set size. Timing is measured at the World Model and excludes any network delays.

Figures 6.19, 6.20, 6.21, and 6.22 detail the time taken to select 5% and 1% of the data set through Range queries. In both sets of queries the SQLite version outperforms the MySQL implementation, taking around half the time to perform the same operations. Overall, however, the performance of both implementations is sufficient for many applications, taking between 50 and 100 milliseconds to retrieve 5,000 Attribute values from each World Model.

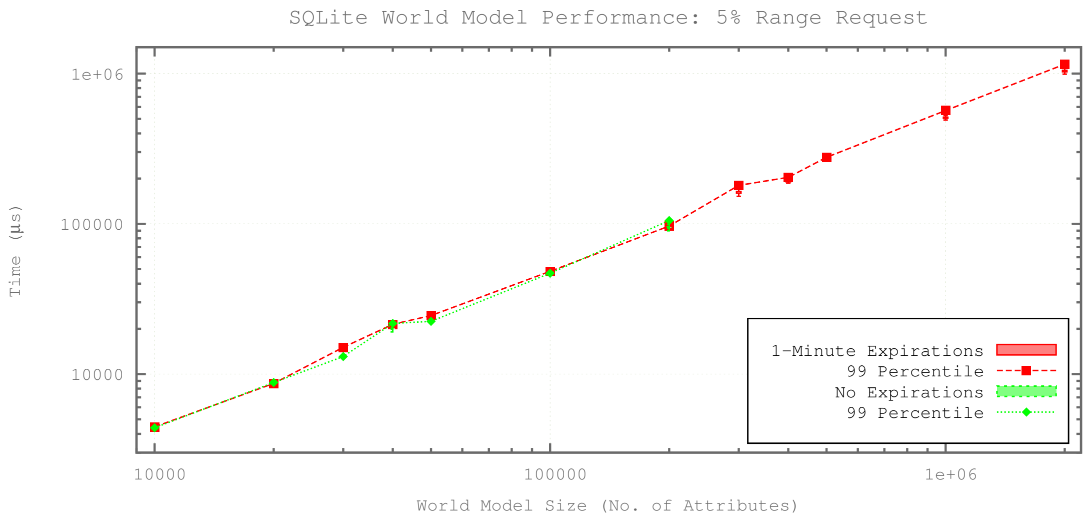


Figure 6.20: SQLite World Model range query time when requesting 5% of the total data set size. Timing is measured at the World Model and excludes any network delays.

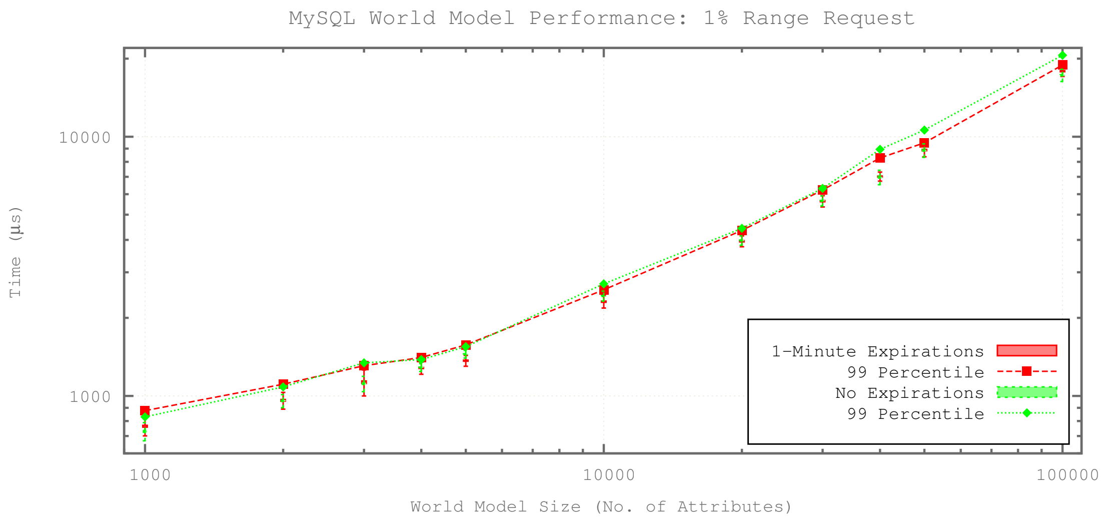


Figure 6.21: MySQL World Model range query time when requesting 1% of the total data set size. Timing is measured at the World Model and excludes any network delays.

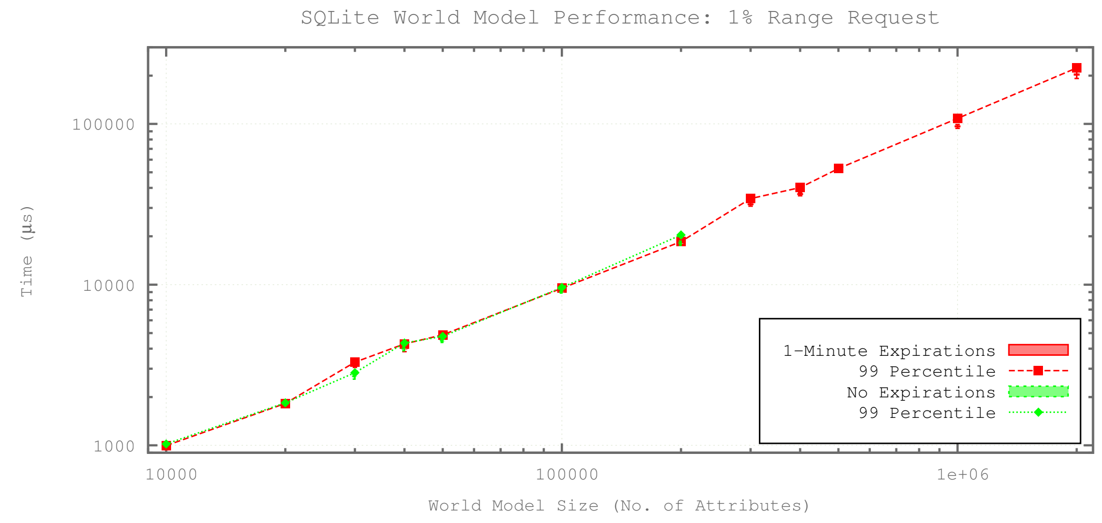


Figure 6.22: SQLite World Model range query time when requesting 1% of the total data set size. Timing is measured at the World Model and excludes any network delays.

6.7 Snapshot Queries

Snapshot queries are used to retrieve the “current” state of the World Model at some selected point in time. Specifically, the timestamp provided in the request is matched against the Creation and Expiration Timestamps of Attribute values. Any Attribute value with a Creation Timestamp at or before the query timestamp and an Expiration Timestamp value at or after the timestamp is returned to the requesting Client. This can be particularly useful if some of the values in the World Model are generated long after a physical event or reading takes place. For instance, a sensor may gather data once per minute and store it locally. This data may then be transmitted to a WSN sink node and sent into the Owl Platform World Model once per hour or per day. The timestamp for the Attribute value should match the time when the data was collected, not when it was entered into the system. WSNs that carefully budget radio or network communication to conserve energy can still be effectively used to create a retrospective view of the system.

Snapshot queries are distinguished from a Range query where start and end timestamps are the same, in the way that the Attribute values are evaluated. In a Range query, only those Attribute values with Creation Timestamp values at or after the start timestamp of the query are returned. In other words, a Range query provides a sequence of Attribute values that *would have been sent to a Standing query* if all of those values had arrived at the time indicated in their respective Creation Timestamps. In a sense, a Range request can act as a “replay” mechanism in the World Model. In contrast, a Snapshot query provides the Attribute values which are valid at the time specified in the query. Even if the Creation Timestamp for a value is years prior to the query timestamp, it is included in the response so long as the Expiration Timestamp is equal to or greater than the query timestamp.

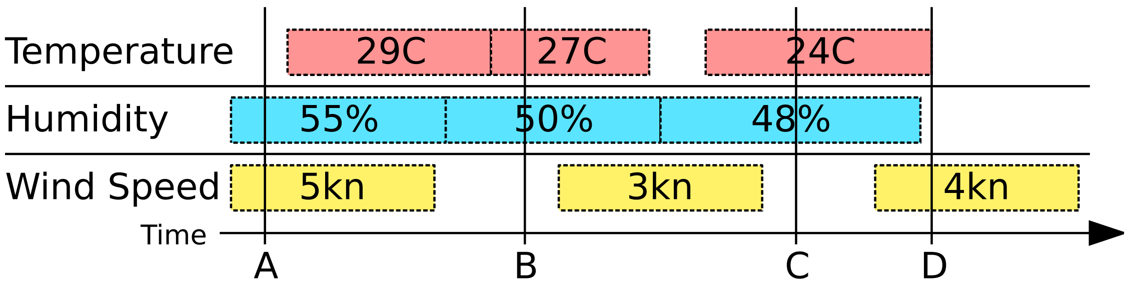


Figure 6.23: Diagram showing the change in 3 Attribute values over time. The horizontal axis indicates the passage of time with the past to the left and the future to the right.

For example, Figure 6.23 depicts a situation where three different Attribute values are changing over time. At time *A*, a Snapshot query for all three attributes would return only the values (Humidity→55%, Wind Speed→5kn). Likewise, a Snapshot query at time *B* would return (Temperature→27C, Humidity→50%). A Range query between times *A* and *B* would return the values (Temperature→29C, Humidity→50%, Temperature→27C), in that order. At time *D*, a Snapshot query would return only (Wind Speed→4kn) because the Temperature value had expired *at that moment* and the Humidity value had expired before *D*. Similarly, a Range query between *C* and *D* would also only return (Wind Speed→4kn) because it was the only value with a Creation Timestamp within the requested time range.

In the World Model evaluation experiments, each Attribute value has a Creation Timestamp that is set to some n minutes in the past, where n is the ordinal value of the Attribute value when generated by the profiling program, and the 0^{th} Attribute value is marked with the time at the start of the experiment. The Profiler then generates a Snapshot request at a specific point within the data set's time range. For instance, if the Attribute values span 1000 minutes, then the "2.5%" Snapshot query timestamp would be set 975 minutes before the first (futuremost) Attribute value. Likewise, a "97.5%" Snapshot query timestamp would be set 25 minutes before the first Attribute value. In the case where no Attribute values are expired, the Snapshot query will return every Attribute value that has a Creation Timestamp value equal to or earlier than the query timestamp. Consequently, the equivalent Snapshot query for unexpired Attribute values will match more Attribute values than the expired data set, except when the query timestamp value is equal to the Creation Timestamp of the oldest Attribute value in the data set.

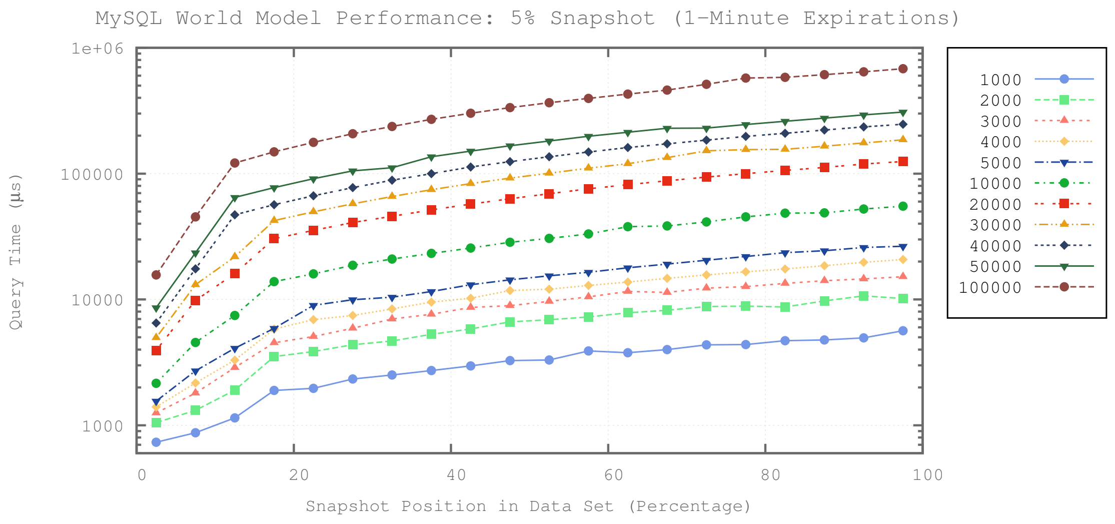


Figure 6.24: MySQL World Model snapshot query time at different positions within the total data set. With 1-minute expirations, each Snapshot returned a single non-expired Attribute value.

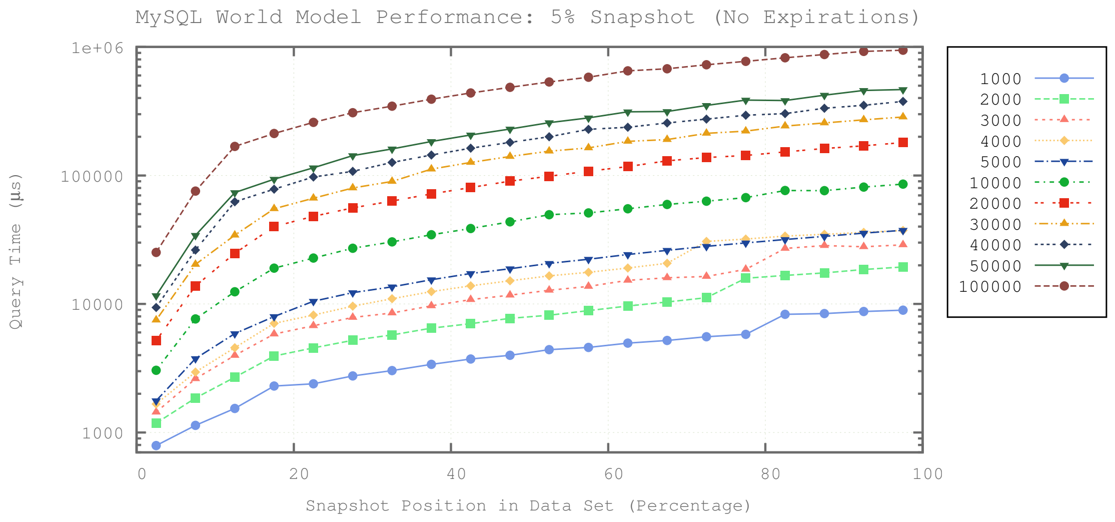


Figure 6.25: MySQL World Model snapshot query time at different positions within the total data set. Without expirations, each Snapshot returned all Attributes created at or before the timestamp sent.

Figures 6.24 and 6.25 show the time taken for the MySQL World Model to perform a sequence of Snapshot queries, where the timestamp is set to different points within the data set time range. Although the overall trend in the two experimental result sets is similar, the no-expirations data set takes longer to execute the query. The reason is that the query first selects the data from the database based on the timestamp values, and then performs the regular expression matching on the

Identifiers and Attribute names. In the case of the no-expire values, the time-matched set is larger and additional executions of regular expression matching take place.

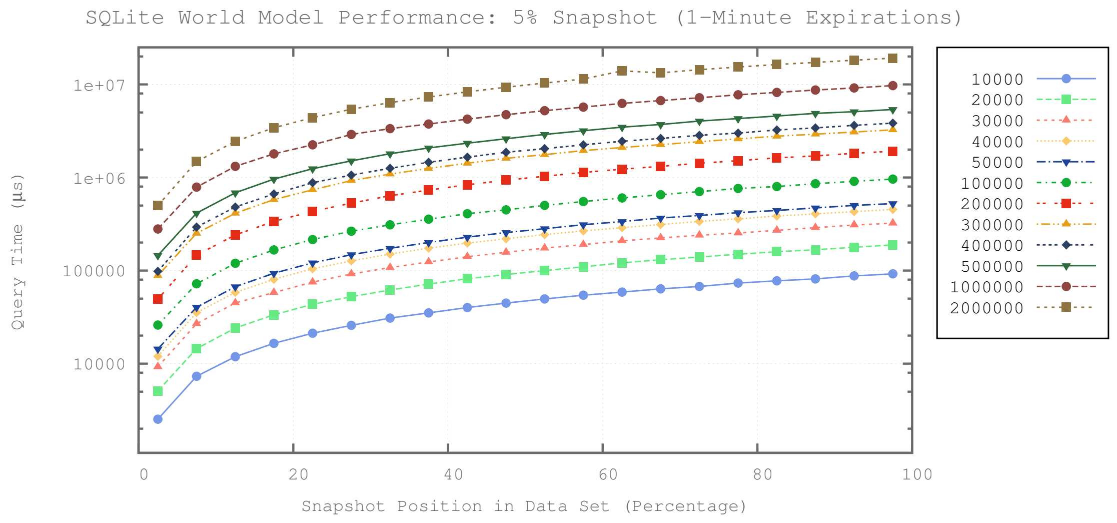


Figure 6.26: SQLite World Model snapshot query time at different positions within the total data set. With 1-minute expirations, each Snapshot returned a single non-expired Attribute value.

Figures 6.26 and 6.27 show results of similar experiments for the SQLite implementation. In an notable change of pace, the SQLite version takes longer than the MySQL version to execute the Snapshot queries. Again, the execution time is generally linear with respect to the size of the data set and the expired data queries run faster as a result of the smaller result set used for regular expression matching.

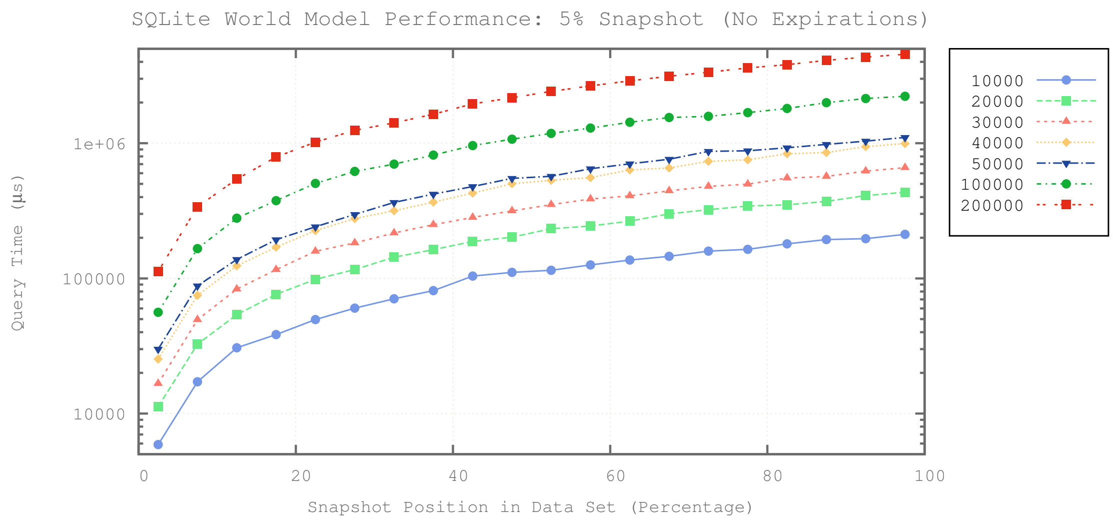


Figure 6.27: SQLite World Model snapshot query time at different positions within the total data set. Without expirations, each Snapshot returned all Attributes created at or before the timestamp sent.

6.8 Improving Performance

Overall, the performance of both World Model implementations appears to be sufficient for modest numbers of Identifier entries. As far as could be determined through the experiments described in this chapter, the performance of both implementations was roughly linear with the size of the data set in use. Linearity with storage size is a strong indicator that these prototype implementations are capable of standing up to modest workloads. Unfortunately, when data set sizes grow to several hundred thousand or millions of Identifiers, the performance of individual queries could easily become a bottleneck in the system. For instance, if a user-facing web page dynamically loaded data to populate a calendar or graph, a 500,000 entry data set in the SQLite World Model would take nearly 1 second to execute a 10% range query. The results are even worse for MySQL. Clearly, improvements need to be made for a World Model to scale into the millions or tens of millions of Identifier entries.

Both MySQL and SQLite should be able to benefit from query optimizations. A single query may result in the same operations being executed millions of times, and shaving off even 1-2 nanoseconds from each operation can result in significant gains in performance. By working closely with each database's available performance tuning tools, it should be possible to reduce the cost of the queries and procedures used. In addition to query planning, careful design of tables and index definitions should further reduce the number of page accesses required. The MySQL implementation uses separate tables for storing unique Attribute Names, Identifiers, and Origin values, and queries need not load the primary data table into memory to check values. This type of optimization could be also be ported back to the SQLite database schema.

The largest challenge for the MySQL implementation would appear to be optimization of the stored procedures currently used. Data manipulating procedures (expire, update, delete) contain significant amounts of application logic, which reduces the number of executions that can take place simultaneously. Instead, the logic should be extracted back to the World Model process code and the database should focus solely on data management tasks. Additional improvements would involve divorcing the network messaging portion from the database interaction code, such that messages and database inserts/reads can be handled asynchronously from one another. This, in combination with a database connection pool, would allow the World Model to exploit MySQL's abilities to efficiently handle multiple connections.

The SQLite implementation is likely in need of a redesign of the database schema used, to more closely match the one in use by MySQL. Additional indexes and careful query planning will also enable improvements in efficiency. Due to it's intended purpose for small, application-specific databases, the SQLite implementation may not be appropriate for large-scale systems. Instead, it could be tuned and optimized to work well in contexts where a full-sized database server (like MySQL) is not available.

Chapter 7
Experiences in Programming and Use

Owl Platform began as a research project designed to make management and utilization of Wireless Sensor Network data easier. Over several years, it has been used in a number of different ways by different groups. In some cases, it has been used to speed-up data collection for experimernts, or to combine multiple systems for analysis. From its earliest stages, it has been used by the development team, other graduate students, and even undergraduate students for various projects. Most recently, it has become one of the core technology platforms used at a start-up company developing a wireless sensor product. Whenever it was used for these projects, new insights into both its strengths and limitations were found. In fact, transforming the original World Server/Distributor model into a single World Model component was strongly motivated by the experiences of an undergraduate using it just a few months after the first prototypes were completed. Overall, it has demonstrated itself to be a flexible, robust system for collecting, analyzing, and disseminating WSN data in support of many differing goals.

7.1 Research Tools

As the GRAIL RTLS changed into the Owl Platform, the first applications of the new system were focused on reproducing previous work. Developing, deploying, and testing this new system meant that the previous system was left largely unattended, and in some cases dismantled to reassign the hardware for Owl Platform. Consequently, the need to restore a functioning system for the research group was paramount, and the earliest Solvers and Applications written for Owl Platform were previously extant services, extensions of those, or previously-planned ideas.

One of the first applications to be ported into the new Owl Platform was device-based mobility and localization. The subject of GRAIL version 2 and several years of research, it was highly desired to have a functioning indoor localization system operating again to support research activity. The result of this effort was porting the bayesian localization solver such that it utilized the new network protocols of the Owl Platform rather than the file-based interfaces it relied on in GRAIL version

2. A new mobility-detection solver was also developed and used to initiate localization of devices if they were found to be moving.

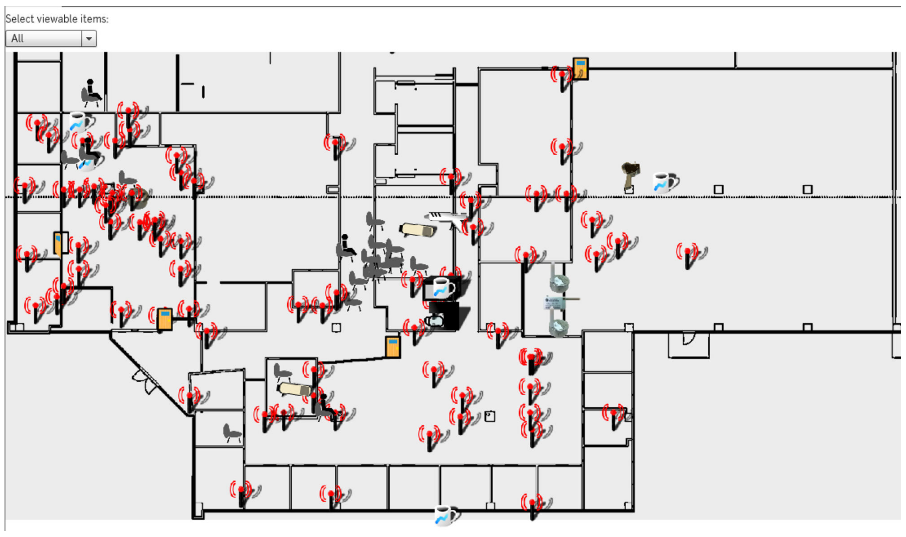


Figure 7.1: Screenshot of the original Flash-based map application.

Equally practical and motivated by research, the localization and mobility components were completed, but without a usable interface for human users. An Adobe Flash®-based map interface (Figure 7.1)was written in a couple of months and made available to the entire research lab where the work was undertaken. Users could navigate to a publicly-accessible website and view the location of sensor tags. To further motivate users to visit the page, and to solicit feedback, Pipsqueak sensors were attached to volunteers' coffee cups so that they could be tracked down if misplaced. A further benefit of this approach was the ability for faculty and students to locate specific individuals within the laboratory space, using the tracked coffee cups as a proxy for the users's location [29].

In the following months, the same map interface was updated to include support for WSN data generated by the sensors. New prototype hardware versions were developed that sensed a variety of physical events including temperature, doors opening and closing, power outlets, and the office coffee maker [12]. The system was also used to support further development of the Pipsqueak sensors by enabling rapid data collection and analysis. Forwarding data from deployed Pipsqueak WSNs enabled both real-time and post-event analysis to understand the physical signal propagation properties and frame collision rates under various configurations.

During the course of these development efforts, it was found that much of the data generated by Solvers could be re-used by other "upstream" Solvers and applications. For instance, RSSI mean and variance data was used to support mobility detection, indoor localization, device-free motion localization [30], and passive localization [40]. Other Solvers and Applications were built to detect and announce freshly-brewed coffee. The ability to reuse data among Solvers and Applications, a design decision from the beginning, made development faster and more efficient. For instance, when a programming error resulted in a miscomputed value (e.g. in RSSI variance values), the error needed to be corrected only once in a single Solver rather than in multiple projects where the variance was used as an input value. In another instance, a demonstration application was developed and made ready for the 2012 New York Maker Faire in just 3 days.

External services were also able to be integrated rapidly into the system by exploiting the generalized nature of the data stored in the World Model. In one case, office printers were made accessible to the laboratory staff and students using a web-based map of the office layout. A user would click on the printer icon, select a document, and click a button to print the document. This was particularly helpful for users who did not have the proper printer drivers installed on their computers or mobile phones.

A separate research group in the same laboratory also utilized Owl Platform to aid in the development and deployment of an augmented reality headset [4]. In this case, the researchers modified a Pipsqueak tag to enable tight synchronization with infrared LEDs that provided an accurate way for users to identify their position, direction, and some nearby objects. Strong motivators for this group to choose the Owl Platform was its support for the low-power sensors, low latency of sensor data, and Java libraries that supported rapid development on the Android operating system.

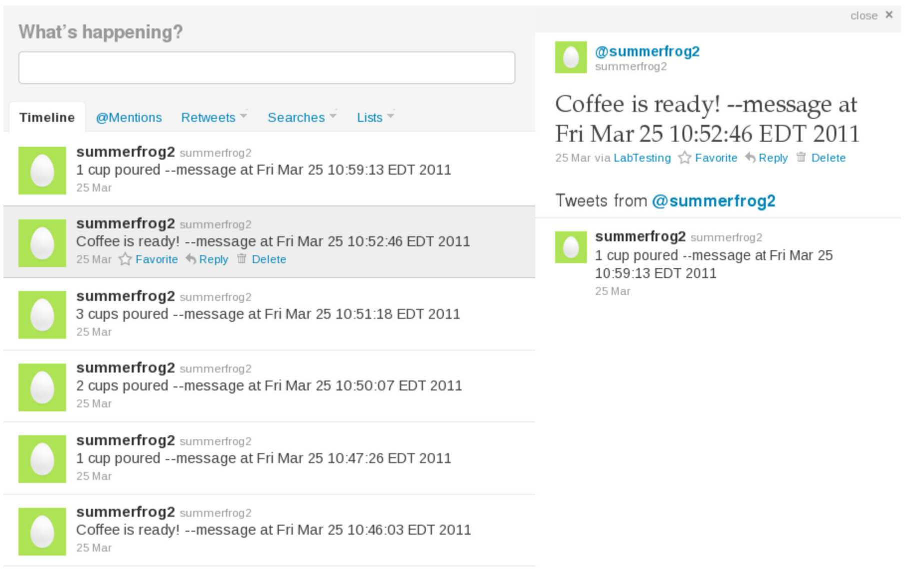
7.2 Undergraduate Projects

Six different undergraduate students utilized the Owl Platform to complete various sensor-related programming tasks, including complex event detection, integrating non-sensor data sources, a new World Model storage engine, and an asset management system. In each case, the student was given a general overview of the Owl Platform, its goals and general structure, and access to publicly-available documentation. Students were either provided with a set of initial tasks to select from and attempt to complete. Where more than one project was completed, they were the result of the student's desire to move in a particular direction with their work. Students were generally allowed to work independently from the research group, except when they would ask specific questions about the architecture or library code usage.

7.2.1 Student 1

With the World Server and Distributor still existing as discrete components, not yet combined into the eventual World Model, a summer intern was recruited to implement a number of applications using the fledgling Owl Platform. The student was given an introduction to the Owl Platform design, was shown the location of technical documents describing both the network protocols and the data types in use, and provided a set of Java-based example applications to use as references. Over a period of 3 months, the student developed 5 different applications using a combination of Pipsqueak sensors, external services, and the Owl Platform services.

The first project was targeted as an introduction to both the sensors and the software the student would encounter. In this project, the student attempted to detect whenever the office refrigerator was left open for a long period of time which could cause food spoilage and wasted electricity. Designed and built in three stages, the student completed the project in approximately two weeks. In stage one, the student wrote a Solver that received RSSI values from the Distributor for a target sensor placed in the refrigerator. When the sensor RSSI value was "low" for a several seconds, the door was declared closed. Likewise, if the RSSI value was "high" for several seconds, the door was declared open. The Solver updated the Distributor with a new data type that indicated at what time the door was opened or closed. The second stage was an improvement on the first and produced a histogram of RSSI values over a sliding window. The Solver would analyze the histogram values, and declare the refrigerator door to be open or closed depending on a change in median values over time. The third stage of the project was a Solver that used the output of the histogram-based Solver to produce a "door ajar" value in the Distributor if the refrigerator door had been left open for longer than a specified period of time. The result of this Solver would be used in the next project, a notification interface for users to receive real-time updates for events.



summerfrog2 summerfrog2
1 cup poured --message at Fri Mar 25 10:59:13 EDT 2011
25 Mar

summerfrog2 summerfrog2
Coffee is ready! --message at Fri Mar 25 10:52:46 EDT 2011
25 Mar

summerfrog2 summerfrog2
3 cups poured --message at Fri Mar 25 10:51:18 EDT 2011
25 Mar

summerfrog2 summerfrog2
2 cups poured --message at Fri Mar 25 10:50:07 EDT 2011
25 Mar

summerfrog2 summerfrog2
1 cup poured --message at Fri Mar 25 10:47:26 EDT 2011
25 Mar

summerfrog2 summerfrog2
Coffee is ready! --message at Fri Mar 25 10:46:03 EDT 2011
25 Mar

@summerfrog2
summerfrog2
Coffee is ready! --message at Fri Mar 25 10:52:46 EDT 2011
25 Mar via LabTesting

summerfrog2 summerfrog2
1 cup poured --message at Fri Mar 25 10:59:13 EDT 2011
25 Mar

Figure 7.2: Example of the Twitter-based coffee notifications by the summer intern student.

The notification client was also written in Java, and would connect to the Distributor and World Model and watch for changes to appropriate Attribute values. The client also provided a web interface where users could “sign up” for email or SMS notifications, and a Twitter account was created which would host all of the alerts. The student independently determined how to send emails from the client, using a Java email library, how to update a Twitter account programmatically (Figure 7.2), and even implemented a Captcha anti-spam validation form. Once again, over a period of 2–3 weeks, the undergraduate intern was able to create a novel application using the Owl Platform to source data about office activities.

The third project addressed a problem which was very personally applicable to the student. During these summer internships, a large number of incoming students and interns results in overcrowding in the laboratory space. The result of a shortage of work spaces is that chairs often go “missing” when someone leaves their desk for a meeting or lunch. The student decided to write a “stolen chair detector” which would notify a user whenever a chair registered to her was moved from its expected location. The solver reused data from the localization and mobility solvers to determine where a chair was located once it had a Pipsqueak sensor attached to it. The Solver updated a value in the Distributor which would be used by the notification solver to alert the “owner” whenever it moved a specified distance from its “resting” position.

The next Solver written by the student would detect when the daily “tea time” was happening in the lab, a social event where students and faculty would gather in the office kitchen for refreshments and conversation. On a typical day, several members of the lab would gather between 3:00 PM and

5:00 PM and brew fresh coffee and tea. One particular student was in attendance to most of these events, and often went around the lab to encourage others to attend. The Solver was written such that each day between 3:00 PM and 5:00 PM if at least 3 coffee mugs were located in the kitchen and then coffee was brewed, a tea time notification would be generated if it had not happened previously that day. Notifications would be sent by the notification solver to users that signed-up previously to receive them.

The final Solver written by the student utilized two different external services to retrieve traffic problems for a specific geographic area. The Solver would request a City and State (or ZIP code) from the user, translate that to a set of GPS coordinates using an external service, and then periodically poll a second external service for traffic updates using the GPS coordinates determined earlier. The resulting traffic events would then be updated in the Distributor where they could either be used to generate notifications to users or as inputs to other solvers. Although it was never implemented, there was some early interest to use this information with a route-planning service to alert users when to leave the lab in order to arrive at scheduled appointments on time.

Over fewer than 12 weeks, an undergraduate student was able to independently design, program, and test 5 different components using the Owl Platform as a framework to manage the data. At the end of his internship, he discussed his experiences with the Owl Platform research team. Partly as a result of his feedback, the Distributor and World Server components were merged to create the World Model. After the student left his internship at the end of the summer period, the notification client was used for several months by the office staff and students to receive notifications of fresh coffee and tea time gatherings.

7.2.2 Student 2

The second student was an undergraduate student working for an honors program as part of his graduate requirement. He worked closely with a computer science department staff member who was familiar with the Owl Platform system in general terms, but who did not have notable experience programming or configuring it. The student developed two separate applications on top of the Owl Platform and also using the Pipsqueak sensors: an open window notification service and an inventory tracker for a campus bike-sharing program [8]. The student estimated that each project took approximately 20-40 hours to complete (2–4 weeks).

The first project was a notification service that would alert faculty members if they left their office window open and the weather report indicated low temperatures over the next 1–2 days. This is actually frequent problem, particularly around school vacation periods, and has resulted in frozen pipes and significant damage to university and personal property. The project started by attaching Pipsqueak sensors equipped with magnetic contact switches, like those used in home security systems, to the windows of several offices in the building. An Owl Platform instance was already operating in the building to support research tasks focused around device localization and Wi-Fi traffic analysis. The new sensors' data was forwarded to the existing Owl Platform system, and the student configured his project to use these components. Once the sensor data arrived at the Aggregator, the student's Solver would monitor the window status to track the open and closed status of the window, adding this information to the World Model representation of the window. A second program, a notification application that the student wrote, periodically contacted an external weather service to determine the upcoming weather conditions. If the forecast indicated low (near-freezing) temperatures and a window was open, the notification app would access an external directory to identify the office occupant and send an email reminder to close the window.

The second project was a bicycle inventory tracking application for an on-campus bike-sharing program. The purpose of the application was to provide the bike program's staff with real-time and historical information about the number of bicycles available at specific rental locations. Pipsqueak sensors were attached to bicycles and a Pipsqueak receiver was placed near the area where the bicycles are stored when not in use. The student wrote a solver that would monitor the presence of RSSI data from each of the sensors, and use absent RSSI values as a proxy for the presence of the bicycle in the storage area. The World Model was updated with the quantity of bicycles available as they left or arrived in the storage area. The student also built a graphical tool that allowed the bike program staff to view a graph of bike inventory over time.

As part of the student's honors program requirements, he wrote a paper about his project and experience working with the Owl Platform. Overall, he described the ease with which programs could be written for someone with limited programming experience. He also noted how the change from Distributor/World Server to World Model architecture caused additional changes in his projects, but also eased development afterward by introducing simpler application programming interface (API) tools. His greatest criticism was on the lack of complete documentation for the project, a frequent failing of research projects, and the additional time he needed to understand some features of the interfaces to successfully complete his projects.

7.2.3 Student 3

The third student ported the World Model from the SQLite storage engine to MongoDB, a key-value document storage engine. Over a period of 5 months, the student worked to design, develop, test, and document the new storage engine and its performance compared to the SQLite version in use at that time [10].

Evaluated Task	**SQLite Time**	**MongoDB Time**
Insert 900,000 Attribute values	1,899 s	380 s
Snapshot	589–636 ms	8–89 ms
Range Request (400 Attributes)	29,798 ms	107 ms
Range Request (4,500 Attributes)	34,424 ms	415 ms
Range Request (44,000 Attributes)	37,285 ms	1,300 ms

Table 7.1: Comparison of SQLite and MongoDB World Model implementations

The first task the student completed was understanding the source code of the SQLite World Model, and how to configure and interact with a MongoDB server installation using the C++ development library. Over a period of 2-3 months, the student completed a functionally-equivalent World Model server that used a single MongoDB server as the back-end storage engine. Interestingly, the student originally attempted to write the entire World Model "from scratch" until he realized that a significant portion of the functionality was separate from the storage engine, and instead decided to re-use the existing code replace the SQLite code with a compatible interface that utilized MongoDB. The student then compared the performance of the two implementations using a set of repeatable experiments and documented the results, reproduced in Table 7.1. Overall, it documents that MongoDB is a strong candidate for use as a World Model storage engine, outperforming the then-current SQLite version, and likely the MySQL version written later.

The student also made a number of observations both about the SQLite World Model and the ability for MongoDB to function as an effective storage engine. One of the most important discoveries was that MongoDB periodically locks access to the persistent (on-disk) storage device to write all changes to disk. Unlike a traditional database server, it does not perform this action when a client requests it and instead performs a set of batched operations. The result is the potential to lose the previous minute worth of data in the event of a server or operating system failure. The student also noted how much longer designing and developing such a complex system actually took compared to his initial estimates.

7.2.4 Student 4

The fourth student developed two Solvers that use the Microsoft Kinect®camera to capture user motion and determine whether the person had fallen down or had taken a drink from a glass. This project was motivated by a research faculty member pursuing a larger research project for elder care and aging-in-place techinques. The student was provided a programming environment set-up for the camera as well as the requisite hardware. Over a period of about 2 months, the student learned to use the camera and included software development kit (SDK) as well as the Owl Platform libraries and architecture, build a Solver that monitored a user in view of the camera, and completed development of two Solvers [31].

In order to integrate the camera's output into the Owl Platform, the student first wrote a simple program that would simply update an on-demand data type in the World Model with the current data from the camera. Generally, this resulted in around 15 updates per second, or whatever the output rate of the camera was configured to do.

The fall detection Solver monitors the position of the subject's head using the "skeletal" vertices provided by the camera software. If the position of the head drops significantly over approximately one second, it determines that a fall has taken place. After this detection, if the subject does not rise within a few seconds the Solver updates the World Model to indicate that the user has fallen. The student noted that this approach suffered from a relatively high false positive rate, particularly if the subject reached down to pick something up.

The drinking solver analyzed the position of the subject's hands and head and tracks the relative distances between them. If the distance between the head and one or both hands decreases below a threshold for a short period of time, the Solver determined that the user had taken a drink from a glass. The approach was particularly poor in determining that the user had actually drunk something, and had not simply placed her hands near her mouth (e.g., eating, smoking, yawning). The student suggested, but did not implement, the integration of mobility data from a cup or other objects that might be used to supplement the results of the camera-based Solver.

The student demonstrated that the Owl Platform could be used for non-traditional sensor data, in this case the complex output from a high-speed camera, and that this data could be integrated into an existing environment with other data. He was able to successfully write a program to integrate the data from the camera as well as two independent Solvers that could be improved to support elder care efforts in the future.

7.2.5 Students 5 and 6

Over 6 weeks, two summer undergraduate intern students worked to develop both a standing-water sensor and an application to shut-off appliances when occupants leave a room [37]. The standing water sensor was a hardware and firmware modification to the Pipsqueak wireless sensor, and was aided by research faculty to ensure proper functionality. The power-control application was developed entirely by the students.

The standing-water sensor was a modification to the Pipsqueak wireless sensor that was motivated by the concept of "smart" spaces which assist people by automating the monitoring and control of their living and working areas. The sensor modification involved addition of sensor pins and an appropriate capacitor to ensure that detection would be successful in anything except distilled water, which does not readily conduct electricity. Initial modification of the sensor firmware was completed by the students, and minor changes were made by research faculty to ensure that the sensor remained low-energy and would operate error-free. The students then updated an existing Solver to support decoding the recently-added standing water sensor status values and add them to the World Model. The sensor is still actively used by the research group to support "smart" buildings and elder care research.

The power-control project was motivated by a faculty member who frequently left a portable space heater operating after they left their office. The unattended heater wasted electricity and posed a safety hazard, and the faculty member wished to have an automatic way for it to turn off if left on accidentally. The students were provided with an IP-enabled power switch, and independently learned to check its status and control the operation of outlets. They also developed a Solver that would combine data from two Pipsqueak sensors, one attached to the office door and one attached to the office chair, with the power switch. The solver would monitor the chair and door sensors, and if the chair was empty and the door was closed for 30 minutes or more, the power switch would automatically disconnect power to the outlet configured for the space heater.

Overall, the students were able to modify a hardware sensor, integrate it into the Owl Platform, and develop a separate application using a third-party power switch in fewer than 2 months. They once again helped to demonstrate not only the versatility provided by the World Model for managing sensor data, but were also able to quickly understand the core concepts of Owl Platform and build functional applications with very little assistance from the research team.

7.3 Commercial Use

Owl Platform is currently used as one of the fundamental technologies behind a commercial laboratory animal care product [18]. After several months as a prototype trial system, it is currently assisting faculty and staff by providing continuous monitoring and alerts based on environmental conditions in multiple animal facilities. In addition to new application-specific functionality, its use as a commercial product has also driven improvements to overall system design, including network protocols, the World Model, and Aggregator.

Sensors are deployed throughout many facilities with low-power base stations forwarding data back to an off-site Aggregator. Solvers decode the sensor data, generate statistics and provide immediate access to environmental variables spanning nearly a year. Staff and faculty at the facility have expressed great enthusiasm for what they view as a system that deploys rapidly, is easy to access, and provides low-cost, flexible monitoring.

The application centers around a web-based user interface where staff and faculty can see real-time alerts and status, analyze data trends across hours or months, and generate reports for ensuring compliance with regulations and best practices. Supporting the web interface are real-time, customizable alerts which keep facility operators aware of any adverse conditions. Many of these features were first developed in the laboratory either as student projects, or as proof-of-concept implementations to demonstrate the power of a general-purpose framework for WSN data.

Chapter 8
Conclusions

Wireless Sensor Networks are complex systems that are only beginning to be utilized outside of the research laboratory. As the availability of wireless sensors increases, costs decrease, and they become a ubiquitous part of an increasingly connected world, the need to manage and manipulate their data will only grow. It is very likely that most of these networks will be deployed for a specific purpose including environmental monitoring, building security, personal health and wellness. The systems that integrate these networks' data, however, need not be so application-specific. Owl Platform has demonstrated that a general-purpose system that trades the convenience of built-in functionality for the flexibility of a system that can grow and evolve over time. The basic components of Owl Platform have also demonstrated that they can perform effectively outside of the constraints of the research laboratory.

From the start, Owl Platform was designed to support Wireless Sensor Networks on the scale of a large University campus. Experimental results have shown that it is capable of accepting thousands of data Samples per second, analogous to hundreds of thousands of sensors. Given the explosive recent growth in both wireless networked devices and sensors, it is necessary that any systems designed today permit WSNs to grow by several orders of magnitude. Systems dedicated solely to a single building or room are giving way to smart power grids, smart cities, and heterogeneous collections of sensors producing overwhelming amounts of data. Modern automobiles contain hundreds of onboard sensors from the windshield wipers to the tires. Ensuring high-speed, low-latency access to this data on a large scale can enable safer roads, more efficient cities, and more productive communities. Although there is plenty of room for improvement, this prototype of Owl Platform has demonstrated its ability to handle large amounts of sensor data with capacity for additional expansion and growth.

This same explosive growth in the kind and quantity of sensors also means that the data management systems need to remain flexible as well as fast. Not simply focused on processing raw data directly from the sensors, the variety of Solvers and Applications developed so far hint at a system that is generic and adaptable enough to adapt to the needs of its users. The World Model exposes APIs that translate well to service like localization, mobility detection, building and environmental monitoring, and the creation of smart spaces. Integration with external services like weather and traffic reports produces richer applications than those that are restricted to sensor data.

With easy to understand libraries and interfaces, it allows even novice programmers to start developing cutting-edge applications in just a few weeks. Encouraging analysis by Solvers to focus on specialized, dedicated tasks ensures that the complexity of the system lies in the network of interactions between the components, not in the individual processes. Building dynamic, complex systems out of simple, manageable components allows better maintenance, fewer mistakes, and more efficient execution.

A multi-layered, modular system provides the opportunity to improve individual components without significant impact on others. As long as components adhere to the contracts required of their interface bindings, implementations can change and improve incrementally with minimal side effects. As demonstrated by the World Model evaluations, improvements in the underlying systems are necessary for the system to support the scales desired for long periods of time. These changes remain invisible to the other components, ensuring that the costs of such improvements remain.

8.1 Future Work

Although Owl Platform has demonstrated that is has great potential, it remains a work in progress. During the evaluation of the Aggregator and World Model, a number of flaws and limitations have become visible that need to be addressed. Some of the needed improvements are readily identifiable and easy to achieve while others require careful consideration and further research.

Evaluation of the Aggregator showed that its performance is currently limited by the cost of evaluating Subscription Rules for each Sample. Anecdotal evidence indicates that caching the results may alleviate some of these costs, but there is a limit to what caching can achieve. Careful thought needs to be given to how best to reduce the number of instructions executed for each Sample. For example, the current version relies on the Java Virtual Machine for garbage collection and memory management. By reducing the number of objects needing garbage collection (via object pooling and reuse), the processor time and memory spent can be reduced and reallocated to other tasks. This is a common optimization for Java-based appliations, and should be achievable within a couple of weeks by an experienced programmer.

The choice of programming language may also be a severely limiting factor for the Aggregator. Porting part or all of the code into a language like C++ may also improve efficiency by exploiting the hosts's hardware capabilities. For instance, Transmitter ID filtering could be offloaded to a Graphical Processing Unit (GPU). The rule checking and filtering can be modeled as a series of data transforms and mapped to modules in the GPU. A change like this requires significant planning and careful design to be effective. By working together with a researchers specializing in GPU-based computation, it may be possible to achieve significant improvements in performance over weeks or months. A hybrid approach might be warranted as well, with the bulk of the Aggregator logic implemented in a portable language like Java to make maintenance and development easier. Performance-critical sections can be modularized with optimized versions written in C++ for GPUs and "fallback" versions available in Java for when appropriate drivers or supported hardware aren't available.

Aside from need for raw speed in the Aggregator, there is strong motivation for richer feedback to the WSNs producing data. Even if vast improvements are made in efficiency, the Aggregator may still reach a point where it can no longer handle the high volume of Samples being received. Much like TCP's flow control mechanism, the Aggregator should inform the WSN gateway when it is overwhelmed with data. What form this notification takes, however, is not immediately clear. A simple approach could have the Aggregator send a value between zero and one that indicates its estimation of the current capacity. When it can no longer keep up with arriving data, it could further indicate the fraction of Samples that it is shedding in order to continue operating.

More complex feedback could include information about what Transmitters the Solvers are requesting, and at what inter-arrival rate. WSNs could then make better decisions about how to send what data, but it also begins to weaken the isolation that is created by a layered approach. This weakening may be worthwhile as a compromise, and the different trade-offs of various approaches need to be studied carefully.

It also appears that while the initial design of the Subscription Request Rules is sufficient for many applications, there are situations when filtering based on Receiver IDs are very beneficial. For instance, the student that implemented the bike rental inventory system relied on proximity to a specific Receiver to know when a bicycle was returned to the storage area. If the Aggregator were receiving data from hundreds of Receivers, and only a handful are important to a Solver, then it stands to reason that the Aggregator should not forward data from the other Receivers. Not only does it place additional burden on the Aggregator and Solver, but it also generates unnecessary network traffic.

The World Model, too, is not without its need for improvement. One of the first observations made while evaluating the network protocols for the World Model was that the lack of acknowledgement (ACK) messages for Updates, Expirations, and Deletes meant that there was no way to ensure that an operation had completed. One of the reasons it was omitted from the original protocols, was because there were almost no guarantees about the properties of operations made by the World

Model. It was originally envisioned as a “best effort” service, but that is no longer sufficient to support the types of applications desired. The changes to be made are somewhat small, but would not be backward compatible. Any Solvers or Applications using older versions of the protocol (or libraries) would be required to upgrade in order to communicate with a World Model using the improved protocol. Alternatively, it may be possible for the World Model to support both protocols, but this introduces greater costs for maintenance any future changes.

The possibility for additional features in the World Model must also be considered. In particular, the ability to delete a subset of Attributes values rather than an entire history, Range Requests with time granularity, access control, and data secrecy and authenticity guarantees.

At present, the World Model is only capable of Deleting all values for a particular Attribute. The ability to specify exactly which values get deleted, either based on time ranges or the value itself, would allow finer-grained control for system administration. This would eliminate the need to access the storage engine directly when these functions are required.

One shortcoming noted when developing web-based applications using the World Model, is that Range Requests are only able to limit the size of returned data sets based on time. In most cases, a requesting Application may have no idea how much data is contained in the World Model between two timestamp values. If too much data is returned, the requestor may become overwhelmed and unable to process all of the data efficiently. If too little, the requestor made need to make multiple requests, wasting time and network resources. The ability to limit Range Requests either by size (in bytes) or by number of entries (Attribute values) could provide significant benefit to systems resources need to be carefully allocated. This is another significant change to the World Model interface, and should be evaluated thoroughly before any change is made.

Access control is a topic that has been discussed by the developers and researchers since the original implementation was completed. Currently, the only access control in the World Model is based on the Origin value, which can easily be forged. A combination of authentication and authorization mechanisms need to be developed to ensure that administrative policies for data access and manipulation can be enforced. Initial discussions have proposed using a traditional user-role-privilege model. User accounts are authenticated at the beginning of a connection. Each user is assigned one or more roles, and each role is assigned one or more privileges. This is a fairly common system design, and it may work well for the World Model. However, careful consideration is needed before choices are made, since adding any type of access control mechanism must increase the complexity of the interaction with the World Model.

In line with access control is the need for data integrity and secrecy, both which can be provided by public key cryptographic systems. The Origin field of Attribute values can serve a dual purpose in this case. Data can be signed by a private key, with the corresponding public key used as the Origin value. Likewise, data can be encrypted with a public key and only decrypted with the corresponding private key. This would allow, for instance, a Solver to generate data and encrypt it in such a way that only an authorized key or set of keys could decrypt it. Because Client requests are made by searching Identifiers and Attribute Names, and not the values, clients would be able to retrieve the data and decrypt it outside of the World Model. The problem of key distribution and signing remain an open problem, but these are problems encountered by any system using public key cryptography. A separate entry for public key signatures and certificates would allow clients to validate data signatures in a similar manner to how web browsers validate connections to HTTP servers.

A greater understanding of the World Model's performance under different workloads also needs to be performed. The basic set of experiments described in this thesis are a starting point, but do not reflect realistic workloads. In particular, the impact of large numbers of simultaneous connections needs to be carefully determined. Because Owl Platform relies so much on the flow of data between Solvers and the World Model, even the slightest inefficiencies in this relationship can be severely limit efficiency when scaled to dozens or hundreds simultaneous connections.

Along the same lines, an exploration of the possibility of exploiting the distributed features of MySQL and MongoDB may provide significant benefits for the World Model. Both systems are capable of replication and parallelization, particularly when it comes to the read-only access of the World Model Client interface. This is another significant undertaking, likely requiring months of effort to design, develop, and evaluate. However, the potential for the World Model to support even greater numbers of connections and data rates makes such an investigation worthwhile.

Across all layers of the system, transport layer security is a critical feature that is currently absent. It is now general knowledge the national governments are capable of intercepting and recording vast amounts of data that transit public networks. In many cases sensor data can be inherently private, and reasonable steps should be taken to ensure it is not intercepted by unauthorized parties. The incorporation of end-to-end encryption has been a planned feature from the start, but has never been a high priority because the focus has been primarily on research projects. The recent transition to a commercial product, however, should motivate the addition of some type of encryption for the data connections, even if features like access control and data authentication are postponed. Although it would require modifying all of the existing protocol libraries, and both the Aggregator and World Modmel, it would not necessarily require changes to Applications and Solvers. Much like HTTP servers that provide both secure and non-secure interfaces, these components could also provide encrypted and unencrypted interfaces. With the adoption of access control techniques, policy decisions could be enforced which restrict what data or operations can be performed on unencrypted connections.

References

[1] Mike Addlesee, Rupert Curwen, Steve Hodges, Joe Newman, Pete Steggles, Andy Ward, and Andy Hopper. Implementing a sentient computing system. *Computer*, 34(8):50–56, 2001.

[2] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.

[3] Ian F Akyildiz, Xudong Wang, and Weilin Wang. Wireless mesh networks: a survey. *Computer networks*, 47(4):445–487, 2005.

[4] Ashwin Ashok, Chenren Xu, Tam Vu, Marco Gruteser, Richard Howard, Yanyong Zhang, Narayan Mandayam, Wenjia Yuan, and Kristin Dana. Bifocus: using radio-optical beacons for an augmented reality search application. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 507–508. ACM, 2013.

[5] Zhao Cao, Charles Sutton, Yanlei Diao, and Prashant Shenoy. Distributed inference and query processing for rfid tracking and monitoring. *Proceedings of the VLDB Endowment*, 4(5):326–337, 2011.

[6] Yingying Chen, Gayathri Chandrasekaran, Eiman Elnahrawy, John-Austen Francisco, Konstantinos Kleisouris, Xiaoyan Li, Richard P Martin, Robert S Moore, and Begumhan Turgut. Grail: A general purpose localization system. *Sensor Review*, 28(2):115–124, 2008.

[7] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[8] Jonathan Chiu. "grail" by jonathan chiu and prof. richard martin. `http://www.ece.rutgers.edu/node/894`, 2012. Accessed: 2014-08-30.

[9] Colin Dixon, Ratul Mahajan, Sharad Agarwal, AJ Bernheim Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *NSDI*, volume 12, pages 337–352, 2012.

[10] John Esmet. Mongodb world model. `https://git.owlplatform.com/wiki/index.php/MongoDB_World_Model`, 2012. Accessed: 2014-08-30.

[11] Bernhard Firner, Shweta Medhekar, Yanyong Zhang, Richard Howard, Wade Trappe, Peter Wolniansky, and Eitan Fenson. Pip tags: Hardware design and power optimization. In *The 5th Workshop on Embedded Networked Sensors*, 2008.

[12] Bernhard Firner, Robert S Moore, Richard Howard, Richard P Martin, and Yanyong Zhang. Poster: Smart buildings, sensor networks, and the internet of things. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 337–338. ACM, 2011.

[13] Bernhard Firner, Chenren Xu, Richard Howard, and Yanyong Zhang. Multiple receiver strategies for minimizing packet loss in dense sensor networks. In *Proceedings of the eleventh ACM international symposium on Mobile ad hoc networking and computing*, pages 211–220. ACM, 2010.

[14] The Apache Software Foundation. Apache commons pool. `http://commons.apache.org/proper/commons-pool/`, 2014. Accessed: 2014-09-01.

[15] Jonathan Gottfried. Diy home automation using twilio, powerswitch, arduino, and pusher. `https://www.twilio.com/blog/2012/08/diy-home-automation-using-twilio-powerswitch-arduino-and-pusher.html`, 2012. Accessed: 2014-07-18.

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

[17] Berk Gurakan, Omur Ozel, Jing Yang, and Sennur Ulukus. Energy cooperation in energy harvesting wireless communications. In *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*, pages 965–969. IEEE, 2012.

[18] InPoint Systems Inc. Inpoint systems technology. `http://www.inpointsys.com/new-technology.html`, 2014. Accessed: 2014-08-30.

[19] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67. ACM, 2000.

[20] Joseph M Kahn, Randy H Katz, and Kristofer SJ Pister. Next century challenges: mobile networking for smart dust. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278. ACM, 1999.

[21] Aman Kansal. Senseweb: Wikipedia of sensors, 11 2008. Invited Keynote, Collaborative R&D for Distributed Sensors Workshop.

[22] Konstantinos Kleisouris and Richard P Martin. Parallel algorithms for bayesian indoor positioning systems. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, pages 15–15. IEEE, 2007.

[23] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[24] ADT LLC. Adt pulse interactive solutions web portal user guide. `http://www.adtpulse.com/portal/user-guides/Advanced/Content/WebPortal/Printed_Documentation/Web_Portal_User_Guide_2.pdf`, 2013. Accessed: 2014-07-18.

[25] Verizon Online LLC. A tour of verizon home monitoring and control. `http://www.verizon.com/cs/groups/public/documents/adacct/a_tour_of_hmc051212.pdf`, 2012. Accessed: 2014-07-18.

[26] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[27] David Madigan, E Einahrawy, Richard P Martin, W-H Ju, Parameshwaran Krishnan, and AS Krishnakumar. Bayesian indoor positioning systems. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 2, pages 1217–1227. IEEE, 2005.

[28] Robert S. Moore. Source code: World model profiler. `https://github.com/romoore/wm-profiler/settings`, 2014. Accessed: 2014-08-28.

[29] Robert S Moore, Bernhard Firner, Chenren Xu, Richard Howard, Richard P Martin, and Yanyong Zhang. It's tea time: do you know where your mug is? In *Proceedings of the 5th ACM workshop on HotPlanet*, pages 63–68. ACM, 2013.

[30] Robert S Moore, Richard Howard, Pavel Kuksa, and Richard P Martin. A geometric approach to device-free motion localization using signal strength. Technical report, Technical Report, Rutgers University, 2010.

[31] Xiaokang Qin. Kinect project. `https://git.owlplatform.com/wiki/index.php/Kinect_Project`, 2012. Accessed: 2014-08-30.

[32] Eleanor Randolph. Transplanting madison square garden. `http://takingnote.blogs.nytimes.com/2013/03/28/transplanting-madison-square-garden/`, 3 2013. Accessed: 2014-09-01.

[33] Lenin Ravindranath, Venkata N Padmanabhan, and Piyush Agrawal. Sixthsense: Rfid-based enterprise intelligence. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 253–266. ACM, 2008.

[34] Kay Romer and Friedemann Mattern. The design space of wireless sensor networks. *Wireless Communications, IEEE*, 11(6):54–61, 2004.

[35] Ali Salehi. *Design and implementation of an efficient data stream processing system.* PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2010.

[36] EYE Vista Hotel Service. Hotel izmaylovo ***, general information. `http://www.hotelizmailovo.ru/eng/comm.html`, 2013. Accessed: 2014-09-01.

[37] Silas Waltzer and Sai Kotikalapudi. Owl platform for localization and sensing. `http://summer.winlab.rutgers.edu/projects/wiki/2012/Projects/OwlPlatform`, 2012. Accessed: 2014-08-30.

[38] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. The active badge location system. *ACM Transactions on Information Systems (TOIS)*, 10(1):91–102, 1992.

[39] Alec Woo, Siddharth Seth, Tim Olson, Jie Liu, and Feng Zhao. A spreadsheet approach to programming and managing sensor networks. In *Proceedings of the 5th international conference on Information processing in sensor networks*, pages 424–431. ACM, 2006.

[40] Chenren Xu, Bernhard Firner, Robert S Moore, Yanyong Zhang, Wade Trappe, Richard Howard, Feixiong Zhang, and Ning An. Scpl: Indoor device-free multi-subject counting and localization using radio signal strength. In *Proceedings of the 12th international conference on Information Processing in Sensor Networks*, pages 79–90. ACM, 2013.

[41] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Computer networks*, 52(12):2292–2330, 2008.