

**Aqueduct: Robust and Efficient Code Propagation  
in Heterogeneous Wireless Sensor Networks**

by

**Lane A. Phillips**

B.S., University of Nebraska, 2000

A thesis submitted to the  
Faculty of the Graduate School of the  
University of Colorado in partial fulfillment  
of the requirements for the degree of  
Master of Science  
Department of Computer Science

2005

This thesis entitled:  
Aqueduct: Robust and Efficient Code Propagation in Heterogeneous Wireless Sensor  
Networks  
written by Lane A. Phillips  
has been approved for the Department of Computer Science

---

Richard Han

---

John Bennett

---

Shivakant Mishra

Date \_\_\_\_\_

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Phillips, Lane A. (M.S., Computer Science)

Aqueduct: Robust and Efficient Code Propagation in Heterogeneous Wireless Sensor Networks

Thesis directed by Assistant Professor Richard Han

Dynamic reprogramming of sensor applications and sensor operating systems is emerging as a critical function required by in situ wireless sensor networks. This ability to remotely patch or upgrade software in deployed sensor nodes via the wireless network is complicated by the increasing trend towards heterogeneity in WSN hardware platforms, sensor operating systems, and role-based differentiation, e.g. between aggregators and leaf sensor nodes. Current protocols such as Deluge focus on propagating the same code image to a network of homogeneous sensor nodes. Naive approaches to adapt such protocols for heterogeneity are largely inefficient. This thesis presents Aqueduct, a robust and efficient protocol for propagating dynamic code updates through a heterogeneous WSN.

## Acknowledgements

Richard Han and Anmol Sheth provided suggestions and critiques for my protocol designs over many weekly discussions. Anmol Sheth helped with researching related work and analyzing data. David Lyle and James Carlson were members of my project team when we investigated Deluge for CSCI 5573: Advanced Operating Systems.

## Contents

<b>Chapter</b>	
<b>1</b>	Introduction . . . . . 1
<b>2</b>	Related Work . . . . . 9
2.1	Data Reduction Techniques . . . . . 9
2.2	Dissemination Protocols . . . . . 10
<b>3</b>	Aqueduct Architecture . . . . . 15
3.1	Deluge and Heterogeneity . . . . . 15
3.1.1	Deluge Design . . . . . 15
3.1.2	Adapting Deluge For Heterogeneity - A First Attempt . . . . . 17
3.2	Building Aqueducts . . . . . 18
3.3	Aqueduct State Machine . . . . . 21
3.3.1	MAINTAIN State . . . . . 23
3.3.2	RX State . . . . . 25
3.3.3	TX State . . . . . 26
3.3.4	FORWARD State . . . . . 27
3.4	Further Enhancements . . . . . 28
3.4.1	Link Symmetry . . . . . 28
3.4.2	Caching . . . . . 30

<b>4</b>	<b>Protocol Evaluation</b>	<b>32</b>
4.1	Aqueduct and Deluge Performance . . . . .	33
4.2	Caching . . . . .	39
4.3	Control Overhead . . . . .	41
<b>5</b>	<b>Future Work</b>	<b>46</b>
<b>6</b>	<b>Conclusions</b>	<b>50</b>
	<b>Bibliography</b>	<b>51</b>

## Tables

### Table

4.1	Control Overhead . . . . .	45
-----	----------------------------	----

## Figures

### Figure

1.1	Heterogeneous WSN . . . . .	7
3.1	Forwarding in Aqueduct . . . . .	20
3.2	Aqueduct State Machine . . . . .	22
3.3	Link Symmetry and Caching . . . . .	29
4.1	Testbed Schematic . . . . .	34
4.2	Testbed Photograph . . . . .	34
4.3	Total Data Transmitted . . . . .	36
4.4	Total Requests Transmitted . . . . .	37
4.5	Projection of Aqueduct Requests . . . . .	37
4.6	Projection of Deluge Requests . . . . .	38
4.7	Completion Time . . . . .	40
4.8	Cache Usage . . . . .	42
4.9	Effect of Caching on Programming Time . . . . .	43
4.10	Effect of Caching on Data Transmission . . . . .	44

## Chapter 1

### Introduction

Recent research in wireless sensor networks (WSNs) has highlighted the importance of supporting the capability for remote reprogramming of **in situ** sensor nodes via the multi-hop wireless network [25, 14]. Manually reprogramming hundreds or thousands of deployed sensor nodes, e.g. by visiting each node in the field and attaching the node to a laptop or PDA in order to reprogram the sensor node's software, is a time-consuming and labor-intensive task that may also be simply infeasible in various scenarios, e.g. nodes may settle in areas that are inaccessible to deployers. Motelab enables remote reprogramming of sensor nodes via a wired backplane infrastructure [35]. Such a wired infrastructure will not likely be available in many in situ WSN scenarios. However, this principle of remote network reprogramming can be extended to wireless multi-hop networks. Remotely propagating software updates via the wireless network is likely to be an especially effective approach for managing large scale distributed outdoor WSNs. The emphasis of this thesis is on developing a novel code propagation protocol called **Aqueduct** for robust and efficient networked reprogramming in heterogeneous WSNs. A key benefit is that Aqueduct is designed for practical use, and is tested for robustness and efficiency in a deployed WSN in the presence of highly variable wireless communication links.

The need to update software on deployed sensor nodes is driven by a variety of factors. Reprogramming via the wireless network enables new capabilities to be added

to the locally executing binary image after the network is deployed, e.g. a new application, an improved driver for calibrating an on-board sensor, a more robust routing algorithm, a more energy-efficient MAC layer implementation, or a more accurate time synchronization algorithm. Network reprogramming provides a convenient tool for iterating the design of software in a WSN deployment to accommodate unanticipated requirements or new features that will be needed as experience is gathered in a particular WSN deployment. Such reprogramming is also helpful to patch errors in the software after deployment, e.g. a bug is found in the deployed sensor operating system (OS) or a logical error is found in the WSN routing that results in undesirable looping. In all these cases, the software present on in situ sensor nodes will need to be updated.

Network reprogramming consists of two main components: the installation mechanism for loading the code on board a sensor node; and the code propagation mechanism, e.g. networking protocol, for disseminating software to a sensor node. First, the installation mechanism is responsible for loading the code update into RAM once it has arrived at a sensor node. This process may also involve rewriting the internal flash and resetting the microcontroller. The installation mechanism must also account for the structure of the code to be loaded. This structure differs depending on whether the operating system supports only statically linked complete images [13], dynamic modules [8, 10], or scripting languages for virtual machines [22]. In addition, the installation component may employ a compression mechanism, e.g. an rsync-like utility employing differential compression, in order to reduce the size of the software image update [17, 26].

The second major component of network reprogramming is the code propagation mechanism, which is responsible for efficiently and reliably forwarding software updates to one or more sensor nodes. Examples of such code propagation mechanisms include Deluge [14] and MOAP [31], which will be described in more detail later. A code propagation mechanism needs to specify its mode of communication, i.e. broadcast, multicast, or unicast. Since code must be reassembled reliably and in order at each

receiver, a code propagation mechanism also needs to provide reliability against lost or corrupt packets. A code propagation protocol also should be designed to achieve energy efficiency and bandwidth efficiency. Finally, a code propagation mechanism needs to achieve robustness in the presence of spatially irregular, time-varying, asymmetric, wireless links that would be encountered in a practically deployed WSN.

The Deluge code propagation protocol incorporates a variety of useful engineering principles into its design, some of which can be attributed to the underlying Trickle routing mechanism that it employs. First, Deluge implements a three phase Advertise-Request-Data polite gossip protocol, in which data is only pushed by a sender upon receiving an explicit request for data from an immediate neighbor. This pull-based approach has the benefit that no more data than is necessary is transmitted. Second, as data is advertised by a sender, neighbors suppress redundant advertisements upon overhearing other identical advertisements. This passive exploitation of the wireless broadcast medium allows the rate of advertisements to stay constant in a region and scales well with increasing density. Third, a sender exponentially increases the periodicity of its advertisements if it fails to hear requests. This adaptive approach minimizes advertisement overhead while the network is quiescent, but still responds quickly when there is new data to propagate. Fourth, Deluge benefits from a soft state design in that, as data is reliably flooded hop by hop throughout the network, the failure and loss of state at a single node does not affect more than the minimum number of nodes downstream. If there are other paths for code to propagate to downstream nodes, then the code will reach downstream, bypassing the failed node. Other state-full routing protocols enforce data propagation along well-defined paths, such that a failure or loss of state denies data to a much larger set of downstream nodes.

Evolving or adapting networking reprogramming protocols like Deluge to account for the emerging trend towards increased heterogeneity in WSNs is the focus of this thesis. Heterogeneity complicates the design of efficient code propagation protocols

because the assumption is no longer valid that a single uniform code image needs to be propagated to all nodes. Considerable energy, bandwidth, and memory can be saved in a resource-limited WSN by propagating a code image to only those nodes that need that image. The underlying trends toward heterogeneity are examined next to provide a context for the ensuing discussion on improving code propagation mechanisms via Aqueduct.

Heterogeneity is increasingly manifested in three ways: the expanding diversity of hardware platforms and microcontrollers; a proliferation of sensor operating systems; and increasing differentiation in roles played by different nodes in a WSN. First, code images may vary because the hardware platforms may differ, e.g. MicaZ motes [1] and Telos nodes [6] use different microcontrollers, that is the Atmel Atmega 128 and TI MSP430 F1611 respectively. Since these two platforms share the same 802.15.4-compliant Chipcon CC2420 radio, then an interoperable heterogeneous WSN can be composed of Telos and MicaZ nodes, assuming that a common network stack is also shared. Reprogramming such a hybrid network of Telos and MicaZ nodes would require that two distinct code images be propagated. For example, TinyOS can be compiled to operate on each microcontroller, resulting in two distinct code images that would need to be propagated. Beyond microcontrollers, hardware and code images may vary because sensor hardware and driver support differs across nodes, e.g. some Mica2 motes may be equipped with GPS sensors only, while others are equipped with temperature and  $CO_2$  sensors but not GPS.

Another observation is the increasing proliferation of sensor operating systems, including TinyOS, Contiki, SOS, and MOS [13, 10, 8, 7], most of which support the Mica2 platform. As long as compatibility is maintained in the network protocol stack and radio, then an interoperable heterogeneous WSN can be composed of Mica2's employing different sensor OSs. Each brand of OS would then require a different code image to be propagated during reprogramming. This is analogous to the Internet, in which multiple

heterogeneous operating systems like Linux, Apple OSX and Microsoft Windows coexist and interoperate to support distributed applications while communicating via common networking protocols.

Finally, role-based differentiation is becoming increasingly important in WSNs. Even if all nodes share the same basic hardware and software platform, e.g. all Mica2 motes run TinyOS, different nodes can have different application roles, resulting in heterogeneous code images. For example, aggregators have been proposed for summarizing and compressing sensor data [19, 36], a role that is different than the collection of data required of embedded leaf sensor nodes. This role-based differentiation is exacerbated when there is hardware variation, e.g. bridging nodes with dual radios to interface between 802.11 and 802.15.4 networks, or a subset of watchdog nodes specially equipped with chemical sensors to initiate target tracking. These roles may also be differentiated based on geography, i.e. the deployer may wish to reprogram only nodes in certain regions of a WSN [33, 34].

Figure 1.1 illustrates the challenge of accounting for heterogeneity in the design of efficient WSN code propagation protocols. The network consists of four groups, where groups A and B consist of similar nodes that are separated from each other by groups C and D, which are comprised of a different set of homogeneous nodes. The goal is to reprogram group A's nodes using a newer code image residing in group B's nodes. An efficient design for a code propagation protocol would only involve intermediate nodes in group D to forward the reprogrammed code from B to A. The nodes in shaded group C need not be involved since they are neither recipients nor serve a forwarding role. The above discussion is a simplification of a more general scenario. These "groups" could be defined by any combination of factors, including geographic regions, application roles, operating systems, microcontrollers, or sensor hardware support. Moreover, nodes in a group need not be confined to homogeneous regions, and can be interspersed sparsely or densely with variable concentration throughout the WSN. The basic premise still holds

that an efficient code propagation protocol should limit the number of nodes involved in forwarding the code updates.

The code propagation approach taken by Deluge is to reliably flood the same code image hop by hop to all nodes. A similar approach was taken by MOAP [31], with the limitation that only complete images can be propagated hop by hop. Since Deluge allows code propagation on a finer granularity of pages of code rather than full code images, then its store-and-forward latency is less than MOAP. As a result, Deluge is the focus of the analysis in the rest of this thesis. Deluge’s reliable flooding mechanism can be naively adapted to account for heterogeneous code images by adding an ID field to Deluge packets that indicates which application or code image is being reprogrammed. The primary drawback of adapting Deluge in this way is that each of the different code images would still be reliably broadcast to **every** node in the WSN. For example, every node in both regions C and D in Figure 1.1 would have to be involved in caching and forwarding all code images. Broadcasting in this manner is clearly quite inefficient in resource-limited WSNs, resulting in wasted energy, bandwidth, and memory throughout the heterogeneous network at nodes that do not need to forward or receive the code update.

Aqueduct’s objective is to achieve efficient yet robust code propagation for a heterogeneous WSN by limiting the number of nodes that need to be involved in propagating code while also tolerating wireless variability in a practical manner. Figure 1.1 illustrates the basic concept of Aqueduct, which seeks to establish “aqueducts” of intermediate nodes that efficiently bridge and propagate code between nodes that desire a code update, e.g. nodes in group A, and nodes that possess the freshest code image, e.g. nodes in group B. Data should propagate along these aqueducts and not elsewhere. Establishment of these aqueducts should flexibly and robustly adapt to changing RF quality experienced in real WSN deployments. Thus, aqueducts are not as rigid nor as crisply defined as the figure makes them appear. Since Deluge’s design incorporates

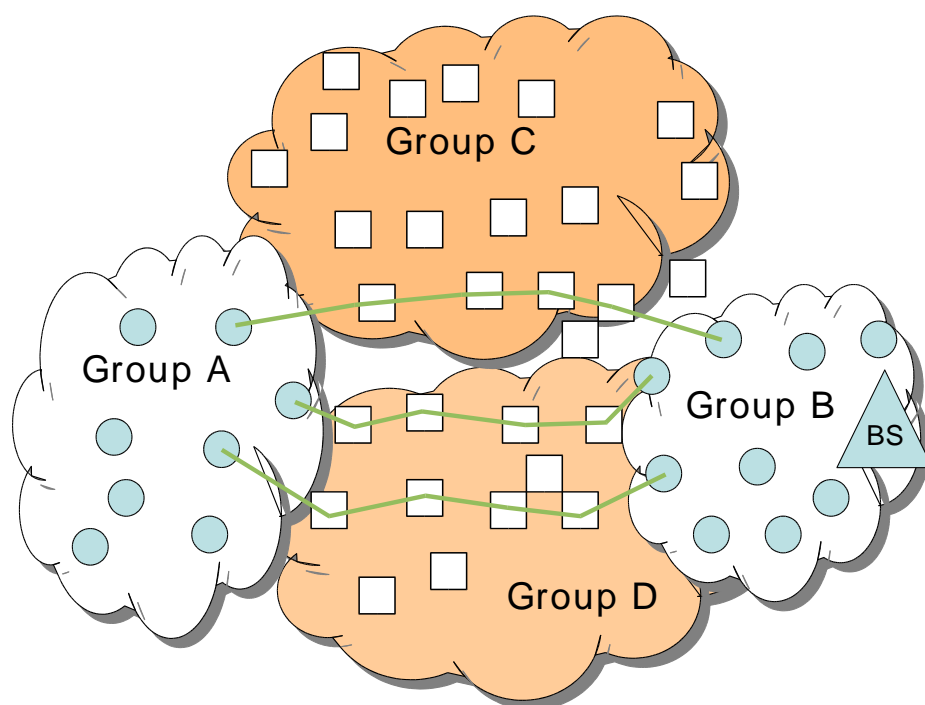


Figure 1.1: Heterogeneous WSN requiring propagation of code for network reprogramming between homogeneous regions B and A separated by heterogeneous regions C and D. Aqueducts act as efficient conduits for propagating code between B and A.

many sound engineering principles like efficient pull-based data dissemination, passive suppression, soft state, and adaptive response, Aqueduct’s approach is to leverage the existing framework provided by Deluge, rather than reinvent the wheel, and incorporate new capabilities that enable robust and efficient code propagation in heterogeneous WSNs. The original contributions of this thesis are the design, implementation and evaluation of Aqueduct, a robust and efficient code propagation protocol for heterogeneous wireless sensor networks. To summarize, Aqueduct adheres to the following high-level design principles:

- (1) dynamic network reprogramming for **heterogeneous** WSNs with diverse hardware, software, and application roles
- (2) efficiency in terms of reduced code overhead by limiting involvement in forwarding
- (3) robustness to spatially irregular, time-varying, asymmetric RF links
- (4) incorporation of new capabilities into the framework provided by Deluge

In the rest of this thesis, Chapter 2 gives a summary of related work. Chapter 3 describes Aqueduct’s design principles, including the distinction between forwarding and member nodes, the subsequent modifications to Deluge’s state machine, and practically robust construction of aqueducts via strong symmetric links. Chapter 4 describes a testbed implementation and evaluation of Aqueduct’s performance in terms of overhead and delay in comparison with Deluge. Chapter 5 contains a discussion of additional features and analyses that we would like to Aqueduct to perform. Chapter 6 concludes the thesis.

## Chapter 2

### Related Work

There are many tools designed for the wired Internet to manage software updates and provide bug fixes. IBM's Tivoli [3], Microsoft's Systems Management Server [5] and HP's OpenView [2] are examples of existing products that are used to propagate updates, manage device configuration and provide centralized application deployment. These tools deal with the complexity of heterogeneous devices in the network, which include mobile devices as well as a diverse set of hardware and software configurations. These tools use IP Multicast techniques to propagate updates to the devices in the network.

#### 2.1 Data Reduction Techniques

Software update tools designed for sensor networks are required to deal with the added complexity of extreme resource constraints. Radio transmission which is the primary source of power consumption [29] should be minimal. One technique of reducing the radio transmissions is **compression**, where the binary image is compressed before being propagated to the network. However, decompression can require a large amount of CPU processing and can be especially difficult on microcontrollers with small word sizes and limited memory, so the savings achieved in data propagation could be overshadowed by the CPU cycles for decompression in a sensor network.

Another technique is **differential patching**, where only the difference between

the new and old image is transmitted. [17] uses a lightweight protocol similar to the Rsync protocol, which compares the program at block level and hence requires no prior knowledge of program code structure or the processor's instruction set. Another approach presented in [27] reduces the difference image size by building the new code image using an edit script of commands that are propagated to the network. The edit script contains commands to copy blocks of memory from one location to another. A list of triples, each containing a begin address, end address, and offset, is used to reconstruct the new image from the old image. However, the above approach is specific to the instruction format of the underlying microcontroller and does not scale when the entire application is modified. Another paper proposes only sending the differences between software versions, conducting the update with an edit script that is generated off of the nodes [26]. Three different methods are detailed for conducting the actual update on the node depending on the node's current needs. The main contributions of this paper are optimizations to minimize the size of the edit script. This solution updates the running image of the software and must correct addressing as it works, which adds a great deal of complexity to the problem.

Use of a high level language which is interpreted by a **virtual machine** is another approach to reduce the amount of data that is transmitted [22]. These instructions resemble common unit operations that are carried out by a typical sensor network application, such as "sense the temperature and transmit it to the base station". Thus updating the application merely requires transmitting a new script to the network. This approach does not support updating the virtual machine or the underlying OS, however.

## 2.2 Dissemination Protocols

Data dissemination protocols in sensor networks have to deal with the complexity of a highly unreliable wireless medium and varying link qualities [11]. Dissemination of the new image to the required nodes in the network should be done efficiently. The

simplest dissemination protocol, although very inefficient, is flooding, where the new image is flooded out to every node in the network. Flooding is a very expensive operation and is not practical for large image sizes. Impalai [24] is a middleware architecture enabling application modularity, adaptivity, and reparability. It provides network re-programming based on a probabilistic broadcast protocol to disseminate the new image to the entire network. Another alternative to flooding is the three-way handshake protocol of **ADV-REQ-DATA** presented by SPIN [12]. In the **ADV** phase each node advertises its version by a local broadcast. An interested node sends a **REQ** for the advertised data which causes the source node to transmit the **DATA**. This algorithm reduces the amount of redundant data transmitted by a simple broadcast flood. Epidemic algorithms further extend this basic protocol to reduce the rate of advertisements.

Epidemic routing algorithms are descendants of flooding algorithms, where various steps are taken to reduce the amount of flooding. Applications of epidemic algorithms on the Internet have been failure detection [28], data aggregation, resource discovery and monitoring [32] and database replication [9]. Epidemic algorithms for wireless network make use of the underlying broadcast medium. Trickle is an epidemic routing protocol based on a polite gossip policy [23]. Trickle borrows concepts from epidemic routing, scalable multicast and wireless broadcast literature. Instead of directly pushing data, a summary or metadata is periodically transmitted to the neighboring nodes and data is transmitted only to the neighbors explicitly requesting the data. Nodes regulate themselves by making efficient use of the underlying broadcast medium. If the same advertisement for data is heard from a neighbor node, then the node suppresses its own transmission. Inconsistencies are resolved by requesting the data heard in the advertisements. The key features of Trickle are low maintenance, rapid propagation and scalability.

A large number of multicast protocols have been designed for wireless sensor net-

works. Trickle adopts its suppression techniques from Scalable Reliable Multicast(SRM) [18] designed for the wired Internet. The suppression techniques of SRM reduce network congestion and request implosion at the server.

MOAP (Multi-hop Over-the-Air Programming) [31] and MNP (Multi-hop Network Reprogramming) [30] use a ripple-like propagation mechanism to propagate the new image to the entire network. At each ripple, a subset of nodes are the source nodes and all the other neighbor nodes are receivers. A publish-subscribe mechanism is used to prevent multiple nodes becoming the source nodes in the same neighborhood. Source nodes publish their newer version and all interested nodes subscribe. MNP differs from MOAP in its technique for detecting dropped packets. MOAP and MNP require the entire image to be reliably received before the next ripple can begin.

The goal of Directed Diffusion [15] is to facilitate efficient communication between multiple sources and sinks in a network. Directed Diffusion consists of multiple phases. In the first phase **Exploratory Interests** are flooded to the entire network. Each node on receiving an interest interprets the interest and matches it with the interests in its cache. If a match is not found, an entry is created for it. This interpreted interest is then forwarded to the rest of the network. In the next phase, once the source matching the interests is identified, it activates its sensors and starts producing data. Initially it sends out **Exploratory Data** events which establishes a multi-path reverse route to the sink. The sink reinforces one of these multiple reverse paths based on metrics like shortest delay, shortest hop etc. Data is then transmitted from the source to the sink only on this reinforced path. Data is also cached on the reinforced path. Caching of data prevents loop formation and suppresses duplicate data. Variants of the above diffusion protocol have been implemented on the TinyOS platform called TinyDiffusion [4]. The One-Phase-Pull variant addresses the common many-to-one routing in a sensor network. In this case, data travels back on the same links that delivered the Interests. The disadvantage of one-phase-pull is that it assumes symmetric links. Another variant

of diffusion is the One-Phase-Push diffusion where the sinks are passive and sources are active. The sources propagate the exploratory data throughout the network and on receiving this data, the sink nodes transmit a reinforcement message which is propagated back to the source. Further non-exploratory data follows only the gradient setup by the reinforcement messages.

Deluge [14] is a recent epidemic protocol for code propagation. It uses Trickle's technique of periodic advertisements and adds support for distributing large objects. Since Aqueduct's design borrows significantly from Deluge, Deluge will be discussed in more detail in Chapter 3.

Reliable multicast protocols appear to be the appropriate choice for reprogramming a subset of a network. Existing multicast protocols for wired networks are inappropriate for WSNs, however, because they may assume unlimited resources when resources are very limited on sensor nodes, and they do not take advantage of the RF broadcast medium, which would mean redundant transmission of data. Existing multicast protocols for WSNs are oriented for general-purpose use and streaming of low-bandwidth data. They do not address some of the unique properties of the reprogramming problem. These properties are (a) code images are very large, (b) the size of an image can be exactly known before an update begins, and (c) code images must be cached in their entirety on any node that plans to execute them. Reprogramming protocols address (a) by attempting to maximize throughput while minimizing control overhead. The result of (b) and (c) is that a reprogramming protocol can be purely receiver-initiated, i.e. NACK-based or pull-based. Designers of multicast protocols have avoided purely receiver-initiated protocols, since these protocols require either infinite memory (which of course is impossible) to be able to respond to NACK's for any previously transmitted packet, or they require the involvement of the application layer to be able to respond to NACKs [20]. Because of (b) and (c) the size of the data is known to be finite and cached somewhere in the network. Since reprogramming protocols like Deluge can be

receiver-initiated, a node can manage its own state locally. Sender-initiated protocols require parents to track the state of their children. This adds complexity to the protocol and makes it less robust to node failure. Since Deluge nodes manage their own state, they know exactly what data they need, and they will request it no faster than they process it. Thus a receiver-initiated protocol only transmits as much data as needed and gets flow control for free.

## Chapter 3

### Aqueduct Architecture

This chapter first examines the Deluge protocol in more detail before describing the key modifications that were instituted by Aqueduct in order to support efficient heterogeneous code propagation.

#### 3.1 Deluge and Heterogeneity

##### 3.1.1 Deluge Design

Aqueduct's code propagation mechanism relies heavily on the framework and engineering principles provided by the Deluge protocol [14]. Deluge's design reliably floods software updates throughout the WSN. The Deluge protocol assumes that all nodes in the network are running the same application and are interested in the same code updates. Robustness is achieved by the reliable broadcast mechanism. Deluge's flooding is a form of epidemic protocol, which means data propagates across a network through local interactions much like how an epidemic spreads. This epidemic employs a polite gossip technique for advertising code summaries, meaning nodes will periodically advertise a summary of their code, but only if less than some threshold of advertisements have already been overheard. Deluge uses a **pull-based**, or **receiver-initiated**, paradigm for distributing data, in which a node will only broadcast a piece of the code image when a receiver requests it. Deluge manages large data objects by dividing them into pages, which are further divided into communication packets. A node will not request

a page of an object until all previous pages have been downloaded completely. A node maintains a bit vector to track which packets from the current page it has received. This allows a sending node to stream whole pages, while receiving nodes can receive packets in any order and detect and handle dropped packets. Downloading only a page at a time keeps the bit vector at a manageable length. Once a node has downloaded some pages, it can then become a source of those pages for other nodes, even before it has downloaded the entire image. This allows **spatial multiplexing** or **pipelining**, where the base station can start sending the next page even as the previous page is still propagating through the network. Splitting objects into pages also enables updates to include only the pages of software that have changed between versions. Deluge uses only local interactions to route data, so it easily scales to large networks, since no global routing state has to be maintained. Deluge uses the redundancy inherent in dense networks to increase reliability. Deluge puts a limit on how many redundant advertisement and request packets are sent, and receiving nodes can eavesdrop for data packets that their neighbors have requested, thus as a network grows in density, the number of packets transmitted remains constant.

Deluge is implemented as a state machine. Normally a node is in the MAINTAIN state. While in the MAINTAIN state, nodes divide time into rounds. At some point in each round, a node broadcasts a summary of its code. A summary is a pair consisting of the version number of the code and the highest available page. If a node hears that another node in its neighborhood has an obsolete version, the current node will broadcast a profile of the current version. A profile consists of a vector listing the age in version increments of each page. The obsolete node will match this profile against its own version number to determine which pages it needs to update. If a node detects that its code is obsolete, it switches to RX state and unicasts to the current node a request for the next needed page. Unicasting the request reduces the chance for data collisions due to multiple nodes trying to send replies. A node that receives a request switches to

TX state and sends all the packets requested for the page. If the obsolete node does not have all the packets after a timeout, it sends a request again. After a certain number of timeouts, a node switches back to MAINTAIN state without a complete page, rather than attempt to maintain a bad connection.

Deluge builds on top of Trickle [23] and adds support for dissemination of large data objects. In Trickle, a node sends out summaries of the software on a given node to minimize traffic and speed updates. The nodes will also suppress sending their own summaries if a similar one has been heard recently. If a node hears an out-of-date summary, it sends an update. The main advantages of Trickle are the limitation of network traffic when no updates are being made and the speed of propagation. A Deluge node will only send a summary or a profile if it has heard less than some threshold summaries or profiles in that round. This technique suppresses redundant messages in dense networks. Nodes in any state may receive data messages, this reduces the number of requests and allows one node to update several others simultaneously. Nodes also suppress sending requests when a request for a lower-indexed page is overheard from another node. This allows the node that has fallen behind in updates to catch up, which enables more simultaneous updates.

### **3.1.2 Adapting Deluge For Heterogeneity - A First Attempt**

A simple way to support heterogeneity in Deluge would be to add a field to the Deluge packets that indicates which application is being reprogrammed. This requires the least modifications to the Deluge protocol and its state machine. In fact, the implementation of Deluge that is distributed with TinyOS already allows two different images to be reprogrammed.

A problem with this approach becomes apparent when the group of nodes being reprogrammed is scattered across the network, e.g. regions A and B in Figure 1.1 are scattered and separated by an intermediate region D consisting of nodes that have no

interest in the code being reprogrammed. Since Deluge requires every node in the network to cache a complete copy of the image being reprogrammed (a side effect of assuming homogeneity), many of the nodes will cache a copy of an application they will never execute, and few of them will ever be required to provide this application's code to a requester, e.g. all nodes in region C and even some nodes in region D. For these nodes, maintaining the application image is a waste of time, radio bandwidth, energy, and flash space.

A naive solution could be to have nodes ignore and not cache Deluge packets that are meant for an application in which these nodes have no interest. However, this would create partitions in the reprogramming network if the regions being reprogrammed are disjoint and not every region contains a seed node. What is needed is a protocol that limits reprogramming traffic to the nodes that are interested in the update, but is able to bridge the gaps between regions of interested node that are filled with uninterested nodes. Aqueduct is designed to fill this need.

### 3.2 Building Aqueducts

Aqueduct observes that different nodes serve different roles in a heterogeneous WSN. Figure 1.1 indicates that the nodes in region D act as **forwarding** nodes, and are contrasted with nodes in regions A and B that play the role of **member** nodes. Aqueduct thus defines two classes of nodes: **member nodes** and **forwarding nodes**. Member nodes are interested in receiving code updates and will be executing the received code image. They cache a copy of the entire code image, and download any updates they hear of. Every other node in the network is classified as a forwarding node. These nodes do not cache the entire application, because they do not need to run it, and they avoid downloading updates. The basic principle behind Aqueduct is that member nodes exchange code through normal Deluge interaction, but when member nodes are separated by more than one hop of forwarding nodes, the forwarding nodes create a

bridge by acting as proxies for member nodes on either side of the gap.

Figure 3.1 illustrates the concept of building aqueducts between a node interested in receiving code updates, and a node capable of providing a fresh update. Aqueduct is grafted onto the three-phase Advertise-Request-Data cycle of Deluge. The primary difference is that nodes that are not interested in the code may nonetheless need to be involved in forwarding. When a member node advertises its code summary, it first sets its DTB (distance to “base”, i.e. itself) to zero. This code summary is flooded throughout the network. Intermediate nodes that hear this code summary will act as proxies and advertise that they too have this code summary. However, each intermediate node will also increment the DTB by one and will further substitute its own ID. Thus, in Figure 3.1(a), the advertisement propagated by  $F_D$  is not the same advertisement as sent by M1, though the effect of propagating advertisements is the same. Intermediate nodes will reject identical advertisements with DTBs larger than already observed, e.g.  $F_E$  will reject the advertisement from  $F_C$ . When an advertisement first reaches an interested member node, then that node switches to RX state. All other nodes are still in MAINTAIN state at this point.

Just as in Deluge, M2 will **unicast** its Request, containing a bit vector of what pages are desired. In Deluge, this is unicast to an immediate neighbor, whereas in Aqueduct the request is unicast to the immediate parent **closest** to the advertising node. Figure 3.1(b) shows that the Request traverses the shortest path back to the member node M1. In actuality, the Request is transformed at each hop into a local Request from child to parent, and each intermediate node switches to the FORWARD state. When a local Request reaches the member node M1, then M1 switches to TX state, as would normally occur in the Deluge protocol.

In the third phase, shown in Figure 3.1(c), each node along the forwarding path pulls code image updates from its parent using the machinery of Deluge. Eventually, the code efficiently propagates to the destination M2 along this bridge, and does not

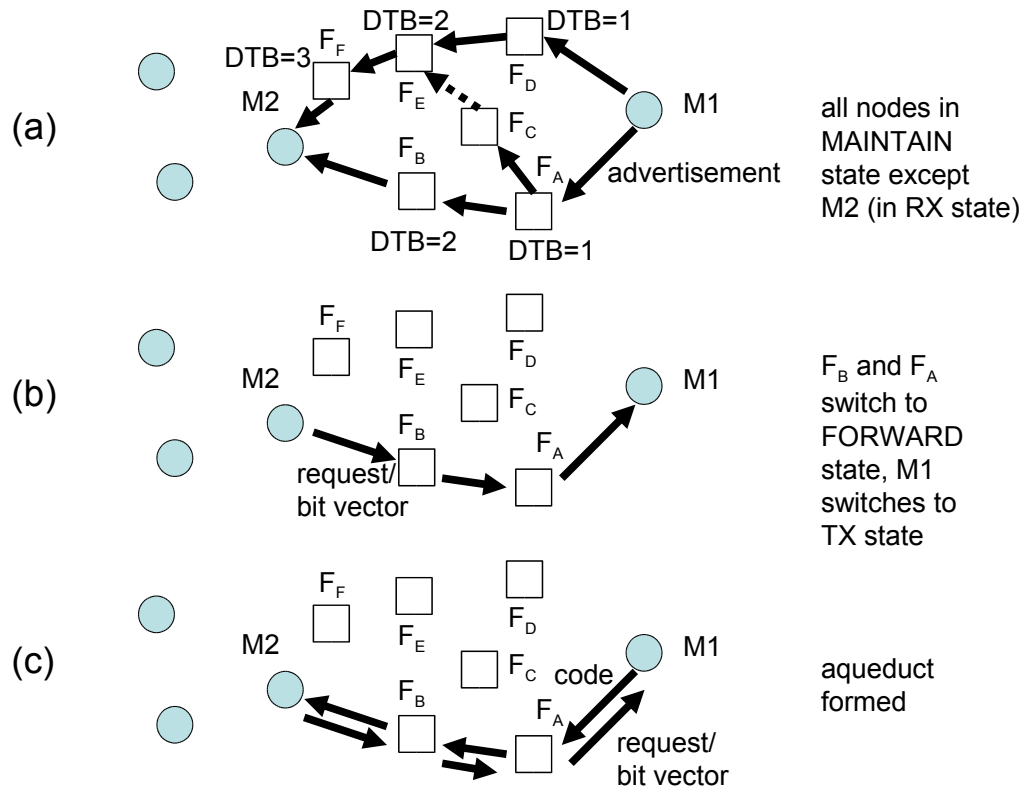


Figure 3.1: In Aqueduct, forwarding nodes are used to construct bridges between a node M1 advertising a fresh code image and a node M2 interested in obtaining such an update.

involve any of the other intermediate nodes. Thus, Aqueduct leverages the existing flood of code advertisements of Deluge to calculate the shortest path for forwarding data, and then efficiently unicasts code updates only along this path. Aqueduct achieves this while only adding a single field, the DTB, to the original Deluge protocol, resulting in only a small increase to the protocol’s control overhead. This approach is completely decentralized and can be applied anywhere in the network where there is a gap that needs to be efficiently bridged between two member nodes.

There are a variety of aspects not shown by this idealized figure. First, there may be multiple member nodes advertising that they have the same freshest code image. In this case, the DTB again is used to effectively select the shortest path to a member node advertising the code image. Preference is also given to advertisements that indicate the most available pages. While this means that requests might not always be routed to the nearest node that can fill them, it is necessary to ensure that nodes can find all the available pages in the network, otherwise a member node with only a partially downloaded image could find itself as the root of its own SPT with no route to a node that has the rest of the image. This idealized figure also assumes symmetric links, when in reality links can be quite asymmetric. How Aqueduct accounts for asymmetry is discussed later.

### 3.3 Aqueduct State Machine

The main objective of the following subsections is to describe how Aqueduct modifies the Deluge state machine to implement the role of forwarding nodes, which efficiently build bridges in a manner that is also robust to wireless variability. Deluge is originally organized as a three-state state machine and specifies its transitions with a series of formal rules [14]. This thesis follows the terminology laid out by Deluge. Aqueduct adds a variety of state transition rules to the three states of Deluge and also adds a fourth state called FORWARD. These four states are shown in Figure 3.2.

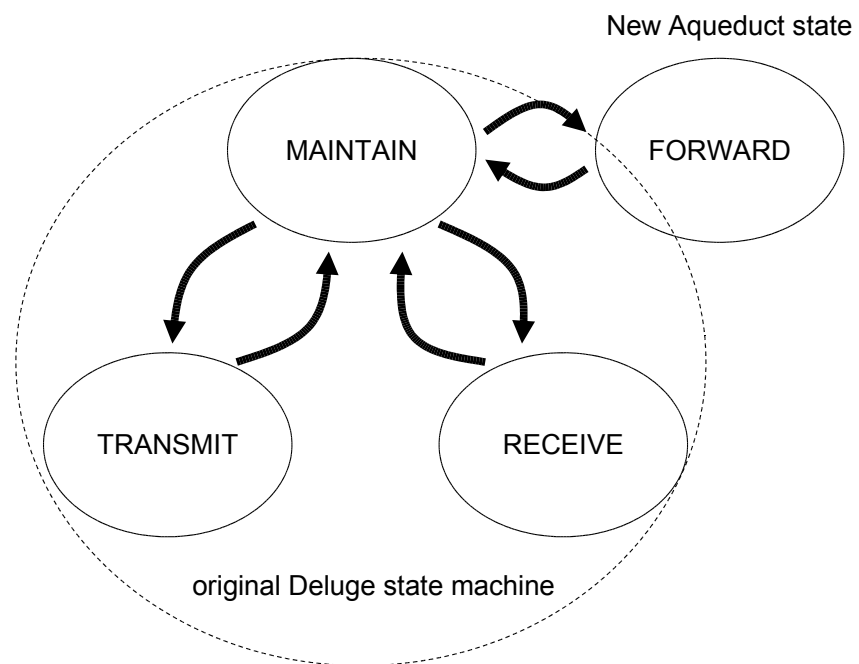


Figure 3.2: Aqueduct adds a new FORWARD state to the original 3-state Deluge state machine. The FORWARD state allows forwarding nodes to forward advertisements without necessarily storing all code updates.

### 3.3.1 MAINTAIN State

Nodes in MAINTAIN state periodically broadcast summaries of their code. A summary takes the form  $\{v, \gamma, DTB\}$ , where  $v$  is the version number,  $\gamma$  is the index of the highest available page, and  $DTB$ , is the distance to base. The DTB is the depth of the node in the shortest path tree (SPT) that has as its root the member node with the greatest known available page index. Member nodes always advertise a DTB of zero. The first two fields are from Deluge, and the DTB is added by Aqueduct. As in Deluge, the period between summary advertisements increases if no inconsistencies are detected in the network. Nodes in MAINTAIN state divide time into rounds. The duration of round  $i$  is specified by  $\tau_{m,i}$  and is bounded by  $\tau_l$  and  $\tau_h$ . During each round, a node broadcasts a summary of its code, but only if it has overheard less than some threshold  $k$  summaries during the first part of the round. As in Deluge, the length of a round starts at  $\tau_l$  and doubles up to  $\tau_h$ . The rules for a node in MAINTAIN state are summarized below. Some of the more formal statements of rules in this and the next subsections are repeated from [14] for convenience.

M.1 During any given round, broadcast a summary after half the round has passed, if less than the maximum number of advertisements has been overheard. More formally, during round  $i$  with start time  $t_i = t_{i-1} + \tau_{m,i-1}$ , broadcast an advertisement with summary  $\phi$  at time  $t_i + r_i$ , where  $r_i$ , only if less than  $k$  advertisements with summary  $\phi' = \phi$  have been received since time  $t_i$ .

M.2 If any packet is overheard that indicates an inconsistency in the network, then set the round duration to the minimum and start a new round. More formally, if any packet that indicates an inconsistency among neighboring nodes (i.e. advertisements with  $\phi' \neq \phi$ , any requests, or any data packets) is overheard during round  $i$ , set  $\tau_{m,i}$  to  $\tau_l$  and begin a new round.

- M.3 If no packet indicating an inconsistency is overheard, double the length of a round. More formally, at the beginning of round  $i$ , if no overheard packet indicates an inconsistency among neighbors during the previous round, set  $\tau_{m,i}$  to  $\min(2 \cdot \tau_{m,i}, \tau_h)$ .
- M.4 If a summary indicating an obsolete version is overheard, broadcast the image profile, if less than the maximum number of profiles has been overheard. More formally, during round  $i$ , transmit the object profile for version  $v$  at time  $t_i + r_i$  only if an advertisement with  $v' < v$  was received at or after time  $t_i$  and less than  $k$  attempts to update the object profile to version  $v$  and been overheard.
- M.5 If an advertisement is overheard indicating a higher available page than this node has, **a member node** will set its DTB to the received DTB plus one and transition to RX state, unless (i) a request for a this page or a lower-index page was overheard in the last two rounds, or (ii) a data packet for a lower-index page was overheard in the last round. More formally, on receiving an advertisement with  $v' = v$  and  $\gamma' > \gamma$ , a member node will set  $DTB$  to  $DTB' + 1$  and transition to RX unless (i) a request for a page  $p \leq \gamma$  was previously received within time  $t = 2 \cdot \tau_{m,i}$ , or (ii) a data packet for page  $p \leq \gamma + 1$  was previously received within time  $t = \tau_{m,i}$ .
- M.6 If a request addressed to this node for a page this node has available is received, **a member node** will transition to TX state. More formally, on receiving a request for data from a page  $p \leq \gamma$  from version  $v$ , a member node will transition to TX.

The above rules are the same as those for Deluge, except rules M.5 and M.6, which are amended to only apply to member nodes. Rule M.5 is also changed to maintain the member node's DTB field. Note that while a member node always advertises a DTB

of zero, it still maintains an internal DTB that will be greater than zero if it knows it does not possess the highest available page.

Aqueduct adds two more rules, which are variations on rules M.5 and M.6:

M.7 On receiving an advertisement with  $v' = v$  and  $\gamma' > \gamma$  or an advertisement with  $v' = v$ ,  $\gamma' = \gamma$  and  $DTB' < DTB$ , **a forwarding node** will set  $\gamma$  to  $\gamma'$  and  $DTB$  to  $DTB' + 1$ .

M.8 On receiving a request for data from a page  $p \leq \gamma$  from version  $v$ , **a forwarding node** will transition to FORWARD.

Rule M.7 has the effect that a forwarding node does not transition to RX state when it hears about a new available page, instead it simply claims that it already has the new page. Maintaining the DTB field has the effect that an SPT is formed with the closest member node advertising the highest page as the root. The SPT is used to forward requests from a member node to the closest member node that can fill the request.

The use of DTB fields to form a routing tree could be seen as a violation of Deluge's local-interactions-only policy. However, no control packets are passed over more than one hop, and the network maintains no global routing state.

### 3.3.2 RX State

The RX state in Aqueduct is nearly identical to the RX state in Deluge.

R.1 After not receiving a request or data packet for a given amount of time **scaled by the DTB**, transmit a request to  $S$ , the node whose advertisement put this node into the RX state. More formally, after not receiving a request or data packet for time  $t = DTB \cdot \omega \cdot T_{tx} + r$ , where  $r$  is a random value in the range  $\tau_r$ , transmit a request to  $S$ .

R.2 After a given number of requests time out, transition to MAINTAIN state even if the page is incomplete. More formally, after  $\lambda$  requests with a packet transmission rate of  $\alpha' < \alpha$ , set  $DTB$  to  $DTB_h$  and transition to MAINTAIN even if page  $\gamma + 1$  is incomplete.

R.3 If all packets for the page have been received and the data passes the CRC, mark the page as complete and transition to MAINTAIN state. More formally, if all packets for page  $p$  are received and data for page  $\gamma + 1$  passes the CRC, mark page as complete and transition to MAINTAIN.

The above rules are the same as Deluge, except that Rule R.1 is amended to scale the timeout value by the  $DTB$  to allow for the delay caused by data being forwarded over multiple hops in the SPT. If  $S$  is a member node, then the timeout will be equivalent to that used in Deluge, since  $S$  would have advertised a  $DTB$  of zero and therefore the receiving node would have a  $DTB$  of one.

### 3.3.3 TX State

The TX state in Aqueduct is identical to the TX state in Deluge, and is summarized below:

T.1 If a request is received for packets from a page this node is already serving, add the requested packets to the list of the packets yet to be served. More formally, on receiving a request for packets  $\Pi'_{rx}$  from page  $p$ , set  $\Pi_{tx}$  to  $\Pi_{tx} \cup \Pi'_{rx}$ .

T.2 Broadcast packets from the list of requested packets until the list is empty, then transition to MAINTAIN state. More formally, continue broadcasting packets in  $\Pi_{tx}$  in round-robin order and remove each broadcast packet from  $\Pi_{tx}$  until  $\Pi_{tx} = \{\}$ , then transition to MAINTAIN.

### 3.3.4 FORWARD State

Aqueduct adds one new state to the state machine to account for the role of forwarding nodes. One iteration (i.e. execution between timeouts) of the FORWARD state is like executing the TX state followed by one iteration of the RX state. In fact, FORWARD state could have been implemented by added new transitions to the existing Deluge states, but the current approach was thought to be simpler. In this section  $p$  is the page being served by a node in FORWARD state,  $\Pi_{tx}$  is the bit vector representing packets yet to be served,  $\Pi'_{rx}$  is the bit vector of packets in an incoming request,  $\Pi_{cache}$  is the bit vector of packets this node has cached (it may not have received the entire page), and  $\Pi_{rx}$  is the bit vector of packets this node will request from its parent in the SPT.

- F.1 On receiving a request for packets  $\Pi'_{rx}$  from page  $p$ , set  $\Pi_{tx}$  to  $\Pi_{tx} \cup \Pi'_{rx}$ .
- F.2 Broadcast packets in  $\Pi_{tx} \cap \Pi_{cache}$  in round-robin order and remove each broadcast packet from  $\Pi_{tx}$  until  $\Pi_{tx} \cap \Pi_{cache} = \{\}$ , then if  $\Pi_{tx} = \{\}$  transition to MAINTAIN, otherwise set  $\Pi_{rx}$  to  $\Pi_{rx} \cup \Pi_{tx}$  and send a request to  $S$ , the node that advertised the lowest DTB, and wait.
- F.3 After not receiving a request or data packet for time  $t = DTB \cdot \omega \cdot T_{tx} + r$ , where  $r$  is a random value in the range  $\tau_r$ , (see [14] for symbols) perform rule F.2.
- F.4 After  $\lambda$  requests with a packet transmission rate of  $\alpha' < \alpha$ , set  $DTB$  to the maximum value and transition to MAINTAIN even if  $\Pi_{tx} \neq \{\}$ .
- F.5 If all packets for page  $p$  are received and data for page  $p$  passes the CRC, mark page as complete and perform rule F.2.

### 3.4 Further Enhancements

After experimenting with this initial design of Aqueduct, several critical enhancements were found to be needed to improve its performance.

#### 3.4.1 Link Symmetry

While examining packet traces from nodes running Aqueduct, a frequent problem was observed, as illustrated in Figure 3.3(a). A node  $F_D$  would hear an advertisement from a node  $F_A$  that was quite far away but much closer to the root M1 of the SPT. Since  $F_A$  would have a low DTB compared to  $F_D$ 's closer neighbors,  $F_D$  would adopt  $F_A$  as its parent in the SPT. When a request was unicast to  $F_D$ , then  $F_D$  would unicast it to  $F_A$ . Since  $F_D$  and  $F_A$  would be so far apart, the link would be of poor quality and  $F_A$  would usually not hear  $F_D$ 's requests. Even if it did,  $F_D$  would miss most of  $F_A$ 's subsequent data broadcasts. Eventually  $F_D$  would time out too many times and drop  $F_A$  as a parent after wasting a lot of time, thereby lowering throughput and increasing delay.

Aqueduct's solution is to have nodes keep neighbor lists and include them in their advertisements. Every time a node hears a transmission from a neighbor, it adds the neighboring node to its list. The neighbor lists are implemented as circular buffers of limited size, so that infrequent transmissions from distant nodes are eventually forgotten as transmissions from closer nodes are received more frequently.

Rules M.5 and M.7 are amended so that, on receiving an advertisement from a node  $F_A$ , a node  $F_D$  will only update its DTB and accept  $F_A$  as its parent in the SPT if  $F_D$  sees its own address in  $F_A$ 's neighbor list. This increases the probability that  $F_D$  and  $F_A$  share a high-quality link, although poor links are still possible. **Thus, aqueducts are built only upon symmetric shortest path links, and avoid paths that may be shorter but are over unreliable asymmetric links.** This addition was essential

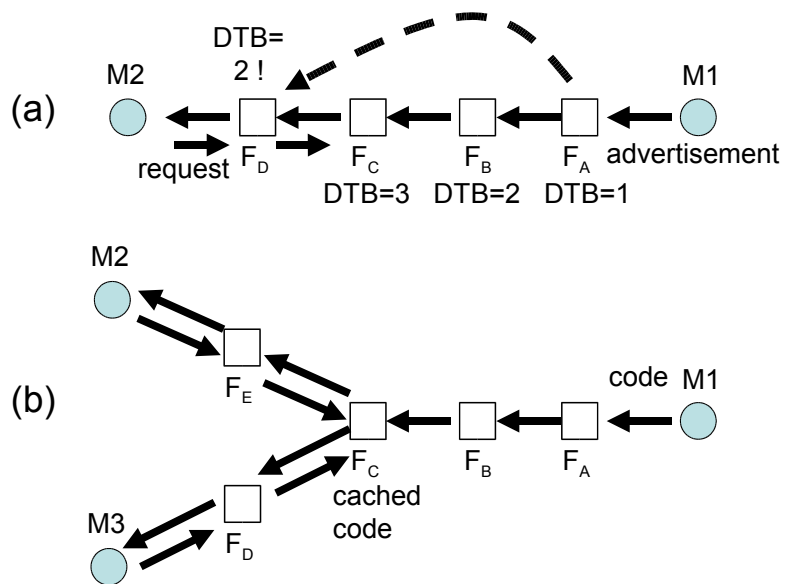


Figure 3.3: (a) Asymmetric links can severely reduce code propagation performance, by giving a node an illusion that its parent is reachable. Aqueduct's solution is to build the SPT only via dependable symmetric links. (b) Judiciously caching code updates at forwarding nodes can dramatically reduce the turnaround time for servicing a request.

to improving the robustness and performance of the Aqueduct protocol. Integrating an even more stringent quality metric is discussed in future work.

Neighbor lists are only exchanged locally; they are never transmitted over more than one hop. Furthermore, a node only stores its own neighbor list. Thus this technique scales well for large networks. Neighbor lists do add more overhead to the original Deluge protocol. However, this overhead is confined to advertisements, which are a minority of the total packets exchanged during an update. The increase in overhead is more than made up for by the reduction in lost packets that are due to poor-quality links.

Neighbor lists also require more advertisements to be exchanged before the SPT is ready to forward requests and data. Previously, a child only had to hear one advertisement from its parent to determine its position in the SPT, but when using neighbor lists, the parent has to hear an advertisement from the child first and add the child to its list before the child will accept a DTB advertisement from the parent. However, if the network has been running for a few rounds, and nodes are not very mobile, then the neighbor lists will already be constructed and valid when a code update is injected into the network. In this case constructing the SPT will then occur as quickly as it would have without neighbor lists.

### **3.4.2 Caching**

By giving priority to nodes that have fallen behind, exceptions (i) and (ii) of Deluge rule M.5 help keep nodes synchronized at the same page during updates. When nodes are at the same page, they can take advantage of the broadcast medium by eavesdropping on data being sent in response to another request. This reduces the total number of data packets that need to be sent in a dense network.

Keeping nodes synchronized also has advantages for Aqueduct. When two member nodes are in the same subtree of the SPT of forwarding nodes, then if those nodes are requesting the same page, then the closest common ancestor of the two nodes can

aggregate the incoming requests. For example, in Figure 3.3(b), if the ancestor  $F_C$  receives a request forwarded from member node  $M2$ , it will forward the request upstream to member node  $M1$  and wait for data packets to come back downstream. Node  $F_C$  then caches the data packets before forwarding them back down to  $M2$ . If member node  $M3$  sends a request for packets from the same page as  $M2$ , then  $F_C$  can serve the request from its cache. This avoids forwarding the request further upstream, which would require the upstream nodes to respond with the same data a second time.

The initial implementation of Aqueduct only cached the last page requested. This means that, in the previous example, if  $M3$  were to request a lower-indexed page than the one  $M2$  requested earlier, then  $F_C$  would “forget” the packets it had cached earlier to make room for the newly requested page. When  $M3$  later requests the same page that  $M2$  did, the request and data will have to be routed all the way to and from the root node  $M1$  again.

To minimize this problem, the protocol rules were modified to support caching of multiple pages. The cache is allocated in the sensor node’s external flash memory as a circular buffer; the first page cached is the first page removed when the buffer is full. If a page is downloaded completely by a forwarding node in FORWARD state, the node will save the page in the cache. Rule M.8 is amended so that if the forwarding node later receives a request for a page that is in the cache, instead of transitioning to FORWARD state, the node transitions to TX state. Note that only completely downloaded pages are stored in the cache. If a forwarding node does not receive a request for a full page, then it will never download the full page.

## Chapter 4

### Protocol Evaluation

This chapter compares the performance of Aqueduct and Deluge. Two variants of Aqueduct were tested. Aqueduct-Sym utilized Aqueduct’s symmetric link mechanism to build a robust SPT. Aqueduct-NoSym built the SPT using only minimum hop count and did not factor in the quality or symmetry of links. All experiments were performed on a 25-node indoor testbed consisting of Mica2 motes running the MANTIS operating system (MOS) [7]. The testbed was formed into a  $5 \times 5$  grid suspended from a ceiling with a spacing of two feet between each node. Each node was attached to a programming board, which was then connected to a USB-to-serial convertor and an AC power adapter. The cables for each row of five nodes were connected to a USB hub, and the five USB hubs were daisy-chained off a PC running Linux. Daisy-chaining the hubs ensured that the device files for the USB convertors would be assigned in a predictable order. Figure 4.1 gives a schematic of the testbed, and Figure 4.2 is a photograph of the testbed. The transmit power of the nodes was decreased in order to form a multihop topology. Specifically, the CC1000 transmit power was set to 6. In our implementation of Deluge and Aqueduct, each page is 2688 bytes with 48 packets per page, and each packet has a data payload of 56 bytes. In the TinyOS implementation of Deluge each page size was 1104 bytes with 48 packets per page and each data packet had a data payload of 23 bytes. The protocol parameters were  $k = 1$ ,  $\lambda = 2$ ,  $\tau_l = 2$  seconds,  $\tau_h = 256$  seconds,  $\tau_r = 0.5$  second,  $\omega = 8$ , and  $T_{tx} = 80$  milliseconds. The performance

numbers for Deluge presented in [14] could not be duplicated, since the Deluge paper does not give any details about the dimensions of network used.

Each experiment began with the version number being incremented on the seed node, this caused the other member nodes to believe the code on the seed node was a new version, thus an update would commence. The time of the initial version change on the seed node and the time that the last member node completed downloading the update were both logged; their difference was the completion time for reprogramming the network. Packets both sent and received were counted for each node. The counts were broken down by sending node and by type. These counts were collected over a wired back-channel, which was solely used for collecting metrics and debugging. Since forwarding nodes running Deluge could continue updating even after all the member nodes had completed updating, packets were counted until 30 seconds after the last request or data packet was detected.

#### 4.1 Aqueduct and Deluge Performance

The first experimental setup used the node in one corner of the  $5 \times 5$  testbed as the seed node for updates. The node in the diagonally opposite corner was also configured as a member node. All the other nodes in the network were forwarding nodes. The image size was varied from two pages to ten pages in two-page increments. Each experiment was run five times.

The count of data packets provides a good estimate of a reprogramming protocol's efficiency, since the bulk of RF traffic during an update consists of data packets, and data packets are also larger than the other packet types. Figure 4.3 shows the total count of data packets transmitted by the entire network. Deluge transmits roughly twice as many packets as Aqueduct-Sym. As the size of the network expands in relation to the size of the subset being reprogrammed, it is expected that Aqueduct would demonstrate even greater savings. The use of asymmetric links by Aqueduct-NoSym causes repeated

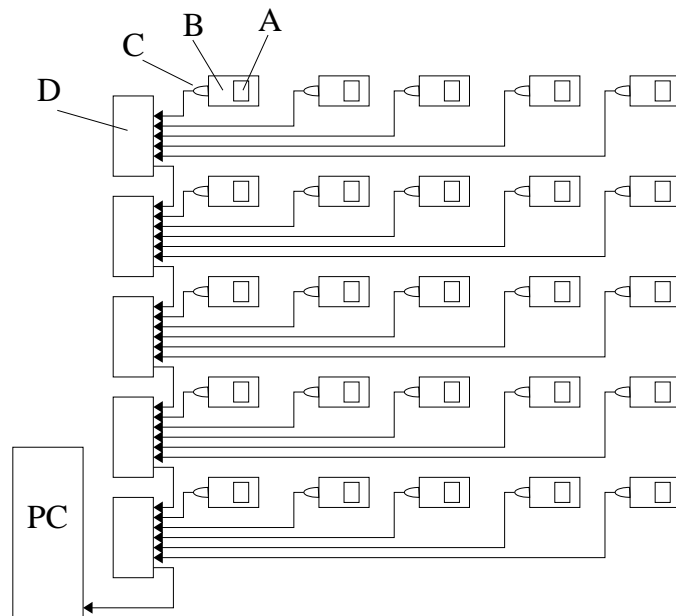


Figure 4.1: Testbed Schematic. The testbed is built out of A. Mica2 motes B. programming boards C. USB-to-serial convertors and D. USB hubs.



Figure 4.2: Photograph of the testbed.

retransmissions of data packets, thus it shows poor efficiency as compared to Aqueduct-Sym.

The improvement of using symmetric links in Aqueduct-Sym versus asymmetric links in Aqueduct-NoSym is also apparent in a comparison of the number of requests, as illustrated in Figure 4.4. Aqueduct-NoSym transmits roughly twice as many requests as Aqueduct-Sym. This was because many requests were to nodes that were too far away, causing timeouts and retransmission of requests. Thus, symmetry is quite beneficial in improving performance in comparison to ignoring the quality of links. Deluge resulted in the most requests transmitted, as the code image has to be globally propagated to the entire network.

Figure 4.5 and Figure 4.6 are projections of all the requests made in a run of the  $5 \times 5$  grid topology for Aqueduct-Sym and Deluge respectively. The upper left is the destination and the lower right is seed node used to propagate code. The node positions are slightly jittered to minimize overlap in drawing the links. The thickness of the link connecting any two nodes is a measure of the volume of requests along that link. We observe that Aqueduct requests largely propagate along particular paths. Deluge requests, however, are spread throughout the network.

Figure 4.7 shows the completion times of the three protocols. Aqueduct-Sym is somewhat slower than Deluge. In both cases the update has to traverse the same distance between the member nodes. Nodes running Deluge often have multiple sources to query for updates, and these sources are able to respond immediately. Nodes running Aqueduct, however, must often wait for a request a request to be served over multiple hops. If the request times out, then the whole process of routing the request over multiple hops must be repeated. Furthermore, the old request could be competing with the new request for use of the radio channel. A balance must be found for Aqueduct between timing out quickly to tolerate bad links and having the patience to wait for the data to be forwarded over multiple hops. The figure also shows that Aqueduct-

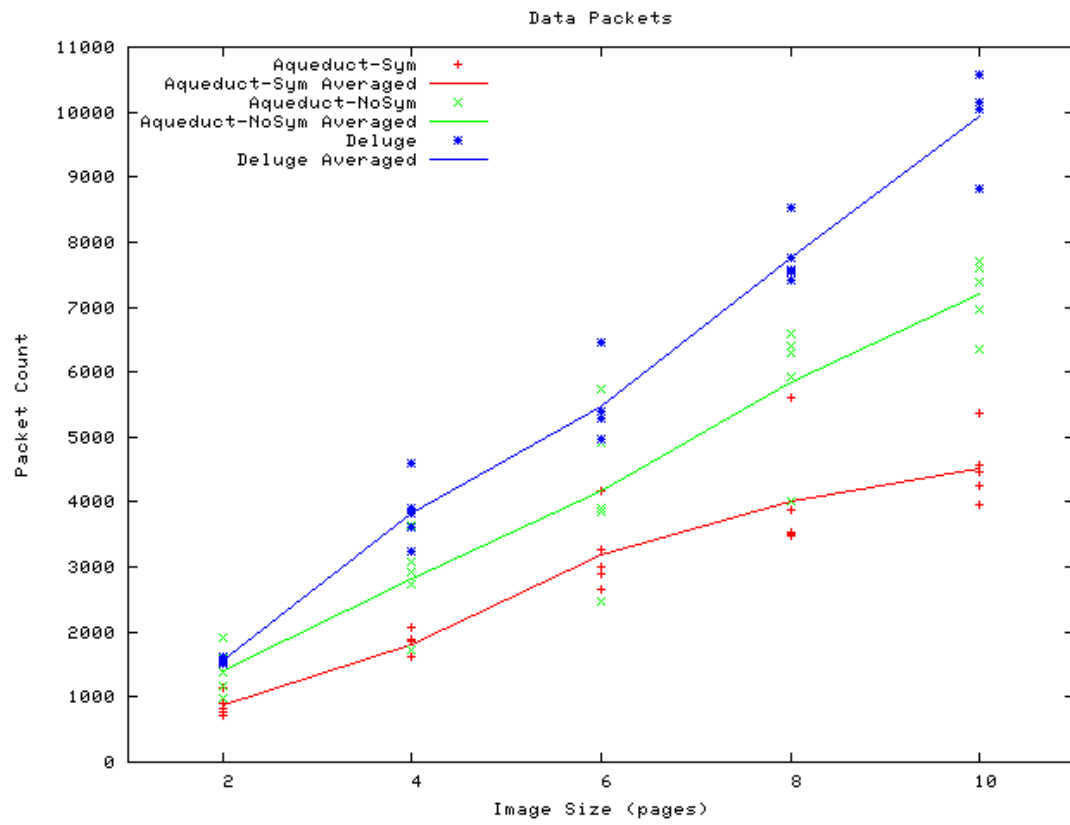


Figure 4.3: Total amount of data packets

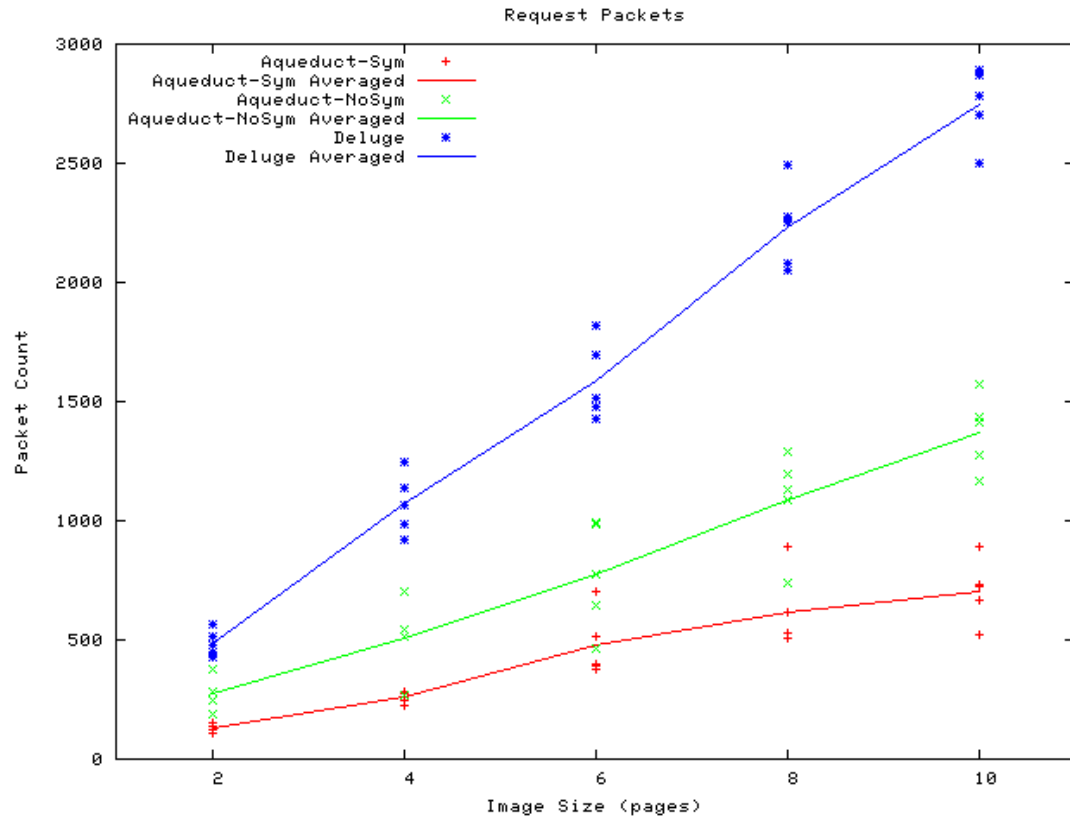


Figure 4.4: Total amount of request packets.

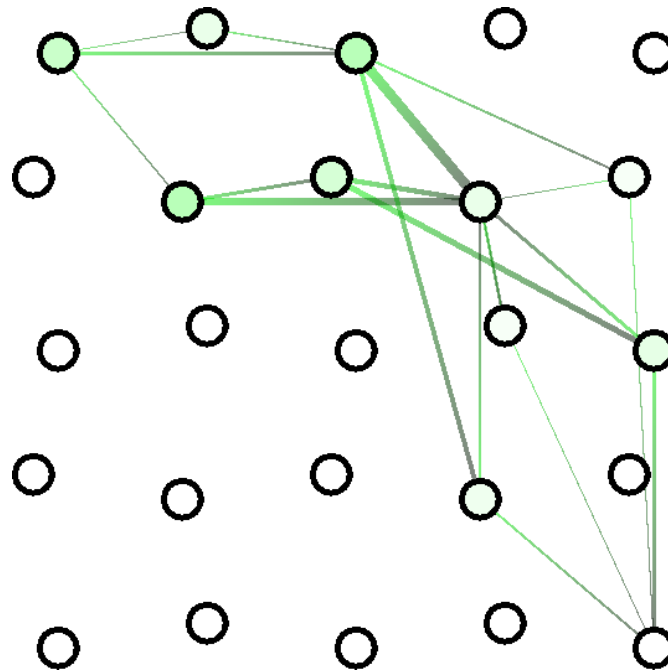


Figure 4.5: Projection of requests for Aqueduct-Sym.

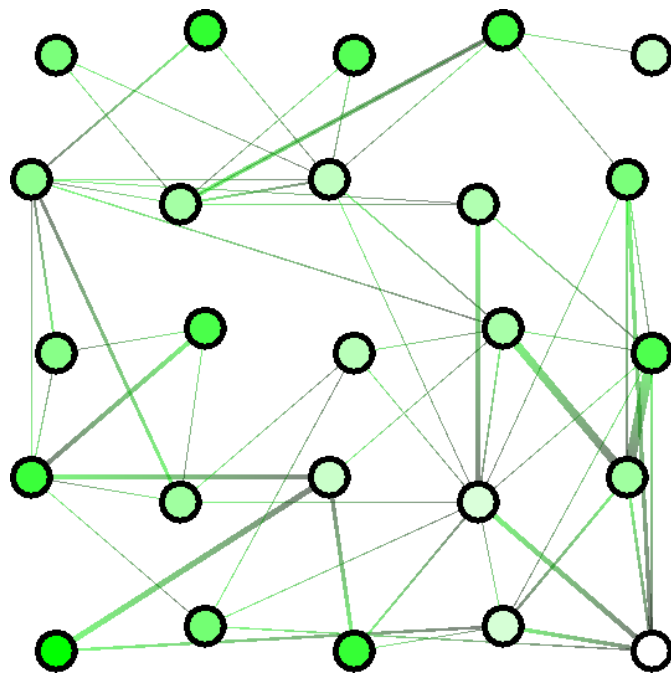


Figure 4.6: Projection of requests for Deluge.

NoSym is considerably slower than the other protocols due to the latency introduced by transmitting requests over bad links.

## 4.2 Caching

The next set of experiments were designed to test the effect of caching. The same corner node was used as the seed node, and the five nodes that comprised an opposite edge of the  $5 \times 5$  grid were configured as member nodes that would download the updates. It was expected that this configuration would cause the paths from each member node to the seed node to share many common forwarding nodes. Caching on these forwarding nodes should reduce latency and redundant data transmission. The image size was fixed at ten pages, and the cache size was varied from two to ten pages in two-page increments. Each experiment was run five times.

In addition to packet counts and completion times, statistics were also kept on cache usage. Figure 4.8 plots these data. Cache usage varies widely, and there appears to be little or no correlation between cache size and cache usage. This may be due to the policy of only caching completely downloaded pages. Incomplete pages are a common occurrence for Aqueduct. When data packets are lost, a page is incomplete. If a request times out on a member node, these incomplete pages are left orphaned on the forwarding nodes. If the request is re-transmitted on a new path, then all the forwarding nodes on the new path will only receive incomplete pages, since the original request was for only a subset of the page's packets. Forwarding nodes only cache an incomplete page if it is the currently requested page. If a request for a different page is received before the old page is completely downloaded, the old page is discarded. Possible solutions to this problem could be to cache incomplete pages or to try different values for the request timeout and  $\lambda$ , the maximum number of timeouts allowed in RX state. Figures 4.9 and 4.10 show similarly unimpressive results for completion time and data packet count respectively. It is important to note that preliminary experiments

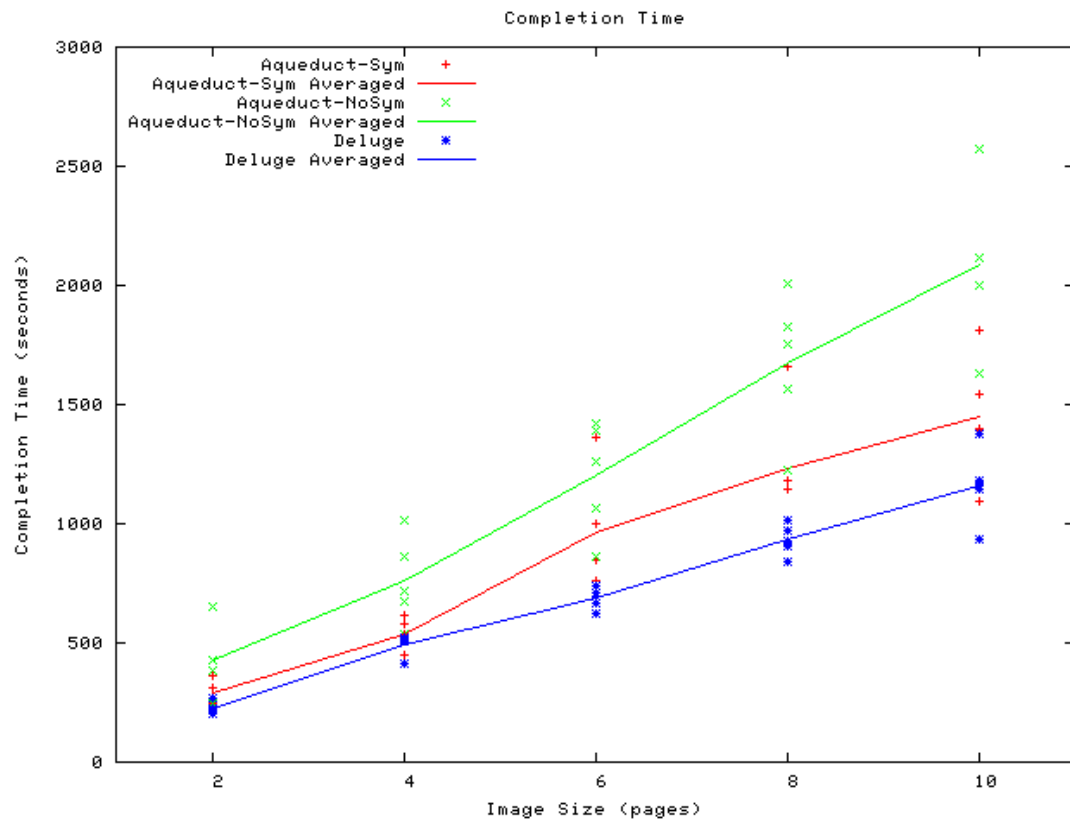


Figure 4.7: Total time to reprogram the member nodes.

showed an improvement in performance with increased cache size. These early experiments used  $\lambda = 8$ , which means that a node would retry requests more times than the current implementation ( $\lambda = 2$ ), which would give forwarding nodes a better chance at downloading an entire page.

### 4.3 Control Overhead

Table 4.1 lists the total bytes transmitted for all the experiments for each of the protocols. Overhead was computed by subtracting the total payload bytes from the total bytes transmitted and dividing the result by the total bytes transmitted. Aqueduct-Sym adds less than 6% control overhead over Deluge. This overhead is negligible when considering that Aqueduct saved around 50% in data traffic in the experimental results.

Finding networks where Deluge outperforms Aqueduct is an important consideration when comparing the two protocols. The worst case for Aqueduct in comparison with Deluge would be a homogeneous network. In this network Aqueduct would not use its heterogeneity support and would behave like Deluge, except that it would transmit larger advertisements. Table 4.1 shows that in the worst case Aqueduct should theoretically transmit less than 6% extra control data over Deluge.

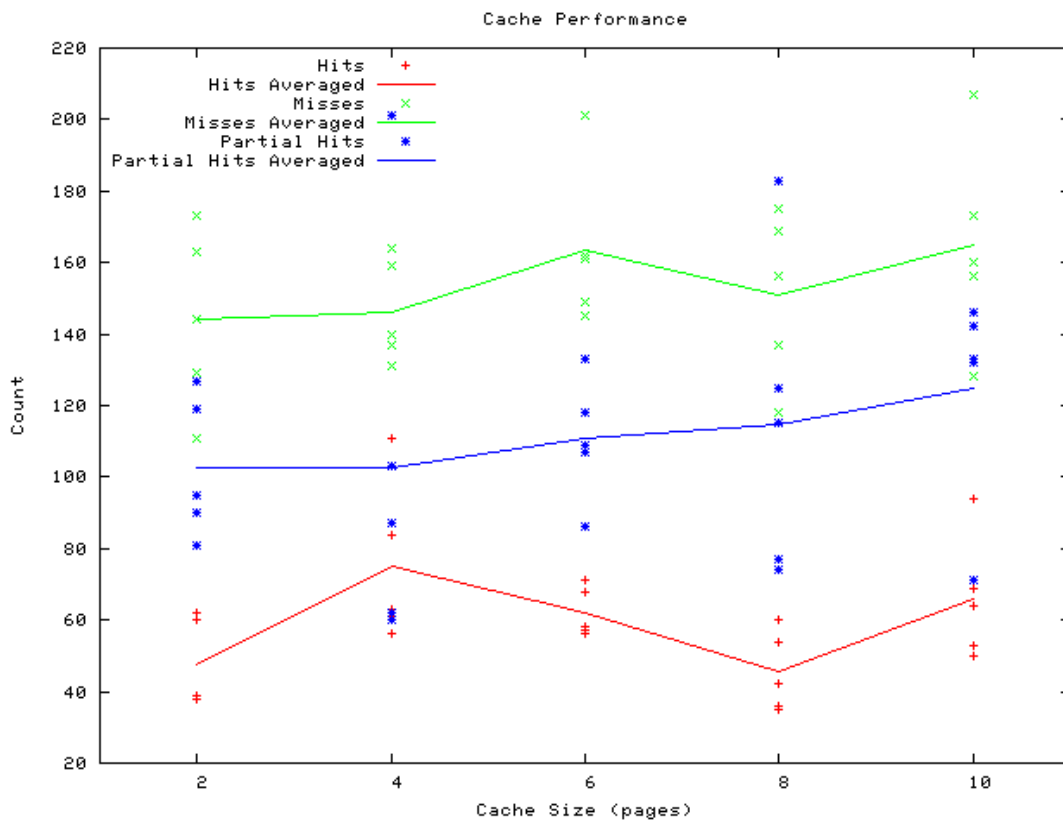


Figure 4.8: Cache Usage. **Hits** is the count of times a node was able to serve a request with a previously cached page. **Misses** are the times when a request had to be forwarded. **Partial Hits** are the times when the request matched the last page that was requested from a node. Some of the request may still have had to have been forwarded.

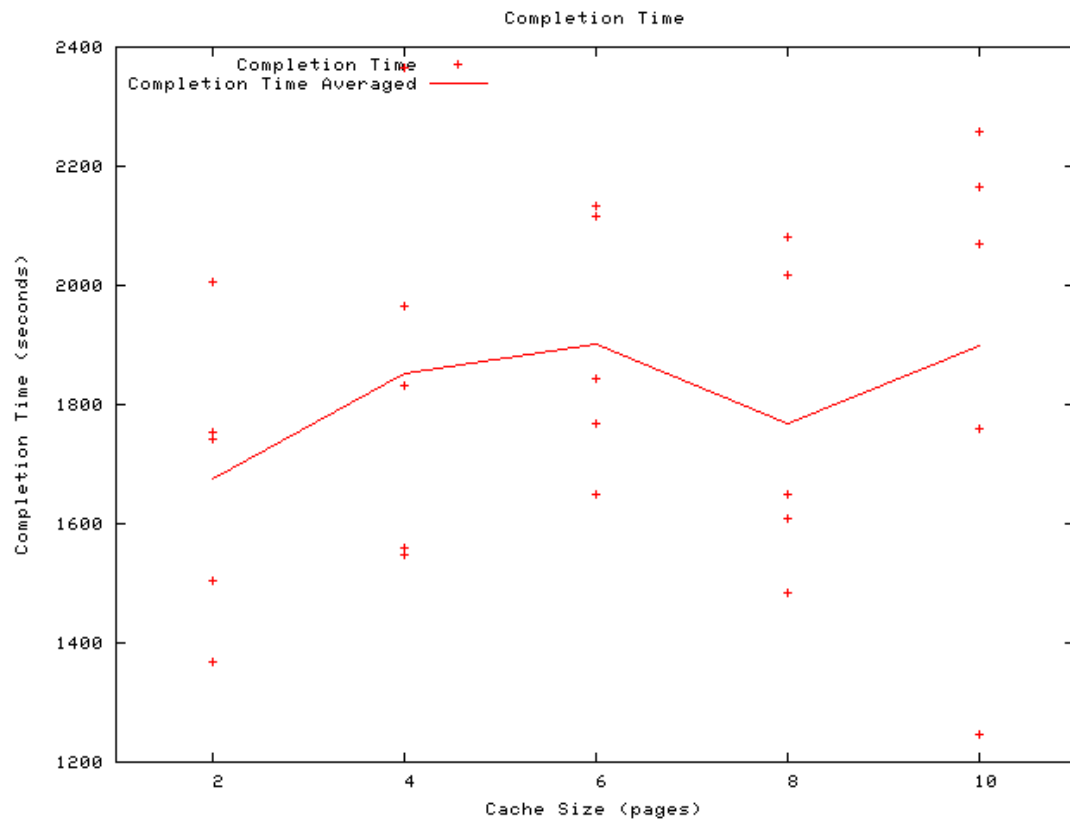


Figure 4.9: Effect of Caching on Programming Time

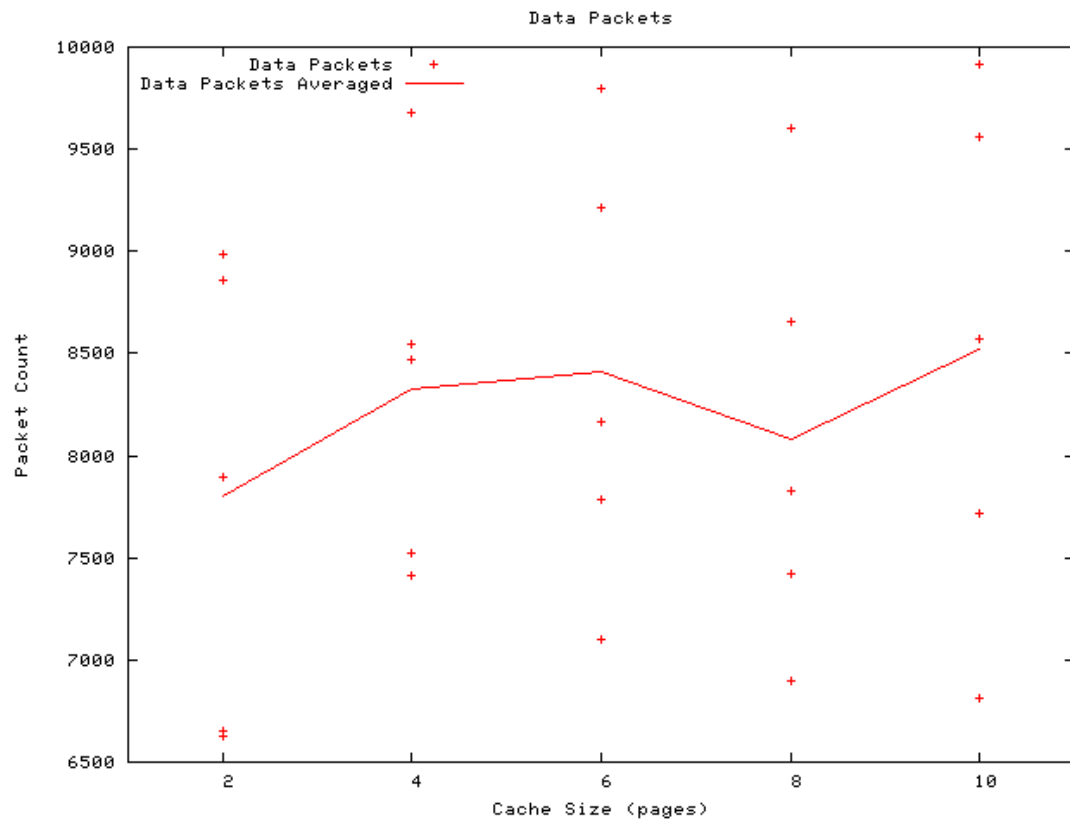


Figure 4.10: Effect of Caching on Data Transmission

Protocol	Summaries	Profiles	Requests	Data	Payload	Overhead
Deluge	203280	3406	649728	9150080	8006320	19.99%
Aqueduct-No-Sym	426376	2080	321472	6861952	6004208	21.12%
Aqueduct-Sym	638046	2145	174800	4608704	4032616	25.65%

Table 4.1: Total bytes transmitted and computed overhead for all experiments.

## Chapter 5

### Future Work

This chapter discusses the many avenues for additional Aqueduct research. The first step is to conduct a more detailed analysis of Aqueduct's behavior. While Aqueduct clearly wins over Deluge in terms of efficiency, it does take a long time for programming to complete, so there is still room for improvement, and there may be opportunities for further improvement in efficiency as well. Finding better values for request timeout parameters may solve some of the poor performance observed in chapter 4. Examining packet traces for every node on the network can give a detailed view of how Aqueduct works in real situations. This technique has already guided fundamental design choices for Aqueduct; it could now be used identify optimal parameter settings. It would also be interesting to find out if the optimal parameter settings are different for different network configurations. Simulation could be used to rapidly test different parameters on many different networks.

Different techniques for determining link quality should be explored, beginning with a review of existing link-quality literature. The current method is very simple and can still set up bad links. Once Aqueduct detects a bad link, it does nothing to avoid using the link later. It also does not have a memory for links that have worked well in the past. A more complex quality metric may appear to offer only a little improvement, but since the radio uses so much more power than the microcontroller, a little extra computation may be worth the effort. However, the current protocol's simplicity is

appealing, and making it more complex could make it more fragile.

Opportunities for improving Aqueduct performance may lie in smarter caching of data broadcasts. Currently only forwarding nodes that have received a request will store data packets. It may be wise for forwarding nodes to cache overheard data packets in anticipation of future requests. Having the data ready when a request comes in will mean fewer hops that the data will have to traverse on the way back. Caching of partially downloaded pages may also improve performance as discussed in section 4.2.

Aqueduct could also be more intelligent in routing requests to forwarding nodes that have previously requested the same data. Currently forwarding nodes send their requests to only one parent in the shortest-path tree that is rooted at a member node. It may be the case that there is a shorter path to another forwarding node that already has the data cached.

Security is a feature that will need to be supported in any serious reprogramming system. Enabling Aqueduct or Deluge on a sensor node is essentially opening a back door to an attacker. Authentication will be necessary to ensure that only authorized users can inject code into a network. Encryption will be needed to keep eavesdroppers from stealing software. It will be a challenge to see if Aqueduct can be enhanced with security in a way that allows it to remain efficient and robust.

Deng et. al. [16] describe an attack on sensor networks in which radio traffic patterns are analyzed to find vulnerable nodes. Since Aqueduct confines data to fewer paths in the network than Deluge, it is possibly more vulnerable to a traffic analysis attack. Deluge and Aqueduct both are robust to node failure, however. An attacker would have to destroy enough nodes to partition the network to get either of these protocols to stop updating.

Aqueduct can ensure that any data downloaded by a node is downloaded correctly; however, Aqueduct provides no mechanism to the user for determining that a given group of nodes has updated to a new version. Especially useful would be a “com-

mit” function that reboots nodes and executes the new version only if all the intended recipients have correctly downloaded it. Aqueduct’s main concern is transferring large files correctly; it would be better to add a commit function to separate network management protocol. Other functions of the management protocol could include setting the application that runs on a particular node, setting the mapping of application identifiers, resetting version numbers (they roll over at 255 in the current implementation), and booting into a failsafe mode. This protocol would be used infrequently and could probably be implemented as a reliable flooding protocol.

Aqueduct needs to be implemented to use less memory. The current implementation creates an instance of Aqueduct state for each application that could be reprogrammed, and no information is shared between instances. This allows nodes to monitor the network for updates of their member applications while simultaneously forwarding updates for other nodes’ applications. Since reprogramming is expected to be a relatively rare event in a mature network, the ability to support multiple simultaneous updates is probably not a valuable feature. If only one update is allowed at a time, then variables that track the progress of an update can be shared between applications. Neighbor lists are identical between applications, so they should be shared as well. Nodes would still need to store the version number, the available page index, the hop count, and the SPT parent address for every application in the network. This information is required so that forwarding nodes can determine that an update is present and propagate information about the update to member nodes. In order to allow as many applications in the network as possible, some method must be found for reducing the size of this list of application information, possibly only part of it would be cached, or perhaps the advertisement phase of Aqueduct could be altered to not require this list.

An Aqueduct implementation for TinyOS would allow a more “apples-to-apples” comparison with the original Deluge implementation, which is likely to be more mature than the MOS implementation of Deluge. The improvements made to Deluge should

work well independently of the implementation. Implementing for TinyOS would also allow the use of TOSSIM [21], a network simulator for TinyOS.

## Chapter 6

### Conclusions

This thesis has presented Aqueduct, a novel robust and efficient code propagation protocol for network reprogramming. Aqueduct adheres to four primary design principles: dynamic network reprogramming for **heterogeneous** WSNs with diverse hardware, software, and application roles; efficiency in terms of reduced code overhead by limiting involvement in forwarding; robustness to spatially irregular, time-varying RF links by constructing symmetric links; and incorporation of new capabilities into the framework provided by Deluge. Aqueduct adds new capabilities to Deluge, modifying its state machine and state transitions, while also adding a new forwarding state. Aqueduct offers a practically useful code propagation protocol for heterogeneous WSNs that has been evaluated over a real testbed and shown to incur significantly lower overhead than Deluge.

## Bibliography

- [1] Crossbow notes, <http://www.xbow.com/>.
- [2] Hp openview change and configuration management - <http://www.managementsoftware.hp.com/solutions/ascm/index.html>.
- [3] Ibm tivoli configuration manager - <http://www-306.ibm.com/software/tivoli/products/config-mgr/>.
- [4] James reserve extensible sensing system, <http://www.cens.ucla.edu/eoster/tinydiff/>.
- [5] Microsoft systems management server - <http://www.microsoft.com/smsserver>.
- [6] Telos sensor nodes, <http://www.moteiv.com/>.
- [7] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: System support for multimodal networks of in-situ sensors. In 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA), 2003.
- [8] R. Shea E. Kohler C. Han, R. Kumar and M. Srivastava. A dynamic operating system for sensor nodes. In To appear in Proc. 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys), 2005.
- [9] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, pages 1–12, New York, NY, USA, 1987. ACM Press.
- [10] A. Dunkels, B. Grnvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In In Proceedings of the First IEEE Workshop on Embedded Networked Sensors, 2004.
- [11] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. Complex behavior at scale: An experimental study of low-power wireless sensor networks, 2002.
- [12] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks, 1999.

- [13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In ACM Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 93–104, 2000.
- [14] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems, pages 81–94, New York, NY, USA, 2004. ACM Press.
- [15] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In Mobile Computing and Networking, pages 56–67, 2000.
- [16] S. Mishra J. Deng, R. Han. Countermeasures against traffic analysis attacks in wireless sensor networks. Technical Report CU-CS-987-04, University of Colorado at Boulder, December 2004.
- [17] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In Secom '04: Proceedings of the The First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, pages 25–33, 2004.
- [18] Sneha Kumar Kasera, Gísli Hjálmtýsson, Donald F. Towsley, and James F. Kurose. Scalable reliable multicast using multiple multicast channels. IEEE/ACM Transactions on Networking, 8(3):294–310, 2000.
- [19] R. Kumar, V. Tsatsis, and M. Srivastava. Computation hierarchy for in-network processing. In Proceedings of the Second International Workshop on Wireless Networks and Applications, San Diego, CA, Sep 2003.
- [20] Brian Neil Levine and J. J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. Multimedia Systems, 6(5):334–348, 1998.
- [21] P. Levis. Tossim: Accurate and scalable simulation of entire tinyos applications. In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003), November 2003.
- [22] P. Levis and D. Culler. Mate: a virtual machine for tiny networked sensors. In ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2002.
- [23] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation. In Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI), 2004.
- [24] Ting Liu and Margaret Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 107–118, New York, NY, USA, 2003. ACM Press.
- [25] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In WSNA, Atlanta, GA, September 2002.

- [26] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In WSNA, pages 60–67. ACM Press, 2003.
- [27] Niels Reijers and Koen Langendoen. Efficient code distribution in wireless sensor networks. In WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications, pages 60–67, New York, NY, USA, 2003. ACM Press.
- [28] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. Technical Report TR98-1687, 28, 1998.
- [29] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems, pages 188–200, New York, NY, USA, 2004. ACM Press.
- [30] S.S.Kulkarni and Limin Wang. Mnp: Multihop network reprogramming service for sensor networks. In 25th International Conference on Distributed Computing Systems (ICDCS), Columbus, OH, June 2005.
- [31] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks, 2003.
- [32] v Robbert and B. Kenneth. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining, 2001.
- [33] Matt Welsh. Exposing resource tradeoffs in region-based communication abstractions for sensor networks. SIGCOMM Comput. Commun. Rev., 34(1):119–124, 2004.
- [34] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation, March 2004.
- [35] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In To appear in Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS), April 2005.
- [36] J. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. In Proceedings of First IEEE International Workshop on Sensor Network Protocols and Applications, Anchorage, AK, May 2003.