A FLEXIBLE APPLICATION LAYER APPROACH TO DATA AGGREGATION

A Thesis by

Andrew Chandler Stanton

Bachelor of Engineering, Wichita State University, 2010

Submitted to the Department of Electrical Engineering and Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

July 2014
The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science, with a major in Computer Networking.

____________________________________
Vinod Namboodiri, Committee Chair

____________________________________
James Steck, Committee Member

____________________________________
Preethika Kumar, Committee Member
DEDICATION

To my father, who went before me, and my wife, who stays beside me
ACKNOWLEDGMENTS

I would like to thank my adviser, Vinod Namboodiri, who waited patiently for me to complete this work. Thank you for challenging and supporting me through each difficulty we encountered in this process. I have come to appreciate what research is because of you.

I am grateful for James Steck and Preethika Kumar for volunteering to be on my committee and also for those who were present at my defense including Kate Kung-McIntyre. Thank you for your encouragement and time investment.

I would also like to thank Ravi Pendse, Amarnath Jasti, Nagaraja Thanthry, and Vijay Ragothaman for giving me the opportunity to work in the Advanced Networking Research Institute at Wichita State University. It was there, through their mentorship, encouragement, and testing, that I received the gift of understanding computer networking.

I would like to thank six of my students from the Advanced Networking Research Institute for their gift of computing power which helped my simulations run quickly. Thank you, Deepak Palani, Gayathri Chandrasekaran, Jahnavi Burugupalli, Srikanth Chaganti, Tilak Shivkumar, and Yashwanth Bommakanti. I will never forget any of my students from the center. I cannot thank every one of you enough for inspiring me every day.

My appreciation goes to those who participated in my live Internet experiment including Babak Karimi, Dennis Sserubiri, Dileep Reputi, Jacob Bolda, James Bendowsky, Jenice Duong, Matthew Hahn, Neha Agarkar, Pushkar Waichal, Sai Srivastava, and everyone else who participated. My appreciation also goes to Trevor Hardy for his support and mentorship.

Much gratitude goes to my father and mother who left this world having sacrificed so much for me. And to all of my family, may we find what we need to heal. Lastly, I am in much debt to my wife for her contributions and patience in this three year venture.
ABSTRACT

Data aggregation techniques traditionally utilize cross-layer optimizations, which prove challenging to implement and are applicable only to a few related scenarios, while each new scenario requires a new approach. Proposed here is a framework that brings data aggregation to the application layer of the network stack while providing abstractions and defining interactions to make the framework flexible and modular. This framework, known as the data aggregation tunneling protocol (DATP), will make the benefits of aggregation attractive for many-to-one communication systems. The framework is modularized into the aggregation protocol, tree, function, scheduler, and end applications. DATP is considered in order to support the scalability of applications enabled by the smart grid. A complete DATP module was developed for the NS-3 network simulator and then tested in a smart grid distribution feeder scenario with optimal aggregation system parameters to determine the advantage gained by the system. Results show significant benefits for the aggregation system across a number of factors including the average message delay. Also developed was a data aggregation Internet application, which demonstrated the versatility of the application layer approach through an experiment in which thirty Internet-connected hosts joined the aggregation system and sent messages to a collector that gathered statistics about the system.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>The Smart Grid</td>
<td>1</td>
</tr>
<tr>
<td>Smart Grid Two-Way Communications</td>
<td>1</td>
</tr>
<tr>
<td>Communication and Networking Architectures</td>
<td>2</td>
</tr>
<tr>
<td>Communication Systems for Distribution Feeders</td>
<td>2</td>
</tr>
<tr>
<td>Data Convergence in Distribution Grids</td>
<td>3</td>
</tr>
<tr>
<td>2. LITERATURE REVIEW</td>
<td>5</td>
</tr>
<tr>
<td>Techniques</td>
<td>5</td>
</tr>
<tr>
<td>Wireless Sensor Networks</td>
<td>6</td>
</tr>
<tr>
<td>Smart Grid</td>
<td>8</td>
</tr>
<tr>
<td>Internet Protocol</td>
<td>10</td>
</tr>
<tr>
<td>Other Related Work</td>
<td>12</td>
</tr>
<tr>
<td>3. DATA AGGREGATION TUNNELING PROTOCOL</td>
<td>14</td>
</tr>
<tr>
<td>Aggregation Protocol</td>
<td>15</td>
</tr>
<tr>
<td>Roles</td>
<td>15</td>
</tr>
<tr>
<td>Messages</td>
<td>17</td>
</tr>
<tr>
<td>Sender and Receiver</td>
<td>22</td>
</tr>
<tr>
<td>Security</td>
<td>22</td>
</tr>
<tr>
<td>Aggregation Tree Controller</td>
<td>23</td>
</tr>
<tr>
<td>Aggregation Function</td>
<td>24</td>
</tr>
<tr>
<td>Aggregation Scheduler</td>
<td>25</td>
</tr>
<tr>
<td>Aggregation Applications</td>
<td>26</td>
</tr>
<tr>
<td>4. SIMULATION MODEL</td>
<td>27</td>
</tr>
<tr>
<td>Network Simulator 3</td>
<td>27</td>
</tr>
<tr>
<td>Discrete-Event Network Simulation</td>
<td>28</td>
</tr>
<tr>
<td>Simulation Core and Architecture</td>
<td>29</td>
</tr>
<tr>
<td>DATP Module</td>
<td>30</td>
</tr>
<tr>
<td>Aggregator</td>
<td>32</td>
</tr>
<tr>
<td>Function</td>
<td>33</td>
</tr>
<tr>
<td>Scheduler</td>
<td>34</td>
</tr>
<tr>
<td>Usage</td>
<td>35</td>
</tr>
<tr>
<td>Output</td>
<td>36</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (continued)

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. FEEDER SIMULATION</td>
<td>38</td>
</tr>
<tr>
<td>Development</td>
<td>38</td>
</tr>
<tr>
<td>Nodes</td>
<td>38</td>
</tr>
<tr>
<td>Topologies</td>
<td>39</td>
</tr>
<tr>
<td>Wireless Communications</td>
<td>42</td>
</tr>
<tr>
<td>Routing Protocols</td>
<td>42</td>
</tr>
<tr>
<td>Trials</td>
<td>45</td>
</tr>
<tr>
<td>Results</td>
<td>48</td>
</tr>
<tr>
<td>Maximum Delay Tuning</td>
<td>48</td>
</tr>
<tr>
<td>Minimum Delay Tuning</td>
<td>50</td>
</tr>
<tr>
<td>Delay in Aggregation Modes</td>
<td>52</td>
</tr>
<tr>
<td>6. EXPERIMENT</td>
<td>55</td>
</tr>
<tr>
<td>Internet Aggregation System</td>
<td>55</td>
</tr>
<tr>
<td>Results</td>
<td>56</td>
</tr>
<tr>
<td>7. CONCLUSION AND FUTURE WORK</td>
<td>61</td>
</tr>
<tr>
<td>Conclusion</td>
<td>61</td>
</tr>
<tr>
<td>Future Work</td>
<td>63</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>65</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>70</td>
</tr>
<tr>
<td>A. NS-3.16 AODV CODE CHANGES</td>
<td>71</td>
</tr>
<tr>
<td>B. NS-3 DATP FILE VERSION TRACKING COMMIT LOGS</td>
<td>72</td>
</tr>
<tr>
<td>C. FEEDER TOPOLOGIES</td>
<td>76</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Data Identity</td>
<td>18</td>
</tr>
<tr>
<td>2. Header Field Flags</td>
<td>19</td>
</tr>
<tr>
<td>3. Condensed Header Field Flags</td>
<td>20</td>
</tr>
<tr>
<td>4. Statistical Output Data Points</td>
<td>37</td>
</tr>
<tr>
<td>5. Selected Network Graph Topologies</td>
<td>46</td>
</tr>
<tr>
<td>6. Application Attribute Settings</td>
<td>46</td>
</tr>
<tr>
<td>7. Parameters of Simulation Trials</td>
<td>47</td>
</tr>
<tr>
<td>8. R4a Maximum Delay in Scheduler-Only Mode</td>
<td>49</td>
</tr>
<tr>
<td>9. R2a Minimum Threshold in Full Aggregation Mode</td>
<td>51</td>
</tr>
<tr>
<td>10. Collector Statistics Gathered from 21:00 to 21:59 October 17, 2013</td>
<td>58</td>
</tr>
<tr>
<td>11. Integration of Data Aggregation Components with Applications</td>
<td>63</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1.</td>
<td>DATP system.</td>
</tr>
<tr>
<td>2.</td>
<td>Message example</td>
</tr>
<tr>
<td>3.</td>
<td>NS-3 simulator architecture.</td>
</tr>
<tr>
<td>4.</td>
<td>DATP class hierarchy.</td>
</tr>
<tr>
<td>5.</td>
<td>Aggregator methods.</td>
</tr>
<tr>
<td>6.</td>
<td>Aggregator component linking pseudocode.</td>
</tr>
<tr>
<td>7.</td>
<td>Aggregator modes.</td>
</tr>
<tr>
<td>8.</td>
<td>DATP function pseudocode.</td>
</tr>
<tr>
<td>9.</td>
<td>DATP scheduler pseudocode.</td>
</tr>
<tr>
<td>10.</td>
<td>DATP helper pseudocode</td>
</tr>
<tr>
<td>11.</td>
<td>Graph algorithm properties and examples.</td>
</tr>
<tr>
<td>12.</td>
<td>Wireless network communication failure.</td>
</tr>
<tr>
<td>13.</td>
<td>Collector’s maximum delay performance.</td>
</tr>
<tr>
<td>14.</td>
<td>Aggregators’ maximum delay performance.</td>
</tr>
<tr>
<td>15.</td>
<td>Collector’s minimum delay performance.</td>
</tr>
<tr>
<td>16.</td>
<td>Aggregators’ minimum delay performance.</td>
</tr>
<tr>
<td>17.</td>
<td>Collector’s average message delay.</td>
</tr>
<tr>
<td>18.</td>
<td>Collector’s total received messages over total applications sent.</td>
</tr>
<tr>
<td>19.</td>
<td>Aggregators’ total packets sent.</td>
</tr>
<tr>
<td>20.</td>
<td>Aggregators’ total bytes sent.</td>
</tr>
<tr>
<td>21.</td>
<td>Aggregation system statistics.</td>
</tr>
<tr>
<td>22.</td>
<td>Aggregation system state</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>23.</td>
<td>Map of aggregation system participants</td>
</tr>
<tr>
<td>24.</td>
<td>Average message delay reduction over forward mode aggregation</td>
</tr>
<tr>
<td>25.</td>
<td>Aggregation tree of topology r1a</td>
</tr>
<tr>
<td>26.</td>
<td>Aggregation tree of topology r1b</td>
</tr>
<tr>
<td>27.</td>
<td>Aggregation tree of topology r2a</td>
</tr>
<tr>
<td>28.</td>
<td>Aggregation tree of topology r3a</td>
</tr>
<tr>
<td>29.</td>
<td>Aggregation tree of topology r4a</td>
</tr>
<tr>
<td>30.</td>
<td>Aggregation tree of topology r4b</td>
</tr>
<tr>
<td>31.</td>
<td>Aggregation tree of topology r4c</td>
</tr>
</tbody>
</table>
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AARFCD</td>
<td>Adaptive Auto Rate Fallback with Collision Detection</td>
</tr>
<tr>
<td>ABC</td>
<td>Abstract Base Class</td>
</tr>
<tr>
<td>AODV</td>
<td>Ad Hoc On-Demand Distance Vector</td>
</tr>
<tr>
<td>DATP</td>
<td>Data Aggregation Tunneling Protocol</td>
</tr>
<tr>
<td>DRF</td>
<td>Distance Reduction Factor</td>
</tr>
<tr>
<td>HFF</td>
<td>Header Field Flag</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>OLSR</td>
<td>Optimized Link State Routing</td>
</tr>
<tr>
<td>RFC</td>
<td>Request for Comments</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Although the result of this research is a data aggregation protocol suitable for any many-to-one communication system, it came into existence by first observing the data aggregation problem for smart grids. Consequently, the topic of data aggregation will be introduced from the perspective of the smart grid.

**The Smart Grid**

The power grid that exists today is undergoing a major innovative effort by integrating communications technology throughout various systems that have historically remained isolated. Because power grid systems have been isolated there have been problems with controlling increasing peak loads, recovering from outages, and integrating new energy systems into the grid. Establishing communications between consumers and utilities will offer many opportunities to solve these growing problems.

**Smart Grid Two-Way Communications**

Bi-directional communications are the key component to transforming the power grid into a smart grid [1]. By bringing communications technology into the grid, disparate systems are able to interact, and new applications can be developed to improve the capability of the power grid. Examples of these applications are smart meter systems and automation control systems. Smart meters enable the real-time transmission of energy use and become the logical communications point between the customers’ environment and the utilities’ environment, where now the customer may agree to let the utility exercise some control over the customer environment and vice versa. The customer may have better control of how energy is used and may even sell back some energy.
Communication and Networking Architectures

Since the power grid and communications infrastructures already exist, is it as simple as connecting them together to create the smart grid? This is not advisable because there would be significant risk from a cyber-security standpoint. In order to overlay the grid with new communication networks, devices must be deployed into segments of the electrical grid, such as from houses through distribution feeders to substations or from substations through transmissions grids to generators. In putting a network in place, either the exact demands on the network made by the applications should be known, or there must be a scalable solution. More likely, it is the latter, based on the history of the Internet and how it has grown time and again beyond its original intention. In the smart grid, there may be hundreds, thousands, or tens of thousands of devices connected together, so network architectures will need to be scalable as the number of applications and devices grows. It could be utilities or service providers that deploy, maintain, and operate these systems.

Communication Systems for Distribution Feeders

Distribution feeders deliver energy to businesses and homes. They are the primary focus of this work when observing smart grid scenarios. A multitude of wired and wireless technologies allow the utility to take advantage of the physical infrastructure of the distribution feeder in order to establish a communication network by connecting devices to the feeder, which in turn provides a power source from which to operate. Multiple technologies may be interconnected to break down network size and scale for appropriate capability. There could be home area networks, meter communication networks, and backhaul communication networks. The utility could deploy a smart meter that gathers data from smart appliances through the customer’s home area network using one technology and then sends the data back to the utility.
through another technology that the meters use to communicate toward a collection point. The utility can use yet another technology to backhaul data from collection points to datacenters. For example, Westar Energy reports through its Smart Grid Recovery Act program deployed in Lawrence, Kansas, whereby their smart meters communicate through a short-range wireless network to collection points that then utilize a long-range wireless carrier to deliver data to the utility [2].

**Data Convergence in Distribution Grids**

When there are a number of geographically distributed devices that must transmit across many hops to reach some collection point, a number of problems begin to form. Communication links closer to the collection point become saturated as data from all of the devices on the network converge towards the collector. This is a commonly studied and understood issue in wireless sensor networks. In the case of a single distribution feeder that connects to hundreds of homes and other devices, as the packets from the remote locations join with packets from all other nodes, a bottleneck forms before reaching the collector. This is where data aggregation techniques are utilized, because without changing how the network is designed, the network can become more efficient by reducing header overhead or by summarizing information. Aggregation comes at the expense of intentional delay and, depending on the summarization technique, can result in loss of information or metadata.

In the wireless sensor network field where data aggregation has been thoroughly studied, a pattern of defining aggregation techniques that are optimized only for a specific environment, application, or technology has emerged. Often, the technique involves interfacing with the network stack, which is the suite of software and hardware on a device, comprised in layers, which supports communications all the way from the application layer to the physical layer.
where data is transmitted onto a network. This is referred to as a cross-layer solution due to the creation of interfaces between different layers that typically do not have knowledge of one another.

In this work, a complete approach to data aggregation has been developed. This approach attempts to generalize the framework for aggregation so that it can be implemented in diverse smart grid environments and other many-to-one communication environments. First, existing aggregation techniques from the fields of wireless sensor networks, smart grids, and Internet communications will be reviewed. Then the data aggregation tunneling protocol (DATP) will be described, followed by a description of the protocol model that was developed in the NS-3 network simulator and the feeder simulations that were conducted to test the protocol. Finally, information will be provided about an experiment that was conducted to test the viability of a data aggregation system working across the Internet.
CHAPTER 2
LITERATURE REVIEW

Techniques

Data aggregation techniques with a multitude of applications are considered in many fields. Two major areas are aggregation of data stored in databases and aggregation of data being transmitted or moving through a network. The latter is known as in-network aggregation and is the focus of this work. It traditionally involves distributed systems sending and aggregating information as data approaches a collection point, or collector. Wireless sensor network (WSN) research is often focused on decreasing communication costs in order to increase the lifetime of the energy-constrained sensor network that delivers data to a collector. Data aggregation and data compression are viable techniques that are often studied [3]. Data compression can be differentiated from data aggregation by where the data is reduced and how the data is reduced. In compression, the data is reduced by the originator using an algorithm that likely may be reversed at a later time, whereas in aggregation, held data is reduced in transit by summarizing data from multiple sources. Compression involves the encoding of information into a smaller size, whereas aggregation summarizes the information, resulting in some expected information loss. The most common aggregation functions include adding up data values (sum), keeping only the highest value (max), taking an average of the value, or removing duplicate values. Some approaches do not modify the data at all but rather retransmit multiple sets of received data at once. This is often referred to as message concatenation or packet aggregation. Another component of aggregation is the holding of data and scheduling when the data should be released by the aggregator. If data is not held, multiple data sources cannot be acted on together.
A survey of data aggregation techniques in WSNs has been presented in the work of Fasolo et al. [4] and Rajagopalan and Varshney [5]. In-network aggregation is defined by Fasolo et al. [4] as “the global process of gathering and routing information through a multi-hop network, processing data at intermediate nodes with the objective of reducing resource consumption (in particular energy), and thereby increasing network lifetime.” Both of these works review protocols like LEACH, EADAT, and PEGASIS, among others. Most protocols are distinguished by the path the data takes to reach the collector. LEACH is cluster based, so there are specific cluster nodes that receive data from the source nodes and aggregate that data before sending it directly to the collector. EADAT is tree based, which is different from cluster based because many aggregators may be present in the path and therefore could form a tree with many branches of aggregators and sources. PEGASIS is chain based, and there is only a single path to the collector. The cross-layer optimizations that occur with these protocols are that they usually give priority to some path based on the energy of received signals or the energy present on the sensor in order to increase the lifetime of the network. They also bind the entire data aggregation methodology into the protocols’ individual approach, while at the same time creating dependencies on the specific use cases.

The flow of communication in WSNs is described by Westhoff et al. [6] as a reverse multicast flow of traffic. This is similar to the information sharing or communication pattern of many-to-one [7]. Many-to-one is a common design in WSNs resulting from the homogeneous nature of the sensors and the overall application-specific nature of the typical WSN. Other patterns of information sharing are one-to-one and one-to-many. Multicast is considered one-to-many because a multicasting device is able to share information with many other devices and do
so efficiently by enabling the source device to send only one copy of the data and enabling the network to split that data at appropriate branches into multiple copies. WSNs share information in the exact opposite manner; thus, reverse multicast is an appropriate term to describe creating efficiency in many-to-one communications, and makes for an important concept to understand when designing an efficient structure in many-to-one communications. Instead of data being split and copied in multicast, the data is combined and cut in reverse multicast. Instead of branching out from the source, data branches in towards the destination collector. Instead of the bottleneck being at the source, it is at the destination. The main focus in the work of Westhoff et al. [6] is on the confidentiality of data in the WSN through concealed data aggregation in order to protect against eavesdropping as well as compromised nodes and keys. The framework for the data aggregation component itself does not consider scheduling. However, the security aspects of the work include the use of privacy homomorphisms, which enable end-to-end encryption in the data aggregation scheme and still allow some aggregation functions to be applied to the data even though it is concealed, such as summing or multiplication. Another advantage is that aggregator nodes do not have to store sensitive keys. Other work on concealed data aggregation in WSNs includes that of Mlaih and Aly [8], and this topic is surveyed in the work of Sang et al. [9].

A problem with all WSN aggregation techniques is that they are motivated by the application-specific nature of the WSN. Since, the application of the WSN is very specific and singular (unlike the Internet), a layer-abiding solution is not considered, which makes the optimizations costly to implement and of limited scope. Different WSNs have different applications, hardware, and protocols, so all of these optimization solutions must be taken into consideration for each new WSN. Niculescu [3] states that “One of the main aspects of sensor
networks is that the solutions tend to be very application-specific. For this reason, a layered view like the one used in OSI [open systems interconnection] imposes a large penalty, and implementations more geared toward the particular are desirable.” The penalty of the layered approach lies in how deeply the optimizations can be considered. Solutions often use cross-layer optimizations, where, for example, if some state of the medium access control (MAC) layer can be touched or read by the application layer, then some improvement can be made to the solution. For example, Yu et al. [10] define a complex scheduling process for data aggregation that requires only specific time slots for the node to send data so they can all get on the same schedule. In layered networking, the application layer does not control when the data leaves a physical interface; neither does the transport protocol or even parts of the MAC layer (MAC layer can be very complex in wireless systems). Thus, the problem is that implementing these complex optimizations becomes costly in modification and may interfere with the design of the stack.

**Smart Grid**

A subset of WSN research applies data aggregation to the smart grid field. Research in this area is directed at providing data security where inherently data aggregation exposes data to every node involved in the system. If data is encrypted or authenticated by a single node in a traditional end-to-end manner, then the data cannot be aggregated at each hop because intermediate nodes do not have access to modifying the data. Data encryption in a hop-by-hop manner has a higher risk of vulnerability due to the increase of devices involved in the encryption of carried data [11]. Proposed solutions aim at concealing data in a manner so that each hop is able to encrypt and enfold its own data within an existing encrypted message without being able to decrypt the data.
Homomorphic encryption systems have the property of being able to apply a limited set of arithmetic operations on encrypted data so that an aggregator can sum two encrypted messages and yet not be able to decrypt the messages. Saputro and Akkaya [11] explored the latency effect of hop-by-hop homomorphic encryption on a data aggregation system. The considered data aggregation system was composed of a pre-formed multi-level tree and pre-exchanged keys. The collector would send a query to the nodes to begin transmitting data, and then the collector would wait for all encrypted and aggregated data from the nodes to arrive.

Similar research on homomorphic encryption has been done. In the work of Li et al. [12], all nodes participate in aggregation, and more is done to define a general approach to how the aggregation tree would be formed, whereas in the work of Saputro and Akkaya [11] only some nodes participate in data aggregation. It seems that in both works, data transmissions are sent synchronously from the nodes, so that an intermediate aggregator schedules its transmission by waiting for the corresponding data message from child nodes before it sends its own message. A decentralized security framework for data aggregation [13] introduced the integration of access control requirements into the smart grid security framework to ensure that the encryption system remains secure and only requires parties to receive authorized access to smart grid data.

Other work in data aggregation attempts to define specific approaches for aggregation in a smart grid environment. Lu and Wen [1] focused on the development of a distributed algorithm for the formation of an aggregation tree, whereas Bouhafs and Merabti [14] focused on the aggregation functions that could be applied to smart grid data.

The goal in each of these works is to focus on one component of the aggregation system, whether it be the tree, function, or scheduling of data. In some cases, other aggregation components are not even defined, leaving questions about how the component was implemented.
If a common protocol for aggregation was defined, then more works could reuse existing solutions for components instead of reinventing them.

**Internet Protocol**

Data aggregation research has mostly focused on WSNs and the smart grid. However, there has been some effort in the past to develop aggregation systems for the Internet. Perhaps the earliest form of aggregation ever considered for the Internet was the multiplexing protocol, designed and published as an Internet engineering note in 1979. It described how to combine multiple units from higher-layer protocols into a single unit of a lower-layer protocol [15]. It was later revised as the transport multiplexing protocol with a more limited scope in *Request for Comments* (RFC) 1692. It was designed to combine small transport packets from multiple applications running between the same server and host pair [16]. This enhancement was thought to be necessary to reduce the network load of applications like telnet and rlogin, because the work done by an Internet node to process large packets was nearly the same as small packets. The stated problem was correct, but the proposed solution never took hold.

Gathercast, a programmable aggregation mechanism, was designed [17, 18] and first published in 1998 [18]. This mechanism reduces the number of packets going through the network in an application-transparent manner. The main benefit highlighted in the work is the reduction of processing demands on routers, and the main application scenario is the combining of multiple acknowledgement packets returning to a web server from the Internet.

The Gathercast mechanism is based on the previous work of the authors, called transformer tunnels [19]. A transformer tunnel is established between two nodes, and transformer functions can be applied to packets that should be sent through the tunnel, like encryption or compression. The aggregation of packets in previous work [17, 18] was an
extension of transformer tunnels by introducing a new function. For example, consider a web server that handles thousands of connections from the Internet. The web server may act as a transformer endpoint and attempt to establish a tunnel with many Internet or internal routers who see packets destined to this web server. The routers with tunnels may then aggregate the packets destined to the server into a new packet with the source address of the router and destination address of the web server, with the aggregated packets as a payload. Benefits include a reduction in load on routers, packet loss, link-layer overhead, link contention, and interrupt-processing overhead on the server. The disadvantages of Gathercast mentioned include increased consequence of dropped Gathercast packets, interfering with reliable transport protocol mechanisms, and some other special situations where performance gains were not observed.

The Gathercast mechanism was simplified in the work of He et al. [20], which essentially said that any router can act as an aggregator for any destination based on its own observations of destination patterns of the packets and can essentially choose to aggregate and send packets through a Gathercast tunnel to the identified destination. What is assumed, but not clearly addressed, is that every destination would have to support the Gathercast mechanism.

Another approach at packet aggregation is presented as JumboGen [21], which performs aggregation within service provider domains where packets entering from one area of the domain are combined, if, based on the routing table, it is known that these packets will exit the domain at a common place. Packet aggregation techniques are usually justified in these papers by the need to reduce packet processing overhead of routing nodes in high packet per second environments like the Internet.

Concast is another example of an aggregation network service using Internet protocol (IP) for Internet applications [22, 23]. Concast approaches aggregation from a reversed view of
the established IP multicast group service for one-to-many communications. Unlike Gathercast, Concast is not application-transparent. The application must be written to use the Concast address, and the network must support Concast at the same time. Senders use the receiver’s unicast IP address as the destination address, so the packets are routed in a unicast fashion toward the collector. The forming of Concast trees is started by senders who send a signaling message that tells Concast nodes or routers along the path of the collector to participate. Two Concast merging models are defined as simple and custom. Simple Concast is based on the inversion of multicast payloads; therefore, instead of duplicating packets, it removes duplicate packets so that only one copy is sent to the collector. Custom Concast is a flexible model that should be designed according to the needs of the collector and the applications that will use it. The stated disadvantages seem to be that it does not provide good reliability mechanisms, which is also true of multicast, and it is defined as best effort. Concast may require a large amount of state resources on routers. Concast may not be effective in partial-deployment scenarios, and it is not inherently a part of the current Internet stack.

None of these aggregation systems have become well-known on the Internet. All of them mirror the same problems found in WSN and smart grid data aggregation solutions. They require network layer adjustments and insertion of cross-layer interactions that would have to change in different use cases.

**Other Related Work**

Other techniques try to solve the data-convergence problem without doing aggregation. Bistro [24] is an attempt to solve the problem by coordinated approaches. The basis of this work is that between all source nodes sending to a common destination collection node, better performance can be found through the Internet by indirectly transferring data through some set of
the nodes in order to avoid bottlenecks in regions or links in the Internet. This problem is at last solved with an application layer approach. Furthermore, Cheng et al. [24] point out that methods like Gathercast and Concast do not operate at the application layer and that the Internet never adopted these protocols. Another similar system is considered specifically for the smart grid environment [25], although it could have a wider scope because of the simplicity of the solution. The premise is that thousands of nodes in a smart grid environment that need to reliably transmit data toward one central server could do so more efficiently by indirectly transferring data messages through a reliable transport protocol to neighbors, so that the reliable connection is between each hop, rather than end to end. In a reliable transport protocol, messages that are lost by the network are retransmitted by the source.

An application layer approach, such as those investigated previously [24, 25], is needed for data aggregation to gain better traction, but both of these approaches stopped at indirect transfer. Indirect transfer is a core characteristic of a data aggregation system that behaves as an application. It involves applications sending data directly to an aggregator node instead of to the collector, and it enables the aggregation system to take control of the content and delivery of the data.
CHAPTER 3
DATA AGGREGATION TUNNELING PROTOCOL

Objectives of the data aggregation tunneling protocol framework are modularity and flexibility. Data aggregation with the DATP consists of the following components: protocol, tree, scheduler, function, and applications. For a broad use of the DATP, it is important that solutions to each component be interchangeable, which means that interfaces must be defined between the components. Figure 1 demonstrates how components of the DATP interact.

Aggregators pass messages onto each other until the messages reach the collector. When a message is received by an aggregator, it is first handed to the aggregation function. The aggregation function can pull existing messages that are held in the scheduler and apply functions to them with the new message. It then returns the result to the scheduler. The scheduler holds onto messages and must decide when a message is ejected and whether it is
ejected with other messages at the same time. Interestingly, in DATP, aggregation can be the result of either the scheduler or the function. Both components have the power to reduce the overall data transmitted by the system. The tree controller is responsible for joining the aggregation tree and communicating with other aggregators in the system. The rest of the section is dedicated to further defining each component and how it interacts with the others.

**Aggregation Protocol**

The protocol is the common communication language between all nodes involved in the aggregation system. Just like other protocols, all participating nodes must be able to abide by the requirements and constraints of the protocol. The major aspects of this protocol will be described, including roles of nodes in the system, message composition, sending and receiving of messages, and protocol security.

**Roles**

There are two roles for devices in an aggregation system: aggregator and collector. Other devices may be involved in the passing and routing of DATP messages, but they should not have to participate or be aware of the protocol functionality. Any application that needs to run over the DATP must be aware of the aggregation system, but only from a limited standpoint. For example, applications should be able to identify an aggregator and whether it is on the same device, on the same network, or elsewhere, and they should be able to send messages to the aggregator in the appropriate form. The DATP is considered to be a tunneling protocol because end applications inject messages into the aggregation system allowing for the aggregation system to assume control of the message rather than it being sent directly to the collection point. Once the message is injected into the system, it will be held by the system at different times and can be merged, deleted, mangled, or modified by the aggregators.
An aggregator is a device in the system that attempts to reduce data cost while minimizing the delay incurred when holding data. In order for aggregation to occur, two or more individual messages must be held on the same node. When this occurs, it is called a message collision. Unlike packet collisions, which occur on a medium where the packet is lost, these are a desirable kind of collision. There can be no benefits from aggregation without message collisions occurring. In order to facilitate collisions, aggregators must choose to temporarily hold onto messages they receive before releasing them. There may be any number of aggregators in the system. Chances for collisions tend to increase as messages get closer to the collector.

Aggregation functionality in the aggregator is handled by three subcomponents: function, scheduler, and tree controller. The aggregation protocol component is only responsible for handling the sending and receiving of messages. The aggregation protocol component has control of disabling the scheduler or the function. However, the scheduler should not be turned off if the function is not turned off, because the function relies on the scheduler for collisions to occur and is made useless if there is no scheduler holding messages. An aggregator that has both components disabled is said to be in a forwarding mode.

The collector is the end-destination point for all aggregation devices—the root of the aggregation tree. Data does not typically have to be consumed by the collector. The collector may apply a final function or reverse function to data and then send data to outside systems that consume, store, or process the aggregated data. This should be considered as an entirely separate system outside the scope of the DATP. There is only one collector in an aggregation system; however, any number of aggregation systems could be deployed in an environment.
**Messages**

All applications in the DATP send messages to the collector through aggregators. Messages are composed of a header and the end data, and are the individual units being worked on by the DATP. Streams and datagrams are the two main transport protocol models that handle the transportation of application data. A message is a single unit of application data. Messages are never seen as a stream or as a datagram in the DATP, although a message or messages will be transferred over a stream or be inserted into a datagram. For example, an aggregator may be open to receiving messages from its children in either a stream or a datagram protocol. If it receives a message in a stream, then it should transmit it to the next hop aggregator in a stream. If it receives a message in a datagram, then it should transmit the new message in a datagram. Thus, aggregators keep messages on separate transport paths through the system. This should prevent the scheduler or the function from moving a message from one transport path to another, even if messages from the same application are on both paths.

Data requires an identity to be relevant. In traditional network stacks, the data or payload is identified by the headers of information that encapsulate it. With those headers, a source and destination address can be delineated, even what kind of application for which the data is intended. The data carried by a packet will not have any relevance outside of those headers while it remains in the network. Only when the packet reaches its destination is the data relevant and the headers discarded, having lost their relevance. Message concatenation is one form of aggregation where the identity provided by the headers is lost as they are discarded while traversing the network and replaced with new headers of intermediate aggregators. In order to preserve the identity of data, the aggregation protocol must provide mechanisms for data to be
identifiable to the extent needed by the collection system. The various forms of identification typically provided by headers can be seen in Table 1.

### TABLE 1

**DATA IDENTITY**

<table>
<thead>
<tr>
<th>Identity</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location</td>
<td>source, destination, address, name, origin, service</td>
</tr>
<tr>
<td>Context</td>
<td>sequence, place, order, length, interfacing, type</td>
</tr>
<tr>
<td>Time</td>
<td>start, elapsed, end, frequency</td>
</tr>
<tr>
<td>Priority</td>
<td>labels, utility, deadline, marking</td>
</tr>
<tr>
<td>Reliability</td>
<td>acknowledgement, error checking, checksum</td>
</tr>
<tr>
<td>State</td>
<td>flags, codes, negotiation</td>
</tr>
</tbody>
</table>

Aggregation inherently causes identity loss, whether by sum or concatenation or any other method. For example, concatenation results in the loss of network stack headers, which could carry location and context identity. Careful consideration for what identities are kept with the data must be taken; otherwise, the advantage of aggregation is removed by introducing overhead back into the system. Flexible application headers are used to persistently identify the data, whereby most of the fields in the header are optional.

The key to the application header is making it flexible so that applications can add only the necessary identity information to the header for their messages. One field in the flexible header is called the header field flag (HFF), which determines other fields that are present in the header. If a field’s flag is turned on, then it is present in the header. The HFF is the only required field in the header. The arrangement of fields is based on the order of the enabled flags in the HFF. The processing order of the flexible header includes the HFF field first, then the size modifier(s), then any remaining fields in any order. For example, the checksum might be
checked next to validate for an error early on. A default size is associated with each field in the HFF.

Table 2 describes each field that the HFF references. A suggested default size is indicated for each field, but defaults could be different in separate aggregation systems. If a different size is needed by a particular application, then a size modifier field can indicate an adjusted size. There should be some room in the HFF for system-specific fields that the aggregation system or the collector may understand.

**TABLE 2**

**HEADER FIELD FLAGS**

<table>
<thead>
<tr>
<th>No.</th>
<th>Name of Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HFF</td>
<td>2</td>
<td>Indicates which fields below are present in the header</td>
</tr>
<tr>
<td>1</td>
<td>Size Modifier</td>
<td>1</td>
<td>Adjusts size of a specific field. Bits 0–3: field indicator; bits 4–6: new field size in bytes plus one; bit 7: indicates another size modifier is present; Same field can be indicated repeatedly and then adds to the field size</td>
</tr>
<tr>
<td>2</td>
<td>Origin</td>
<td>4</td>
<td>Could be address, ID, or even variable length name</td>
</tr>
<tr>
<td>3</td>
<td>Application</td>
<td>1</td>
<td>Identifies the application to which message belongs</td>
</tr>
<tr>
<td>4</td>
<td>Priority</td>
<td>1</td>
<td>Scheduler interprets what priority means</td>
</tr>
<tr>
<td>5</td>
<td>Timestamp</td>
<td>8</td>
<td>Time that message was sent from application</td>
</tr>
<tr>
<td>6</td>
<td>Data Length</td>
<td>1</td>
<td>Data length in bytes (maximum with default size is 255 bytes)</td>
</tr>
<tr>
<td>7</td>
<td>Sequence</td>
<td>4</td>
<td>Sequence number to indicate message order from application</td>
</tr>
<tr>
<td>8–14</td>
<td>System Specific</td>
<td></td>
<td>Fields dedicated for system-specific purposes; could be used for hashes, security fields, length of header, repeat field, delimiter character, system specific size modifier, or locked message</td>
</tr>
<tr>
<td>15</td>
<td>Another HFF</td>
<td>0</td>
<td>Indicates that multiple sources are sharing same message</td>
</tr>
</tbody>
</table>

If no system specific fields are necessary, then Table 3 shows an alternate configuration, which would only require the HFF to be a single byte.
<table>
<thead>
<tr>
<th>No.</th>
<th>Name of Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>HFF</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Origin</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Application</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Priority</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Timestamp</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Data Length</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Sequence</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>Another HFF</td>
<td>0</td>
</tr>
</tbody>
</table>

The most ideal of applications in the DATP may not even require any identity fields because no identity is required for the aggregation system to benefit from it. Perhaps the identity of the data is inherent in the data, or perhaps the application requires its own specific header that is outside of the DATP header and thus exists as data within the message.

If no application field is present in a flexible header, then an aggregator could assume a default application. The default application could be different for each transport protocol, or other ways could be used to identify the application. The application field is important because other internal components of the DATP, such as the function, must know what to do with each kind of application. An aggregation system that passes on messages where the application is not known is said to behave in an application pass-through mode. An example message with its flexible header and data is illustrated in Figure 2.
Some sanity checks would have to be employed to ensure that the HFF and the field values make sense. For example, neither the HFF field size nor the size-modifier field size could be modified by the size modifier. This would constitute an unrecoverable error. If such an error is found while processing a message header in a stream connection, then the connection must be reset. If it is in a datagram, then only that datagram must be discarded. Another important field is the data length, which must be provided in the header, or a delimiter must be set. If neither field is set, then a default size or delimiter could exist for each application. These defaults would be known by the aggregation receiver. The length of the header need not be indicated because it can be determined by the HFF and size modifiers present. If an aggregator fails to be able to identify the data length, then it is considered an unrecoverable error because the aggregator will not be able to determine when the next message’s HFF starts.

When data is traversing the aggregation system the ideal characteristics are that the data is small, periodic, related, sometimes duplicated, quantitative, and coming from many sources. These characteristics will increase the capability of performing some useful aggregation functions on the data. If the function is to modify the data, then the function will need to know where the data is in the message, how long it is, and with what application it is associated.

In order for the function and scheduler components to parse and make sense of the message header and data, the protocol receiver component should build a message descriptor that points to each field in the message and indicates its length. This will standardize an approach to processing the header, so that the functions and schedulers have a common way to read each message. The descriptor may also insert internal fields into the message, which will be discarded before the message is ejected from the aggregator. The message descriptor should include an internal identifier for the message, so if any functions are applied to multiple messages and fewer
messages are returned by the function, then the scheduler can recognize which message existed from the start. Another part of the message descriptor should be the time the message entered the aggregator and whether the message was received in a stream or in a datagram. When a flexible header is received without an application field, the application may be identified by the protocol receiver by some other means, and an internal application field can be inserted into the message by the receiver so other components can associate it with an application. Similarly, if a delimiter is known by the receiver to mark the end of a message, then the receiver can identify that delimiter and update the descriptor with the length of the data in the message.

**Sender and Receiver**

Every aggregator needs a sending and receiving element. It has already been stated that the receiver is responsible for receiving messages from child aggregators or applications, building message descriptors, and validating inbound messages. Also, the receiver is capable of adding flexible headers onto data-only messages received from applications that do not insert flexible headers.

The sender should actually do the work of transmitting messages that are ejected from the scheduler. The sender marks the destination as the next hop or parent aggregator that it learns through the controller. The sender and receiver protocol elements have much room for interaction with other components, but these are the basic requirements.

**Security**

If messages need to be protected in the DATP, the protocol should handle it, but protocol security mechanisms have not yet been developed. Implementations could introduce confidentiality and authentication for the data and or the headers. Fields in the HFF could be used for inserting an authentication digest. If only data needs to be protected, it may be the
applications using the aggregation service that encrypt the data. If homomorphic cryptosystems will be used, then intelligence must be built into the aggregation function to perform the crypto algorithms. The flexible header may be a key component to secure. However, the component that probably has the most influence over aggregation system security is the tree controller.

**Aggregation Tree Controller**

The aggregation tree is the structure that is formed by all participating aggregators and the collector, which represents the root of the tree. The controller is the module on the aggregator that is responsible for communicating with the tree. The tree ought to facilitate a stable, efficient, and flexible environment for all members in order to ensure the aggregation system provides an effective platform for applications. Stability means that messages are not often lost in transition while the tree changes formation and especially that aggregation loops would be prevented, which would bring parts of the system to a complete halt. An efficient tree prevents unnecessary message weaving where a message moves through a longer path. It may potentially be delayed more and would have to be retransmitted onto the network more often, though it may have less chances of collision with other messages. Finally, it is important that the tree can adjust to members joining and leaving the aggregation system in any state.

The main requirement of the controller is that it must update the protocol sender with the next hop or parent aggregator address to which messages are forwarded. The sender should always rely on commands from the controller to determine where to send messages. The controller may also interact with the protocol receiver or any other component of DATP. The entire aggregation tree could have a way to interact with applications in order to point the applications to the correct aggregator.
Aggregation Function

The aggregation function maintains instructions that dictate what operations are to be performed on headers and data in the messages for each application. This function has the capability to add, modify, or remove any data or header fields in messages. There could be many frameworks or rules for how these operations should be performed. Applying functions to message data means that the function component must be application aware and able to ensure that the integrity of the data is not compromised. A general rule is that the output of the function operations should result in a similarly mutable object as the input. Aggregation capability will be limited if a message can only undergo one operation.

As an example of a potential problem, consider a function that merges data between two messages by calculating the average of a value contained in each message and returning only one message with the data field containing the average. If this is all that is done, then the result has become immutable, because the resulting average will not be accurate if used as input to another average operation since context as to the number of values averaged has been lost. To solve this problem, a count of the number of values that have been averaged must also be included in the data. Then, the two data fields could be averaged and counted in another field, and the result would be a new message that would be mutable again. Another problem to solve is what to do with the flexible headers of two messages if the data components are merged into one message. Either flexible header could be discarded, or it is possible to insert a second flexible header associated with the same data, but the scheduler should only be expected to process the first flexible header in any message.

The objective of having results be mutable again highlights that there is a tradeoff between flexible operations versus aggregation benefits. If only one data value of several in two
messages is operated on, many mutability questions arise. Should the remaining data be dumped into one message or remain in two messages? Will the end application or the next function operation be able to parse the data again according to its defined procedures? Great intelligence that makes these kinds of decisions must be built into the functions so that an optimal and reliable result is achieved.

**Aggregation Scheduler**

The scheduler is a critical component to the aggregation system. It is the only component of the aggregator that holds onto messages. Its role is to decide when to attempt to forward them. The scheduler decides how much to delay a message and can use the information in the headers to help determine the delay time. The schedulers of all aggregators may coordinate with each other throughout the aggregation system. The scheduler also has a role in aggregating data, because it has the capability of ejecting multiple messages at the same time, thus serving as a message aggregator as it concatenates the messages together into one payload for the appropriate transport protocol to deliver.

The scheduler must make the held messages available to the function for access. While the scheduler has given a copy of a message to the function, it should lock the message so that it is not possible to prematurely eject the message before the function returns a response. It is not that the function may delay in returning the response, but there could be brief processing delays while the function performs its operations. Unless it is known the scheduler is blocked from taking any action, this is the safest design. Whenever the function receives a new message, the scheduler may not be aware of that new message until the function forwards it on to the scheduler. The function cannot hang onto the new message but must immediately assess if any operations should be applied to the existing messages with the new message. Generally, if two
messages are to be merged, the older message should be modified and the new message would be deleted by the function so that the scheduler would be able to keep track of how long it has held onto existing messages. The scheduler may never see the new message if it is merged with an existing message.

**Aggregation Applications**

Aggregation systems can support multiple applications at the same time. An application can inject a message into the aggregation system by sending the message to an aggregator. The application and the aggregator may or may not exist on the same node. If they are not on the same node, then the application must have a method to identify the appropriate aggregator. This could be accomplished by communicating with a known collector, broadcasting the request, or a method provided by the aggregation tree. The application can use any transport protocol that the aggregators supports.

Since the DATP is a system of applications working together and its implementation does not depend on network stack modifications, other applications that act outside of the aggregation system (e.g., some application choosing not to utilize the aggregation service) can still exist on the same nodes or other nodes. There could also be helper applications that assist in forming the tree, controlling the aggregation system, or querying applications to send data through the aggregation system. In this scenario it is feasible that the protocol receiver may intercept a DATP message that is intended for itself. Likewise, the scheduler or the function might intercept messages intended for themselves. Furthermore, they may also inject messages into the aggregation system to be passed forward in the direction of the collector, which might be a valuable method for a component to communicate to the same component in a parent aggregator.
CHAPTER 4
SIMULATION MODEL

A complete aggregation model was developed in the NS-3 network simulator. It was designed so that anyone could write custom components within the DATP framework and simulate them in an aggregation system. An attempt has been made to define how the components interface with one another so that implementers can be free to focus on finding the best way to do scheduling, aggregate data, or make two components work well together. Before going into the details of the model, a basic description of NS-3 is provided here.

Network Simulator 3

NS-3 is an open-source network simulation project written in C++. It provides a platform where researchers can simulate applications and networks based on current standards and emerging technologies. Many researchers have been using the previous version, NS-2, for some time, but NS-3 is being positioned to become the primary vehicle for driving academic network simulations. NS-3 is not an extension or transformation of NS-2, but was built to overcome NS-2 shortcomings. Improvements include real Internet models like providing IP addressing capabilities (IPv4 and IPv6) and supporting nodes with more than one interface. Also, support for generating standard packet captures exists. The 802.11 models are more extensive than NS-2 models, but NS-3 does not yet have as many models as NS-2. Simulations are defined primarily in C++, but Python bindings exist for most models, giving users the flexibility to define simulations in a scripting language. The simulation script is written to include the necessary modules and set up the scenario to be executed. Helper classes, such as allocating addresses to nodes, placing wireless nodes on a topology, or building layers of wireless models, assist the user to interface with these modules efficiently.
The simulation scripts typically create a number of nodes, place them in a network topology, set them up with a communication stack (like the Internet Protocol), and turn on some basic applications for a set time to send traffic through the network and collect output. After the script is parsed for all these items, a call to run the simulator is executed and the program runs through all of the events created by the modules running on the nodes.

**Discrete-Event Network Simulation**

NS-3 is a discrete-event driven simulator, which means that during the run time of the simulator, simulation processing is controlled by events that are scheduled to activate at known times. For example, in order to have an application send packets to a remote host every second, the application is scheduled to start at the desired time; then when that time is reached a call to send a packet is executed. At the end of that send routine, the application schedules another send event one second later. If the simulator has no other events scheduled, then the simulation time will immediately advance one second. This will continue in a loop until the application is scheduled to stop. If another module is scheduled to stop the node or stop the application, then the application would be able to cancel the future event so the cycle of sending packets every second is stopped.

There are no heterogeneous delays built into the simulator. These events are not happening in real time but rather in simulation time, so in certain scenarios, the lack of heterogeneous behavior results in unexpected or undesirable effects. In the simulator, this problem can play out when multiple applications are started at the exact same time. If 100 nodes schedule the send event in a shared physical medium at the same time, then the data link implementations of the nodes will all register that the medium is free and every node will transmit at the same time causing every packet to be lost.
**Simulation Core and Architecture**

The core of the NS-3 simulator includes base classes, time, timers, random variables, function callbacks, logging, and more. The three main base classes are ObjectBase, SmartRefCount, and Object, which is a subclass of the former two. The Object class has a wealth of base functionality that solves problems from NS-2 where the base classes cannot sufficiently provide many features that different subclasses need and so must implement themselves in different ways. Figure 3 depicts the architecture of the NS-3 simulator [26], including its major features and classes. Two of the main features are object aggregation and object attributes.

![Figure 3. NS-3 simulator architecture.](image)

Object aggregation is an important concept of the simulator. Classes that are inherited from Object or Object-Base receive the aggregation property that allows an object like a node to have all other objects that are created on top of it to be closely associated to the node and to be referenced through the node. A node could have a network device, an Internet stack, a routing protocol, and an application aggregated to it. Since all of these objects belong to the node, all of...
them can access other objects through the node. An application can access the IPv4 implementation on the node to become aware of the node’s own address. Object aggregation in NS-3 is based on the component object model.

The attributes system allows an object to be created with different values for components of the object that should be treated like variables. The values are easy to change, can be given defaults, and can be checked for compliance (e.g., minimum, maximum).

**DATP Module**

The DATP was written as a module in NS-3. In order to install the module in the NS-3 simulator, the directory with DATP code can be retrieved from the public repository [27] and copied into the source directory of the installed NS-3 version. Then NS-3 needs to be reconfigured and rebuilt according to instructions in the NS-3 manual. Components of the DATP are broken into different classes, which are shown in Figure 4.

```
+-- Application (ABC)
   |   +-- DatpAggregator
   |   |   +-- DatpApplication (ABC)
   |   |   |   +-- DatpApplicationOne
   |   |   |   +-- DatpApplicationTwo
   |   |   |   ++-- DatpApplicationThree
   |   |   +-- DatpCollector
   |   |   +-- DatpTreeController (ABC)
   |   |       +-- DatpTreeControllerAodv
   |   ++-- Header (ABC)
   |       +-- DatpHeader
   |       +-- DatpGenericApplicationHeader
   ++-- Object (ABC)
      |   +-- DatpFunction (ABC)
      |       +-- DatpFunctionSimple
      |       +-- DatpScheduler (ABC)
      |       ++-- DatpSchedulerSimple
      +-- DatpHelper
      +-- DatpApplicationHelper
```

Figure 4. DATP class hierarchy.
Abstract base classes (ABCs) were created for the function, scheduler, tree controller, and applications. These classes cannot be used in simulations because they have pure virtual functions that must be implemented in derived classes. Simple derived classes are included for each component. The simple scheduler holds onto every message for a predefined maximum and minimum time. Messages are released earlier than the maximum time if another message has reached its maximum hold time and other messages have exceeded their minimum hold time. The simple function breaks the data segment of a message into unsigned four-byte integer units and executes a sum operation between the respective units of the new message and the existing message of the same application. The simple tree controller class accesses the ad hoc on-demand distance vector (AODV) instance of its node to determine what the next hop gateway is for the collector and passes it as the next hop to the aggregator.

A simple DATP application base class was also created. This base case takes care of creating the connection to the aggregator so derived classes can focus on what data and headers are sent in the message. The aggregator class implements the aggregation protocol functionality. It can be used as a base class but fully implements its methods since its code does not need to be changed for different schedulers, functions, or tree controllers. The collector class in the model simply does a statistical gathering of received messages. Another set of classes are the DATP flexible header and a generic DATP data header, which simply holds a single four-byte unsigned integer as data. The given flexible header implements the condensed HFF in order to become a one-byte field. Finally, the helper classes make it easy to implement the entire aggregation system with only a few lines of code in a simulation.

Now that a basic understanding of the various classes has been given, a deeper investigation of some of the components is necessary.
**Aggregator**

The aggregator’s most important role in the model is to create the aggregation components and link them together with the methods necessary for them to communicate with one another. Figure 5 shows what methods exist for each component to communicate with the next. In NS-3 simulation, this linking has been accomplished using the callback model, where the calling class sets a method to be called by the called class. For example, the function needs to query the scheduler to determine if there are any existing messages that can be operated on with a new message that the function has received. In this situation, the function is the calling class (calls the scheduler to perform a task). The scheduler is the called class (called by the function to perform a task). The function is set with a method of the schedulers that it can call whenever it is ready. Figure 6 shows a simplified code for how this linking is done in the aggregator class.

![Figure 5. Aggregator methods.](image)

```cpp
treeController->SetParentAggregatorCallback (DatpAggregator::SetParentAggregator)
aggregator->SetNextReceiverCallback (DatpFunction::ReceiveNewMessage)
scheduler->SetPacketEjectCallback (DatpAggregator::Sender)
scheduler->SetQueryResponseCallback (DatpFunction::ReceiveQueryResponse)
function->SetQueryCallback (DatpScheduler::ReceiveQuery)
function->SetNewMessageCallback (DatpScheduler::ReceiveNewMessage)
```

Figure 6. Aggregator component linking pseudocode.
The aggregator class creates three versions of the components (tree controller, function, and scheduler). The aggregator passes on knowledge of the collector address to the tree controller. The aggregator sets its own method to be called by the tree controller callback in order to receive updates about the correct parent aggregator. The aggregator sets the function’s method to be called when the aggregator sends it a new message. The scheduler is informed of the methods to call for responding to a query and for ejecting a packet for transmission back to the aggregator. The function is informed of the methods to call for sending a query to the scheduler, sending a new message to the scheduler, or sending an existing message back to the scheduler after it has been operated on. The aggregator can be informed to use any type of derived function class, scheduler class, or tree controller class to be used and these derived classes can be programmed with their own version of these methods for handling the callbacks. Furthermore, the aggregator can be set with an attribute that turns off the function component or the scheduler component. An error will occur if the scheduler is turned off without also turning off the function. Figure 7 shows the path of a message for the possible modes.

Figure 7. Aggregator modes.

Function

The functions job is to operate on new messages received from the aggregator with existing messages it finds out about from querying the scheduler. It should perform all
operations as soon as it can and return the results to the scheduler. Figure 8 illustrates the basic
task of the function for when it receives a new message from the protocol receiver. This could be more complex than the simplified description given here, but regardless of how the function is designed to operate on messages, it must interact with the scheduler and the receiver.

DATP Function ReceiveNewMessage
query scheduler with copy of new message header using callback
if received existing message from scheduler
   apply operation on existing message and new message
   send scheduler the modified existing message using callback
else
   send scheduler the new message using callback

Figure 8. DATP function pseudocode.

Scheduler

The scheduler implemented in the model does not really have to lock messages given back to the function, because of the linear execution environment of the simulator. When the function first queries the scheduler, all methods are called one-by-one to finish the task until the scheduler receives either a new message or an existing message. The function and scheduler work synchronously, whereas in real environments, they may be separate processes or entities on an aggregator altogether. After receiving a new message in the simple scheduler class given, the scheduler creates a timer to define the delay for the new message until it must be ejected. It may be ejected before that time if the timer for an older message has expired and the timer for the newer one has reached the minimum threshold. This logic is shown in Figure 9.

DATP Scheduler MessageTimerExpired
for each held message
   if message delay is greater than minimum threshold
      add message to packet
      remove message from database
      eject packet to sender

Figure 9. DATP scheduler pseudocode.
**Usage**

Helper classes are defined to make implementation of the aggregation system easy in a simulation program. One helper class creates and installs the collector and aggregator applications. The other helper class creates and installs applications. After the helper class installs an aggregator, it calls the install method of the aggregator, which creates the three other chosen components (function, scheduler, and tree controller). It is important to note that the aggregator, collector, and tree controller are all derived from the NS-3 application class. This was a much debated decision because they may or may not be perceived as an application. However, by defining them as applications, it enables better control of when they begin sending and receiving network communications and when they stop. More complex schedulers or functions may need the same capability but were not defined in this manner.

Figure 10 shows a simplified code for how an aggregation system can be implemented. The helper is set with specific types to use for the components, and then the NS-3 applications are installed on the collector and aggregator nodes. A similar process is done for the application helper, which enables any number of applications to be added to the desired nodes.

```c
DatpHelper datp
datp.SetAggregatorAttribute (tree controller type aodv)
datp.SetAggregatorAttribute (function type simple)
datp.SetAggregatorAttribute (scheduler type simple)
ApplicationContainer aggSystemApps = datp.Install (collector, aggregators)

DatpApplicationHelper datpApplication
datpApplication.AddApplication (application type 1)
datpApplication.AddApplication (application type 2)
ApplicationContainer aggApps = datpApplication.Install (aggregators)

aggSystemApps.Start (time1)
aggApps.Start (time2)
aggApps.Stop (time3)
aggSystemApps.Stop (time4)
```

Figure 10. DATP helper pseudocode.
Output

After the aggregation system was built with these classes, an example simulation script was created to test the system, identify bugs, and produce outputs suitable to identifying the behavior of the aggregation system. An in-depth study was conducted to identify measures that reveal the effectiveness of the entire aggregation system. Simulation outputs were divided into three areas, which all provide different viewpoints of the system. The single most critical node in the system is the collector, so statistics gathering was built into the collector. However, the collector represents a simple receiver having little to no visibility of the actual aggregation as it occurs in the system. It may not truly reflect the effectiveness of the aggregation system. The entire set of aggregators became the next gathering point. The final outputs were gathered from the applications so an idea of the input into the system could be identified.

The simple aggregation system was designed so that the scheduler in the aggregators and the collector would have some increased knowledge about previous aggregation functions applied to messages. For example, since the data in each message contained the count of the number of messages already merged, that data point was available to the collector so the collector could count the exact number of messages received. Also, header timestamps were updated when merged to include a weighted average of the end-to-end time delay so that the collector was able to perfectly calculate the end-to-end delay of all received and merged messages. The same was done to the internal receive time timestamp so that the scheduler was able to calculate the perfect holding delay of all sent and merged messages. These techniques greatly improved the capability to measure the performance of the data aggregation system for delay and delivery rates.
Table 4 provides a description of all data points that can be captured from the aggregation system. These data points can have different meanings, depending on the area in which they are captured. The collector logically sees the end-to-end delay of a received message or merged message by observing the timestamp given in the flexible header. The aggregator can see the same timestamp, but it is not as meaningful because the message has not yet reached its final destination.

### TABLE 4
STATISTICAL OUTPUT DATA POINTS

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages Received</td>
<td>Messages received by aggregators or collector</td>
</tr>
<tr>
<td>Bytes Received</td>
<td>Message bytes received by aggregators or collector</td>
</tr>
<tr>
<td>Packets Received</td>
<td>Packets received by aggregators or collector</td>
</tr>
<tr>
<td>Packets Sent</td>
<td>Packets sent by aggregators</td>
</tr>
<tr>
<td>Bytes Sent</td>
<td>Message bytes sent by aggregators or applications</td>
</tr>
<tr>
<td>Messages Sent</td>
<td>Messages sent by applications</td>
</tr>
<tr>
<td>Packet Failures</td>
<td>Packets dropped by aggregators due to no parent aggregator set</td>
</tr>
<tr>
<td>Probe Packets</td>
<td>Probes received by collector</td>
</tr>
<tr>
<td>Messages Merged</td>
<td>Merges by aggregator functions, or merges derived by collector</td>
</tr>
<tr>
<td>Bytes Merged</td>
<td>by counting message data values</td>
</tr>
<tr>
<td>Message Delay</td>
<td>Perfect hold delay (aggregators) or perfect end-to-end delay (collector)</td>
</tr>
<tr>
<td>Messages Concatenated</td>
<td>Messages combined into single datagram by aggregators or derived by collector</td>
</tr>
<tr>
<td>Total Messages</td>
<td>Sum of messages received by collector including messages merged</td>
</tr>
<tr>
<td>Total Bytes</td>
<td>Sum of bytes received by collector including bytes merged</td>
</tr>
<tr>
<td>Packet and Byte Reduction Ratios</td>
<td>Sent over received (aggregators) or merged and concatenated over total (collector)</td>
</tr>
<tr>
<td>Average Message Delay</td>
<td>Message delay over sent messages (aggregators) or over total messages (collector)</td>
</tr>
</tbody>
</table>
CHAPTER 5

FEEDER SIMULATION

With a working DATP model developed in NS-3 release 16 (NS-3.16), the simulation of different aggregation parameters in a smart grid environment could be completed. Early on, much time was spent on developing a feeder scenario while becoming familiar with the NS-3 simulation environment. A node topology was tested based on a distribution feeder model, wireless parameters were tested, and routing protocols were tested. An explanation of the simulation development will be given, followed by a description of the trials that were executed, and finally the results of the simulations.

The objective of the DATP feeder simulation was to show that the applications in a wireless feeder topology could benefit from the data aggregation system through increasing the capability to deliver messages to the collector without imposing a burdensome delay to message delivery. The simple scheduler also needed to be evaluated to determine optimal parameters.

Development

Nodes

Nodes in NS-3 represent any device that can participate in network communications. The node can act as a router and pass traffic, it can run network applications, or it can do both. In creating a wireless network simulation, the number, placement, and functionality of nodes is a critical foundation to an operational simulation. When applying the data aggregation principles to a wireless network in a distribution system, it was desirable that the number and placement of wireless nodes be modeled like a real distribution feeder instead of relying on random node placement.
In previous work [28], the GridLab-D simulator was utilized to study the effects of lost price change updates on home energy consumption in a dynamic pricing distribution feeder. GridLab-D is a power distribution system simulator. Through research funded by the U.S. Department of Energy Modern Grid Initiative, 24 distribution feeder taxonomy models representing prototypical radial distribution feeders were developed. The GridLab-D models define hierarchy and connection between objects in the distribution system. Objects include meters, transformers, switches, fuses, relays, commercial loads, and complex house model loads. Generic nodes separate different distance segments of the above ground or underground electric lines. Distances in feet were built into the power model, but exact X and Y coordinates were not given. The feeder starts from a substation and connections are defined between every object in the simulation without forming any loops. The GridLAB-D simulation model objects became a logical choice for use as wireless nodes in the network simulation model.

The number of objects in each model varies but is well over one thousand in many of the models. To help with visualizing the feeder, GridLAB-D also provides a network graph of each model, which has most of the kinds of objects in it from the simulation model, but it leaves out objects like houses. These network graphs were used to generate a list of nodes, their names, and their physical positions for the simulation. Some of the objects from the network graph were cut out of the simulation since the objects were physically local to one another. The transformer was considered to be the starting point in the network simulation for data entering the distribution feeder.

**Topologies**

The topology generation method used for the network simulation seeks to place objects defined in the GridLab-D model in such a fashion that might reflect some properties that are
significant in the distribution feeder without requiring a micro analysis of their placement. The primary concern in a resulting physical topology was that wireless nodes or clusters of nodes would not be isolated from any other part of the grid, thus preventing communication.

Graph data was provided in a format used by Node-XL, a Microsoft Excel extension that generates network graphs based on a layout algorithm. Three of the advanced graph algorithms included in the version 1.0.1.200 release of Node-XL are Fruchterman-Reingold, Harel-Koren Fast Multiscale, and Sugiyama. The results of the algorithm include X and Y coordinates and a generated image of the network graph. Figure 11 gives the differing properties of each algorithm and an example topology. Note that all of the algorithms attempt to minimize the number of edge crossings. The Fruchterman-Reingold algorithm can require a large number of iterations to approach this objective. The GridLab-D documentation suggests using the Harel-Koren Fast Multiscale method [29], which produces different results each time the graph is created.

Network graph algorithms do not use edge distances as an input, so the graphs that are produced are not reflective of the distances built into the GridLab-D simulation model. However, the algorithms themselves, by nature, have properties that are likely in a distribution feeder, like minimizing edge crossings. The edges shown in the graphs represent the connections between grid objects in the feeder.

The Sugiyama algorithm is a layered directed graph approach that should always generate the same output positions while the input configuration parameters remain the same. Much preliminary work was done with a topology generated from this algorithm. Nodes were spread far enough apart that in many cases there were nodes or clusters not able to communicate up the feeder to the substation where the collector would be. The Node-XL graph always generates vertices positions within bounds of a 10,000 by 10,000 unit box. When these coordinates are
input into NS-3 that operates in meters, nodes that are not within around 120 meters of another node will not be able to communicate. If a few individual nodes are isolated, then the simulation would be fine, but if entire clusters of nodes are not able to make a connection with the collector, than the entire cluster would be disconnected from the aggregation system.

Figure 11. Graph algorithm properties and examples.
Therefore, the coordinates of each node in early tests were cut in half in order to ensure full communication capability. This approach maintains the structure of the graph but makes the distances reasonable for a wireless simulation. Because other topologies were explored, another approach was taken to determine the minimum distance reduction factor (DRF) for all nodes based on moving the most isolated node within communication distance of its closest neighbor instead of using a fixed 50% DRF. This approach was generally good for evenly spread topologies that did not have nodes isolated on remote areas but would become unruly with topologies with highly isolated nodes.

**Wireless Communications**

The NS-3 Wi-Fi module is extensive, and there are many options for different wireless models including infrastructure and ad hoc modes, propagation loss models, propagation delay models, error rate models, rate control algorithm models, and node mobility models. The ad hoc mode was always the selected mode. Early on, different rate control algorithms were tested for general usability, and since one has to be chosen (there is no default), the Adaptive Auto Rate Fallback with Collision Detection (AARFCD) rate-control algorithm was chosen. All other models were the default selected by higher layers of the Wi-Fi model including the standard error rate model, constant speed propagation delay model, and log distance propagation loss model. Nodes remained statically positioned in the simulations.

**Routing Protocols**

Extensive simulation testing work was done on the routing protocol to enable communication between the aggregators and the collector. The routing protocol was necessary so that all nodes in the network would be able to route towards the collector. The objective became having a routing protocol that would efficiently propagate routes for the collector in a
small amount of time. It would have been fine not to use a routing protocol, but in NS-3 when the network is an ad hoc wireless network, the global routing functionality (where every node is initialized to know how to route to every other node) does not work because of the variables in such a wireless network. Therefore, a routing protocol was necessary to establish connectivity without an independent solution. First, the Optimized Link State Routing (OLSR) was utilized, but it was quickly realized that this would generate a large quantity of traffic and a long wait time for the routing tables to fully propagate, since OLSR is a link state pro-active wireless routing protocol. It was realized that full route propagation was not needed since every node only needed to know the next-hop toward the collector in the simulations. Therefore, the ad-hoc on-demand distance vector routing protocol became the logical choice. The route to the collector could propagate at the beginning of the simulation as each node attempted to send a probe packet to the collector. AODV defaults were changed, including disabling AODV Hellos, significantly boosting the active route timeout, and turning off gratuitous replies. The configuration defaults were changed based on the nature of the environment being simulated. With all these changes, the communication still failed at key points in the simulation between four nodes before the collector. These four nodes formed a box in topology as can be seen in Figure 12.
After analyzing the packet captures directly from the nodes, it was found that any one node in the box was able to successfully communicate to its adjacent nodes but was not close enough to the opposite node. If one node solicited a request that both adjacent nodes would respond to, then the adjacent nodes sent the response at the exact same time, which cancelled out both responses at the original and opposite nodes. Therefore, when an AODV route request packet was sent, the adjacent node responses would collide at either end and be dropped by the network. To remediate the potential for this issue to occur in various environments, a small random delay was introduced into AODV to delay the result enough to enable the responses to be generated at different times and, therefore, for the later-responding node to find the physical medium not free. An in-depth search was also done to discover the root issues behind introducing such random delays under the guise of node-processing delays and to collaborate with the NS-3 developer community on the issue. A number of bugs were reviewed on the topic [30, 31], and two were submitted for the two routing protocols [32, 33]. The AODV code changes to the NS-3.16 release that were used in the simulations can be seen in Appendix A.
Trials

Many simulation trials were executed to develop and obtain conclusive results. In each trial, a series of simulations was executed across multiple topologies with different parameters of the aggregation variable being tested. Six prototypical feeder topologies with varying node sizes were selected for simulations. The names and positions of the nodes were exported from the network graph file provided by GridLab-D after the following modifications:

1. Remove all triplex meters and triplex nodes from the vertices sheet (except on R1-12.47-3, which had only three such objects).
2. Remove all connections to and from these objects in the edges sheet.
3. Generate graphs with the three advanced network graph algorithms.
4. Subjectively evaluate the communication feasibility of generated graphs and select candidates for further testing.
5. Run test simulations on the candidates with varying DRFs, ensuring that nodes can identify parent aggregators through analyzing the output of the formed aggregation tree.

In the end, seven network graph topologies were chosen from the six feeder topologies for simulations. All selected graphs had no more than three or four lost nodes. Table 5 provides information about each topology. These seven topologies helped to determine whether aggregation results were consistent across diverse environments.
<table>
<thead>
<tr>
<th>Name</th>
<th>Alias</th>
<th>Nodes</th>
<th>Algorithm</th>
<th>DRF %</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1-12.47-3</td>
<td>r1a</td>
<td>137</td>
<td>Sugiyama</td>
<td>77</td>
</tr>
<tr>
<td>R2-12.47-2</td>
<td>r2a</td>
<td>536</td>
<td>Sugiyama</td>
<td>50, 46*</td>
</tr>
<tr>
<td>R4-25.00-1</td>
<td>r4a</td>
<td>407</td>
<td>Harel</td>
<td>68</td>
</tr>
<tr>
<td>R4-12.47-2</td>
<td>r4b</td>
<td>557</td>
<td>Harel</td>
<td>53, 60*</td>
</tr>
<tr>
<td>R4-12.47-2</td>
<td>r4c</td>
<td>557</td>
<td>Sugiyama</td>
<td>61, 60*</td>
</tr>
<tr>
<td>R1-12.47-2</td>
<td>r1b</td>
<td>718</td>
<td>Fruchterman</td>
<td>79</td>
</tr>
<tr>
<td>R3-12.47-1</td>
<td>r3a</td>
<td>1328</td>
<td>Harel</td>
<td>50, 36*</td>
</tr>
</tbody>
</table>

* DRF changed to second value after 7th trial

Simulations were run with the simple scheduler, simple function, and AODV tree controller implementations discussed in the previous chapter. One aggregator system was created. The first node given in the exported node list was always selected as the collector. Every other node was an aggregator with three applications installed. Each application sent messages with different data sizes at different frequencies, which can be seen in Table 6. In their flexible headers, the applications included an eight-byte timestamp field, one-byte application field, one-byte priority field, and one-byte data length field for a total header length of 12 bytes with the HFF field included. The priority field was not acted on by the aggregation system but was included to even out the header to 12 bytes.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Application One</th>
<th>Application Two</th>
<th>Application Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency per Second</td>
<td>10</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Data Length in Bytes</td>
<td>20</td>
<td>60</td>
<td>40</td>
</tr>
</tbody>
</table>
Three aggregation variables were tested across multiple trials. The first was to test for the best maximum scheduler delay with a uniform minimum delay. The second was to test for the best minimum scheduler delay with a constant maximum delay. The final was to test for three modes in which aggregators can operate. Aggregators can operate fully functional with both a scheduler and function in place (full aggregation mode), with a scheduler but no function in place (scheduler only mode), or without either (forward mode). Table 7 presents the specific variables and values tested for the trials.

<table>
<thead>
<tr>
<th>Trial</th>
<th>Topo.</th>
<th>Mode</th>
<th>Variable</th>
<th>Sets</th>
<th>Maximum and Minimum Delay Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>full, sch.</td>
<td>Maximum</td>
<td>4</td>
<td>Max: 16, 4, 1, 0.25 ms (50% Min)</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>full</td>
<td>Minimum</td>
<td>4</td>
<td>Min: 75, 50, 25, 0% (4 ms Max)</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>full</td>
<td>Minimum</td>
<td>11</td>
<td>Min: 99, 90, 80, 70, 60, 50, 40, 30, 20, 10, 0% (4 ms Max)</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>full</td>
<td>Maximum</td>
<td>10</td>
<td>Max: 20, 16, 12, 8, 4, 2, 1, 0.5, 0.2, 0.1 ms (50% Min)</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>full</td>
<td>Maximum</td>
<td>10</td>
<td>Max: 1.2, 1, 0.8, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.05 ms (50% Max)</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>full</td>
<td>Minimum</td>
<td>11</td>
<td>Min: 90, 80, 70, 60, 50, 40, 30, 20, 10, 5, 1% (1 ms Max)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>full, sch.</td>
<td>Maximum</td>
<td>25</td>
<td>Max: 50, 30, 20, 5, 3, 2.5, 2, 1.8, 1.6, 1.4, 1.0, 0.8, 0.6, 0.5, 0.4, 0.3, 0.25, 0.15, 0.1, 0.05, 0.025, 0.01, 0.005 ms (50% Min)</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>full, sch.</td>
<td>Minimum</td>
<td>13</td>
<td>Min: 99, 95, 90, 80, 70, 60, 40, 30, 20, 10, 5, 1, 0.5% (1 ms Max)</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>full, sch., fwd.</td>
<td>Modes</td>
<td>3</td>
<td>0.5 ms Max and 20% Min</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>full</td>
<td>Maximum</td>
<td>20</td>
<td>Max: 40, 30, 20, 10, 5, 2.5, 2, 1.5, 1, 0.8, 0.75, 0.5, 0.4, 0.25, 0.2, 0.15, 0.1, 0.05, 0.025, 0.001 ms (10% Min)</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>full, sch., fwd.</td>
<td>Modes</td>
<td>3</td>
<td>1 ms Max and 0% Min</td>
</tr>
</tbody>
</table>

The complete output results, final simulation script, trial run scripts, topology files, and AODV code diff file can be accessed via the public repository [27]. Furthermore, Appendix B provides an idea of the progress made on the DATP code according to the file version tracking commit logs that recorded the development progress of the project. Finally, a full size figure of each of the seven topologies is given in Appendix C.
**Results**

The trials that tested the maximum and minimum scheduler attributes were evaluated by dividing the average message delay by the packet reduction ratio, which resulted in a value that represented the system’s cost versus benefit. The lower the value, the better the aggregation system did at delivering aggregated data at a low delay per message. This result could be viewed from either a collector perspective, which measured end-to-end message delay, or an aggregator perspective, which measured the combined scheduler hold delay of all aggregators.

With optimal scheduler attributes selected, a comparison of the three aggregation modes was completed by evaluating the average message delay and total received messages from the collector’s perspective. Because the forward mode aggregation system gains no aggregation benefit, there could not be a comparison of the data reduction among the three modes.

**Maximum Delay Tuning**

Results for the maximum delay trials show that there is an optimum range of maximum delay values. The collector and aggregator perspectives disagree in this range, but they do overlap. The collector findings suggest the range is between 1,000 µs and 2,500 µs, which can be seen in Figure 13.

![Figure 13. Collector’s maximum delay performance.](image-url)
The results from Figure 13 are from trial 9 and include all of topologies with aggregators in a full aggregation mode. Aggregator findings suggest that the ideal maximum delay range is between 500 µs and 1,500 µs, as shown in Figure 14.

Figure 14. Aggregators’ maximum delay performance.

The results from Figure 14 are from trial 6 and include two topologies in both full aggregation mode and scheduler only mode. The consistency of the shape of the results across different topologies suggests that they are uniform across diverse environments. The best cost/benefit ratio can be seen when the maximum delay is below 10 µs; however, this is only because both the cost and benefit of the aggregation system are so low that the system is ineffective. Table 8 shows this outcome for the r4a topology in scheduler-only mode.

**TABLE 8**

<table>
<thead>
<tr>
<th>Maximum Delay (µs)</th>
<th>Message Delay (s)</th>
<th>Messages Concatenated</th>
<th>Packet Reduction Ratio (%)</th>
<th>Average Message Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>67.6949</td>
<td>19,313</td>
<td>0.508096</td>
<td>9.99E-05</td>
</tr>
<tr>
<td>50</td>
<td>34.6736</td>
<td>9,992</td>
<td>0.352231</td>
<td>5.00E-05</td>
</tr>
<tr>
<td>25</td>
<td>16.9734</td>
<td>5,198</td>
<td>0.281897</td>
<td>2.50E-05</td>
</tr>
<tr>
<td>10</td>
<td>6.83033</td>
<td>2,778</td>
<td>0.244787</td>
<td>1.00E-05</td>
</tr>
<tr>
<td>5</td>
<td>3.43182</td>
<td>1,875</td>
<td>0.229589</td>
<td>5.00E-06</td>
</tr>
</tbody>
</table>
**Minimum Delay Tuning**

Figure 15 shows the results for trial 7, which was conducted on two topologies, in full and scheduler mode, and with a wide range of thresholds. The results suggest that the minimum threshold variable does not assist the aggregation system in efficiently aggregating data because the lowest delays occurred when the threshold was set at 10% and lower. The original idea was that the minimum threshold delay would enable the scheduler to collect a higher quantity of messages to either be acted on by the function or concatenated together in one packet. However, there seemed to be a decreasing benefit as the minimum delay was increased.

![Figure 15. Collector’s minimum delay performance.](chart)

This outcome can be explained by observing that while the maximum threshold ensures that at least one message will be delayed for the maximum amount of time, any messages that come in during that period will already be merged according to the function or at least concatenated by the scheduler. During this period of waiting, the minimum delay applied to incoming messages only prevents their immediate aggregation if they have not waited long enough, thus increasing their delay and possibly decreasing the number of messages aggregated in each transmission.
Table 9 provides the r2a full aggregation results, which reveal better outcomes across nearly all data points as the minimum threshold decreases.

**TABLE 9**

**R2A MINIMUM THRESHOLD IN FULL AGGREGATION MODE**

<table>
<thead>
<tr>
<th>Minimum Delay Threshold (%)</th>
<th>Total Messages</th>
<th>Packets Received</th>
<th>Messages Received</th>
<th>Messages Merged</th>
<th>Message Delay (s)</th>
<th>Messages Concatenated</th>
<th>Packet Reduction Ratio (%)</th>
<th>Average Message Delay (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.5</td>
<td>30,098</td>
<td>7,182</td>
<td>7,183</td>
<td>22,915</td>
<td>60,798.5</td>
<td>1</td>
<td>76.14</td>
<td>2.020</td>
</tr>
<tr>
<td>99.0</td>
<td>29,634</td>
<td>7,602</td>
<td>7,607</td>
<td>22,027</td>
<td>50,496.3</td>
<td>5</td>
<td>74.35</td>
<td>1.704</td>
</tr>
<tr>
<td>95.0</td>
<td>28,754</td>
<td>7,510</td>
<td>21,244</td>
<td>35,327.3</td>
<td>53,678.1</td>
<td>60</td>
<td>74.09</td>
<td>1.229</td>
</tr>
<tr>
<td>90.0</td>
<td>31,153</td>
<td>7,270</td>
<td>23,762</td>
<td>53,678.1</td>
<td>121</td>
<td>76.66</td>
<td>1.723</td>
<td></td>
</tr>
<tr>
<td>80.0</td>
<td>34,923</td>
<td>7,425</td>
<td>27,234</td>
<td>48,044.2</td>
<td>264</td>
<td>78.74</td>
<td>1.376</td>
<td></td>
</tr>
<tr>
<td>70.0</td>
<td>35,381</td>
<td>6,401</td>
<td>27,860</td>
<td>24,146.9</td>
<td>1,120</td>
<td>81.91</td>
<td>0.682</td>
<td></td>
</tr>
<tr>
<td>60.0</td>
<td>52,710</td>
<td>6,023</td>
<td>8,310</td>
<td>44,400</td>
<td>2,287</td>
<td>88.57</td>
<td>0.759</td>
<td></td>
</tr>
<tr>
<td>40.0</td>
<td>67,819</td>
<td>5,385</td>
<td>7,801</td>
<td>60,018</td>
<td>2,416</td>
<td>92.06</td>
<td>0.366</td>
<td></td>
</tr>
<tr>
<td>30.0</td>
<td>56,012</td>
<td>5,402</td>
<td>8,008</td>
<td>48,004</td>
<td>2,606</td>
<td>90.36</td>
<td>0.154</td>
<td></td>
</tr>
<tr>
<td>20.0</td>
<td>38,775</td>
<td>5,189</td>
<td>7,424</td>
<td>31,351</td>
<td>2,235</td>
<td>86.62</td>
<td>0.124</td>
<td></td>
</tr>
<tr>
<td>10.0</td>
<td>60,173</td>
<td>5,394</td>
<td>8,273</td>
<td>51,900</td>
<td>2,879</td>
<td>91.04</td>
<td>0.153</td>
<td></td>
</tr>
<tr>
<td>5.0</td>
<td>59,215</td>
<td>5,074</td>
<td>7,304</td>
<td>51,911</td>
<td>2,230</td>
<td>91.43</td>
<td>0.166</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>54,225</td>
<td>5,208</td>
<td>7,774</td>
<td>46,451</td>
<td>2,566</td>
<td>90.40</td>
<td>0.128</td>
<td></td>
</tr>
</tbody>
</table>

The results from the aggregator perspective were similar and are shown in Figure 16. Therefore, the minimum threshold is not an effective variable for the simple scheduler.

![Figure 16. Aggregators’ minimum delay performance.](image-url)
**Delay in Aggregation Modes**

By comparing the results of the three aggregation modes acting in each topology in trial 10, it is apparent that the aggregation system provides significant overall improvement for the entire application environment. The scheduler ran with a one millisecond maximum delay and a zero minimum delay. The end-to-end average message delay is significantly reduced in full aggregation mode, compared to other modes in every topology, as can be seen in Figure 17. In five of the seven topologies, the scheduler-only mode also decreases the average message delay compared to the forward mode case. These results show that the aggregation system is capable of providing a delay decreasing benefit instead of imposing a delay increasing cost.

![Figure 17](image.png)

**Figure 17.** Collector’s average message delay.

In both full and scheduler-only modes, the number of delivered messages is greater than in the forward aggregation mode. Figure 18 shows the percentage of messages that were delivered to the collector in each mode.
Figure 18. Collector’s total received messages over applications' total sent.

It is counter-intuitive to assume that the aggregation modes that cause messages to be delayed for a period in each aggregator would result in a lower end-to-end message delay. However, this is feasible, considering that the act of aggregating data or packets results in smaller packets and fewer packet transmissions, which would result in a much higher availability of a loaded and shared wireless medium. In a topology where hundreds of nodes are attempting to simultaneously gain access to a medium to transmit and retransmit packets on towards one destination, the resulting higher availability of the medium would decrease the wait time for packets being transmitted. Figures 19 and 20 confirm this conclusion. Figure 19 shows the combined number of packets sent by the aggregators, and Figure 20 shows the combined message bytes sent by all aggregators. The full and scheduler-only mode both show a significant reduction in packet transmissions. The scheduler-only mode has a much higher number of sent bytes than either of the other modes, suggesting that the average packet size would be larger. Therefore, the full mode does better in both areas, and the scheduler-only mode does better in one area and worse in the other.
With these results it is evident that a DATP system implemented in a smart grid distribution feeder could improve the capability of the network to support smart grid applications.

![Figure 19. Aggregators’ total packets sent.](image)

![Figure 20. Aggregators’ total bytes sent.](image)
CHAPTER 6
EXPERIMENT

When DATP concepts were being developed, a real-world experiment was created to test the possibility of using data aggregation as an application on the Internet today. During the experiment, a total of about 30 Internet-connected hosts were connected to the data aggregation system simultaneously. These hosts came from the university, the cloud (Amazon EC2), social networking responders (friends and colleagues through LinkedIn [34]), advertisement responders (Google Ads), and micro labor market responders (Amazon Mechanical Turk).

**Internet Aggregation System**

The challenge of a system like the DATP running on the Internet is similar to problems faced by peer-to-peer applications involving access to and from many hosts. Hosts on a private network can only initiate outbound connections but need special configurations in order for inbound communication requests to make it to them. The system was programmed using Python and consisted of a collector script and an aggregator script, which was distributed and revised through a public repository [27]. Users could run the aggregator script on their computers and optionally configure their gateway router to allow inbound communication so that the aggregator functionality could be enabled; otherwise, it simply would act as an application and send data into the system.

In the Internet system, the collector required message exchanges with new aggregators to determine whether they were capable of receiving messages from other nodes. A list of potential parent aggregators was sent to each new node joining the system. In this case, a simple latency test was conducted by the aggregator to find feasible parents and then begin sending its own data to that parent. The data it sent into the system was a very small message of four bytes containing...
the node’s own public address. This aggregation system only concatenated messages together. The boundaries between the components were not clearly defined in the aggregator script, but essentially the scheduling part of the code kept a timer, whereby it would release any held messages each time the timer expired. The function component did not exist, since there was no manipulation of the data. The application component injected a message into the system at a random interval for each node between 100 and 1,000 milliseconds, and the scheduler timer was one tenth of that time. The stability of the aggregation system depended on a significant amount of intelligence being built into the tree controller. Various checks and tests would be conducted at some interval to ensure that aggregators did not accidentally form an aggregation loop, where messages are forwarded from one aggregator to the next in a circle.

**Results**

Since the data sent by each aggregator application was the node’s own identity, it was possible to track at the collector what nodes were actually sending into the aggregation system. October 17, 2013, was the planned date to request all participants to run the aggregator and join the aggregation system. The collector was run on a Raspberry Pi micro computer running the Rasbian operating system. The event was considered successful as nodes joined the system, and statistics on the data received were recorded each hour by the collector. A graph of aggregation statistics collected each hour for 48 hours can be seen in Figure 21. Each datagram received by the collector was counted, and each message in each datagram was counted against the address it contained. An aggregation rate was calculated based on how many network header bytes were saved due to multiple messages being in a single datagram, as opposed to each message being in its own datagram. Every host that ran the aggregator script acted as an application in the system, but it would only register as an aggregator to the collector if it was capable of receiving
messages from other nodes. Not all hosts were set up with this capability, which is why the number of aggregators is less than the number of applications in the system.

![Figure 21. Aggregation system statistics.](image)

The results look as expected. As more nodes joined the system, the number of messages and datagrams received by the collector increased. It can be seen that between 03:00 and 06:00 on October 16, the number of messages received equaled the number of datagrams received, which meant all hosts were sending directly to the collector, and therefore, no messages were being concatenated. The peak hour for the aggregation system was between 21:00 and 21:59 on October 17, and more specific results from the collector for that hour can be found in Table 10. Datagrams were only received by the collector’s child aggregators, but the number of messages sent by each aggregator or application was kept by reading the messages in each datagram. Aggregators also reported what nodes were sending to them.
<table>
<thead>
<tr>
<th>Child</th>
<th>Datagrams</th>
<th>Messages</th>
<th>Latency (s)</th>
<th>Aggregator On</th>
<th>Children</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>11,249</td>
<td>0.078818083</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>16,893</td>
<td>4.997863054</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>5,624</td>
<td>0.015528917</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>8,159</td>
<td>0.066055059</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>12,820</td>
<td>0.27411294</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>39,743</td>
<td>9,377</td>
<td>0.083891153</td>
<td>TRUE</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>19,143</td>
<td>3,927</td>
<td>0.042908907</td>
<td>TRUE</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1,567</td>
<td>0.007288933</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>12,818</td>
<td>4,470</td>
<td>0.007214069</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>8,974</td>
<td>0.088577032</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>8,072</td>
<td>0.052386045</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>12,822</td>
<td>0.152813911</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>4,483</td>
<td>0.06539011</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>3,600</td>
<td>0.011765003</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>11,043</td>
<td>0.025933981</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>5,063</td>
<td>0.01550888</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>19,228</td>
<td>0.044641972</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>6,521</td>
<td>0.026785851</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>4,476</td>
<td>0.186764956</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>6,391</td>
<td>0.101531029</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>6,146</td>
<td>0.110506058</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>4,976</td>
<td>0.016806126</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>8,769</td>
<td>0.007320881</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>8,613</td>
<td>0.070086002</td>
<td>TRUE</td>
<td>2</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>3,588</td>
<td>0.110184908</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>3,579</td>
<td>0.202064037</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>6,836</td>
<td>0.105162144</td>
<td>TRUE</td>
<td>2</td>
</tr>
<tr>
<td>28</td>
<td>25,370</td>
<td>4,718</td>
<td>0.006709099</td>
<td>TRUE</td>
<td>5</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>3,898</td>
<td>0.057585955</td>
<td>TRUE</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>3,734</td>
<td>0.070154905</td>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
<td>5,983</td>
<td>0.08889699</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>8,759</td>
<td>8,759</td>
<td>0.068076849</td>
<td>TRUE</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>10,5833</strong></td>
<td><strong>23,4358</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finally, a detailed graph of the state of the aggregation system according to the collector can be found in Figure 22, which includes such information as the number of nodes that joined and left the system each hour, and also the number of control messages received in hundreds.

![Figure 22. Aggregation system state.](image)

Between 19:00 and 19:59, a spike in joins can be seen, but the number of applications and aggregators present in the system remained steady. This was due to the collector software being manually restarted around 19:06. In fact a log from the collector script reported this message, “10/17/2013 19:06:04 - CRITICAL - Collector is going down. Informing all known open aggregators that we are going terminal.” The collector being restarted would in effect trigger aggregators to attempt to leave and rejoin the system. This latter item was the desired result, because a significant bug had been identified that evening that would prevent data collection of the aggregation tree that was formed during that hour. The bug was fixed, and a patched version of that aggregator was distributed to all aggregators. The statistics gathered by
the collector enabled the generation of Figure 23 through the use of Google Maps Engine [35]. An IP address location service [36] was used to obtain geographical coordinates for each IP address observed on the system and then given as inputs into Google Maps Engine. The connections in the graph represent the path that messages took towards the collector.

![Map of aggregation system participants](image)

**Figure 23.** Map of aggregation system participants.
CHAPTER 7
CONCLUSION AND FUTURE WORK

Conclusion

The most significant finding of this work is that a simple application layer approach to data aggregation can result in improvements for many-to-one systems, which have traditionally relied on complex cross-layer optimization techniques. Furthermore, based on the research done on existing data aggregation techniques, it is evident that the DATP approaches the data aggregation problem much more holistically and provides a complete view of what a data aggregation system is made of, compared to the works identified. The other protocols studied have attempted to solve the data aggregation problem solely from a tree formation, scheduling, or function standpoint. But now there is a framework by which these individual pieces of the solution can be part of a complete data aggregation system.

The most surprising operational advantage gained by the DATP is that the average message delay can be decreased by the aggregation system. Figure 24 further emphasizes this conclusion by showing the percentage reduction of the average message delay for the full aggregation mode and the scheduler-only aggregation mode over the forward aggregation mode.

![Figure 24. Average message delay reduction over forward mode aggregation.](image-url)
The full aggregation mode consistently reduces the message delay by an average of 90%, while the scheduler-only aggregation mode does so by 30%. The simple function used in the full aggregation mode does lean towards being more advantageous than most practical functions may be implemented because it completely merges messages, including both the header and the data. It could be considered the best-case function. The margin between the best-case function and no function (scheduler-only mode) is 60%. This demonstrates that there are many promises to data aggregation as opposed to simple packet aggregation but that packet aggregation is also not useless on its own.

The objective of the smart grid at the distribution feeder level is to provide a two-way communication network that enables smart meters and any other smart grid devices to run network applications that connect to the utility. As the scope of deployment for smart grid devices increases and the scope for the number of applications increases, this should affect the decision for the type of infrastructure that is sought for investment. Whether it is a wired or wireless solution, whether it is long haul or short haul, or whether it is some kind of mix will dramatically affect the cost and capability of the network. Depending on what solution is implemented, data aggregation may or may not significantly help increase network capability. Regardless, the vision that is being conveyed in this work is that the scope for utilization of data aggregation should be more than a fix for capacity constraints, but rather a best-practice method for implementing many-to-one communication systems of which a smart distribution feeder system would surely qualify. However, there are many other applicable systems in which the DATP could play a part, including the following:

- Internet applications requiring many-to-one communication.
- The Internet of Things where data from things goes to a collector.
• Log and system event aggregation.
• Aggregation for Big Data.
• Wireless sensor network data aggregation.

**Future Work**

The DATP NS-3 code should be developed further by including other features, which have been described in the protocol, such as support for reliable transport protocol usage and for components to increasingly interact with applications. Table 11 gives examples of ways that the components of the DATP could interact with applications to a greater degree. Every component of the DATP could find ways to operate more effectively if awareness of the applications behavior is given.

**TABLE 11**

INTEGRATION OF DATA AGGREGATION COMPONENTS WITH APPLICATIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Applications may want to influence how the function operates on their respective messages data and or headers. A system specific field could be a flag that tells what operation to perform on the data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler</td>
<td>Applications may want to influence the priority given to application messages. Ideally, the scheduler would be capable of determining the priority or importance of a message based on the header fields. Perhaps the application would want the scheduler to use a different scheduling method.</td>
</tr>
<tr>
<td>Collector</td>
<td>The collector needs to know what to do with each application message it receives. Most of this is independent of the aggregation system, but it may need to execute some reverse-functions before sending data forward.</td>
</tr>
<tr>
<td>Aggregator</td>
<td>Applications may want to be able to send only the data to an aggregator and then have the aggregator create a full message with a DATP header based on some predetermined configurations per application.</td>
</tr>
</tbody>
</table>

Now that there is a data aggregation protocol in place that is complete, flexible, and modular, further work can be done to implement established and novel solutions to components of the data aggregation system. Others could build onto the DATP NS-3 code base with new modules for functions, schedulers, or tree controllers. Past research done on aggregation systems
reveals that solutions have been focused on only a piece of the data aggregation system. Now those pieces can be fit into a formalized approach to data aggregation to legitimize solutions and establish data aggregation as a viable solution to many-to-one communication systems.

Most importantly, the security of the DATP should be studied thoroughly, and modifications that fit into the DATP framework should be suggested, in order to ensure that real systems remain secure while utilizing the protocol.
REFERENCES


REFERENCES (continued)


REFERENCES (continued)


[34] Internet Experiment LinkedIn Group, URL: http://www.linkedin.com/groups/Andrew-Stantons-WSU-Masters-Thesis-6518497 [cited June 30, 2014].
REFERENCES (continued)


APPENDICES
APPENDIX A

NS-3.16 AODV CODE CHANGES

The following is a Mercurial code diff which can be imported as a patch to an NS-3.16 installation for the purpose of reproducing the results that were gathered:

diff -r c806da296b56 -r 7fd049814ad src/aodv/model/aodv-routing-protocol.cc
+++ b/src/aodv/model/aodv-routing-protocol.cc   Sun May 05 16:16:40 2013 -0700
@@ -910,6 +910,7 @@
{  
    if (!m_htimer.IsRunning ())  
    {  
        NS_LOG_LOGIC ("Reset HelloInterval since broadcasting RREQ");
        m_htimer.Cancel ();
        m_htimer.Schedule (HelloInterval - Time (0.01 * MilliSeconds (  
            m_uniformRandomVariable->GetInteger (0, 10))));
    }
@@ -1150,7 +1151,18 @@
    rreqHeader.SetUnknownSeqno (false);
    }
    
+  //Forward RREQ with a random delay to prevent broadcast collision
+  uint32_t randomDelay = m_uniformRandomVariable->GetInteger (10, 50) * 10;
+  NS_LOG_DEBUG ("Forward RREQ from " << src << ", with delay " << randomDelay << " us");
+  Simulator::Schedule (Time (MicroSeconds (randomDelay)),
+                       &RoutingProtocol::ForwardRequest, this, rreqHeader);
+}
+}
+}
+}
+}
+}
+ void  
+RoutingProtocol::ForwardRequest (RreqHeader rreqHeader)
+{  
+  NS_LOG_FUNCTION (this);
+  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
+    m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
+  {  
+    if (!m_htimer.IsRunning ())  
+    {  
+        NS_LOG_LOGIC ("Reset HelloInterval since broadcasting forwarded RREQ");
+        m_htimer.Cancel ();
+        m_htimer.Schedule (HelloInterval - Time (0.1 * MilliSeconds (  
+            m_uniformRandomVariable->GetInteger (0, 10))));
+    }

diff -r c806da296b56 -r 7fd049814ad src/aodv/model/aodv-routing-protocol.h
+++ b/src/aodv/model/aodv-routing-protocol.h    Sun May 05 16:16:40 2013 -0700
@@ -216,7 +216,11 @@
/// Receive RERR from node with address src
void RecvError (Ptr<Packet> p, Ipv4Address src);
///
+  
+  ///\name Forward control packet
+  
+  ///\{  
+  ///Forward RREQ  
+  void ForwardRequest (RreqHeader rreqHeader);
+  ///\}  
+  ///\name Send  
+  ///\}  
+  /// Forward packet from route request queue

71
APPENDIX B

NS-3 DATP FILE VERSION TRACKING COMMIT LOGS

changeset: 9248:274702606d4c
tag: tip
user: Andrew Stanton
date: Tue Apr 08 02:50:51 2014 -0500
summary: add option to pcap all nodes, change agg start to 0.0, and change total output in datp to start with distinguishing character

changeset: 9247:ce8a1f136403
user: Andrew Stanton
date: Sun Apr 06 16:10:30 2014 -0500
summary: remove extra variable in datp aggregator

changeset: 9246:0afcl45c4b22f
user: Andrew Stanton
date: Sun Apr 06 14:39:09 2014 -0500
summary: stop tracking node position config files

changeset: 9245:36e4fe9d9dd63
user: Andrew Stanton
date: Sat Apr 05 22:36:37 2014 -0500
summary: adjusted drfs, timing, apps optional, tracing, and more

changeset: 9244:f79aa3eefb86
user: Andrew Stanton
date: Sat Apr 05 16:18:47 2014 -0500
summary: changed approach to distance reduction factor, and significantly changed feeder node creation, naming, and positioning to more simple

changeset: 9243:53daa5eba1dd
user: Andrew Stanton
date: Sat Apr 05 12:23:41 2014 -0500
summary: added datp and tree tracing to feeder

changeset: 9242:14e8451653f2
user: Andrew Stanton
date: Fri Apr 04 01:44:13 2014 -0500
summary: add tree tracing through helper, needed a get collector address function in aggregator to facilitate

changeset: 9241:18837c0ab937
user: Andrew Stanton
date: Fri Apr 04 00:33:37 2014 -0500
summary: adjust example to work with script to run multiple trials, add script, move datp trace to helper

changeset: 9240:013cd14f9dfb
user: Andrew Stanton
date: Fri Apr 04 00:31:28 2014 -0500
summary: cleanup efforts on feeder simulation

changeset: 9239:3435136dea3f
user: Andrew Stanton
date: Fri Apr 04 00:30:53 2014 -0500
summary: change random delay for tree controller aodv probes

changeset: 9238:7705519e1771
user: Andrew Stanton
date: Tue Apr 01 21:25:06 2014 -0500
summary: overhauled the application code to make a pure base class that handles the sockets, while derived classes define controlling attributes, and what goes into the message that is sent

changeset: 9237:07dbb7a897b6
user:        Andrew Stanton
date:        Mon Mar 31 01:18:58 2014 -0500
summary:     fixed bug with scheduler delay where delay too high, introduced statistic to track total number of messages data that passed through
changeset:   9236:a7ce08b5a7fd
user:        Andrew Stanton
date:        Mon Mar 31 00:06:50 2014 -0500
summary:     implemented great trick in simple function to average the times of the header in order to come out with perfect delay
changeset:   9235:6061303c6a9b
user:        Andrew Stanton
date:        Sun Mar 30 23:42:47 2014 -0500
summary:     significant modifications to grab and output statistics for many components of aggregation system
changeset:   9234:3b0b8d3a35ab
user:        Andrew Stanton
date:        Sun Mar 30 12:34:11 2014 -0500
summary:     significant changes to add capability to retrieve statistics via fn calls to all apps and objects
changeset:   9233:02f3b329f0db
user:        Andrew Stanton
date:        Thu Mar 27 19:14:46 2014 -0500
summary:     further prints on the collector
changeset:   9232:340d4744ca1e
user:        Andrew Stanton
date:        Tue Mar 25 22:06:41 2014 -0500
summary:     new features added to collector so that we can track many more statistics
changeset:   9231:f738b2306e2a
user:        Andrew Stanton
date:        Tue Mar 25 22:06:07 2014 -0500
summary:     fix bug in headers so that if values are set twice, then it will not add into hff again or into internal size
changeset:   9230:1010322db4b3
user:        Andrew Stanton
date:        Mon Mar 24 01:29:50 2014 -0500
summary:     fix bug in headers so that if values are set twice, then it will not add into hff again or into internal size
changeset:   9229:bda22d91f82b
user:        Andrew Stanton
date:        Mon Mar 24 00:41:19 2014 -0500
summary:     implement min/max hold time attributes in DatpSchedulerSimple
changeset:   9228:ccb855b2222f
user:        Andrew Stanton
date:        Mon Mar 24 00:30:47 2014 -0500
summary:     fixed simple scheduler bug identified by turning function off.. need to make map key based on message identifier instead of application, so that more than one message of same app can be held by scheduler
changeset:   9227:d7e82e6a9ca4
user:        Andrew Stanton
date:        Mon Mar 24 00:04:03 2014 -0500
summary:     significant updates to datp example
changeset:   9226:f1bfb946042c
user:        Andrew Stanton
date:        Sun Mar 23 22:51:26 2014 -0500
summary:     minor comment editing in aggregator and tree controller
changeset:   9225:f08095680ae5
user:        Andrew Stanton
summary: convert aggregate(On) attribute to separate functionOn and schedulerOn attributes and implement how these attribute values play out in Install, Receiver, etc. Basic fixup of the helper, but needs more controls.

changeset: 9224:237161ef2283
user: Andrew Stanton
date: Sun Mar 23 20:42:56 2014 -0500
summary: Adjust feeder for DATP, remove requirement in helper that collector not be in the other containers

changeset: 9223:ae9a9c7452c
user: Andrew Stanton
date: Sun Mar 16 20:13:42 2014 -0500
summary: fixup dstat example, change attribute defaults in aggregator to simple classes, adjust comments in header, and increase logging in tree-controller aodv

changeset: 9222:0c224e5043b6
user: Andrew Stanton
date: Sun Mar 16 18:55:08 2014 -0500
summary: one application had random delay different than the other

changeset: 9221:b79ee0d9f56a
user: Andrew Stanton
date: Sun Mar 16 15:43:02 2014 -0500
summary: missed reference to tree-controller-smi in wscript, adjusted to aodv

changeset: 9220:e0c2d8b175b0
user: Andrew Stanton
date: Sun Mar 16 15:40:11 2014 -0500
summary: fix merging errors in function-simple, move testdatp to example folder, rename tree-controller-smi to tree-controller-aodv and adjust references to previous name in multiple source files

changeset: 9219:b021c271be86
user: Andrew Stanton
date: Thu Mar 13 00:27:22 2014 -0500
summary: Bugs fixed: crash on parentAgg not set, crash on forward packet through agg, simple scheduler does not pass packets through. Increased logging and error watching.

changeset: 9218:03f03d7a9425
user: Andrew Stanton
date: Wed Mar 12 19:01:03 2014 -0500
summary: results after fixing compiler errors on datp module

changeset: 9217:ba9ba74db934
user: Andrew Stanton
date: Sun Mar 09 22:50:56 2014 -0500
summary: major progress on helper, aggregator, collector, functions, schedulers, and tree controller -- removed extraneous files

changeset: 9216:9f8bacd41667
user: Andrew Stanton
date: Sun Mar 09 16:49:03 2014 -0500
summary: make tree-controller as base class, change file names from agg and from app to full name, create duplicate files to base class app, function, scheduler, and create smi tree-controller

changeset: 9215:3676e8ed4a0b
user: Andrew Stanton
date: Fri Mar 07 00:33:14 2014 -0600
summary: remove unused var, cause c++ compiler has some warnings as errors (??)

changeset: 9214:d1c52255dc92
user: Andrew Stanton
date: Thu Mar 06 22:33:02 2014 -0800
summary: significant progress in developing datp model -- adding fn, smi, tree-controller

changeset: 9213:758f1137a013
user:        Andrew Stanton
date:        Sun Mar 02 18:24:13 2014 -0800
summary:     back to datp once again... did hg file cleanup - saving state before further modifications to actual files
changeset:   9212:9cc26eaca971
user:        Andrew Stanton
date:        Sat Oct 12 16:19:03 2013 -0700
summary:     after taking a long break on datp, I am coming back now, and stuck at where I left off since the program is crashing in DatpAggregator.
changeset:   9211:7fdf049814ad
user:        Andrew Stanton
date:        Sun May 05 16:16:40 2013 -0700
summary:     some AODV changes, and last test was 2013-04-18 23:17
changeset:   9210:c806da296b56
user:        Andrew Stanton
date:        Wed Apr 10 20:52:52 2013 -0700
summary:     start tracking feeder code changes, 2013-04-07 21:38
Figure 25. Aggregation tree of topology r1a.
Figure 26. Aggregation tree of topology r1b.
Figure 27. Aggregation tree of topology r2a.
Figure 28. Aggregation tree of topology r3a.
Figure 29. Aggregation tree of topology r4a.
Figure 30. Aggregation tree of topology r4b.
Figure 31. Aggregation tree of topology r4c.