6-7-2019

Aggregation of Electric Water Heaters for Peak Shifting and Frequency Response Services

Thomas Leighton Clarke
Portland State University

Aggregation of Electric Water Heaters for Peak Shifting and Frequency Response Services

by

Thomas Leighton Clarke

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Portland State University
2019

Abstract

The increased penetration of renewable energy sources poses new challenges for grid stability. The stochastic and uncontrollable generation of solar and wind power cannot be adjusted to match the load profile, and the transition away from traditional synchronous generators is reducing the grid capacity to arrest and recover from frequency disturbances. Additionally, the distributed nature of many renewable energy sources makes centralized control of generation more complicated.

The traditional power system paradigm balances the supply and demand of electricity on the grid by regulating generation. As this becomes more difficult, one alternative is to adjust the load instead. This is not entirely novel, and utilities have incentivized large industrial customers to reduce consumption during peak hours for years. However, the residential sector, which constitutes 37% of electricity consumption in the U.S., currently has very little capacity for load control.

Smart electric water heaters provide utilities with an appliance that can be remotely controlled and serves as a form of energy storage. They have very fast response times and make up a large amount of residential energy consumption, making them useful for load peak shifting as well as other ancillary grid services. As smart appliances become increasingly widespread, more and more devices can be brought into the utility's control network and aggregated into a flexible resource on a megawatt scale.

This work demonstrates the usefulness of aggregated electric water heaters for peak shifting and frequency response. Because a large number of assets are required, emulators are developed based on observations of real devices. Emulated water heaters are then connected to an energy resource aggregator using an internet-of-things network. The aggregator successfully uses these assets to shift consumption away from peak hours. An algorithm was developed for detecting upward frequency disturbances in real-time. The aggregator uses this algorithm to show that an aggregation of water heaters is well-suited to respond to these frequency disturbances by quickly adding a large amount of load to the grid.

Dedication

To T.A.C.

Acknowledgements

Special thanks to Tylor Slay for being the MVP. You are a rare gem of a scientist and a person, and this work wouldn't have been possible without you. Thank you Bob Bass for being a great teacher, boss, and advisor, and for giving me the opportunity to be a part of this team. And, of course, thanks to Kevin and Yuki for putting up with me for the last two years.

Thank you to Conrad Eustis, Kevin Whitener, and PGE for supporting this research and providing me access to the SSPC data.

I'd also like to thank my mom for always being there for me, Jim Houck for giving me a place to live, and all of the friends near and far who didn't give up on me even when I disappeared from their lives for years at a time.

Finally, thank you Hong Lei and Richard Campbell for joining my thesis committee. I respect and appreciate the time and effort you have committed to my studies.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Problem Statement

The increasing contribution of renewable energy sources to electricity generation comes paired with new challenges for power grid reliability. Traditionally, the vast majority of electric power has been provided by a relatively small group of large, centrally-controlled generators. These generators can be dispatched to operate above or below their normal levels in response to changes in the balance between supply and demand. Additionally, they provide stability to the grid electrical frequency through their mechanical inertia. However, both of these qualities may be lost in the transition to renewable energy sources. Two of the main renewable sources, wind and solar power, are inherently stochastic in their generation capacity. The sun cannot be made to shine, and the wind cannot be made to blow. The times when supply is available often do not match the times of peak consumption, and the climatic factors that influence power output often also affect the load profile, leading to greater unpredictability. Generation sources that provide energy to the grid through DC/AC or AC/AC power conversion, such as photovoltaics, battery-inverter systems, and Type 4 wind turbine generators, have no mechanical inertia whatsoever, and detract from the grid's resilience to frequency disturbances. Due to the distributed nature of sources such as rooftop solar installations, accidental or intentional disconnection of a load region from the grid results in loss of generation as well, and may exacerbate problems.

Regardless of the transition to renewable generation, some characteristics of the power system will not change. The amount of energy supplied and consumed by the grid must always match. This balance can be maintained through control of generation, control of load, or both. The large-scale storage of readily dispatchable electric power, such as through batteries or pumped-storage hydroelectricity, is not currently economically feasible and possibly never will be [19]. If the new means of generation cannot be controlled sufficiently to provide grid stability, then the solution must lie with real-time control of power consumption. Providing the grid with energy balancing and ancillary services through control of the load is known as "demand response" (DR). This research characterizes smart electric water heaters as an asset for DR, and tests their use for load peak shifting and recovery from frequency disturbances.

1.2 Objectives of Work

Water heaters are primary candidates for DR, and a large body of work covering their potential application already exists. The research presented here distinguishes itself by incorporating the individual devices into an aggregator called Distributed Energy Resource Aggregation System (DERAS). DERAS is a PSU-built system that uses an internet-of-things (IoT) framework to communicate with different assets. Each individual asset is equipped with a distributed control system (DCS) that observes and reports parameters such as the asset's current state and the amount of energy that it can absorb. DERAS aggregates these parameters for all of the assets in its network and provides the user with data describing the entire system. Some benefits of this approach are that numerous small resources can be observed and controlled on a megawatt scale while preserving customer anonymity. Additionally, the stochastic differences between devices are evened out as more assets are added to the network, yielding more predictable load and generation patterns. By sending control signals to the assets, DERAS can then provide several services, such as executing a predetermined energy-balancing schedule or responding to frequency disturbances.

One of the enabling technologies for this is the CTA-2045 communication protocol specification, which allows DERAS to communicate with a variety of devices from different manufacturers. Two water heater models are used for this study: one with resistive heating elements, and one hybrid with both a heat pump and resistive elements. Each device comes from the manufacturer equipped with a CTA-2045 interface. The objective of this work is to demonstrate that DERAS can provide valuable services to a utility using a group of water heater assets via CTA-2045. In this work, the term ‘EWH’ will be used to refer to traditional resistive water heaters, and ‘HPWH’ will be used for heat pump water heaters (including those that also have resistive elements).

2 Literature Summary

2.1 Consequences of Solar and Wind Power for Grid Reliability

The U.S. Energy Information Administration reports that wind power contributed 6.6% of electricity generation in the U.S. in 2017, and solar power (including photovoltaic and solar thermal) contributed 1.6% [2]. While the U.S. lags behind the E.U., where the 2017 share of renewable energies was 17.5% [16], some regions of the U.S. are far ahead of the overall country. Many states have passed ambitious legislation requiring significant transitions in the first half of the 21st century. For example, Oregon has set a goal of a 50% renewable energy share by 2040, and both California and Hawaii have passed laws to reach 100% by 2045 [17][6].

High penetration of solar and wind power contributes to a problem known as the "duck curve." Peaks of solar power production at midday and wind at night tend to coincide with periods of low power consumption. This can lead to two major problems. First, the supply of power can exceed the demand so severely that it must be wastefully curtailed to avoid bringing conventional generation below its "must-run" minimum level [21]. Second, as these periods of high generation and low consumption end (for example, as the sun sets and people come home in the evening), a very steep ramp is created, which puts stress on the rest of the power system. Figure 2.1 shows an example of a duck curve forming as increased projections of distributed PV generation (here called DPV) are added to the gross load of a 2018 day in Hawaii. As the amount of PV increases, the afternoon load decreases below the minimum generation level, and the evening ramp becomes steeper. Due to the widespread use of PV in Hawaii, the daily ramps have shifted, steepened, and increased in number [20].

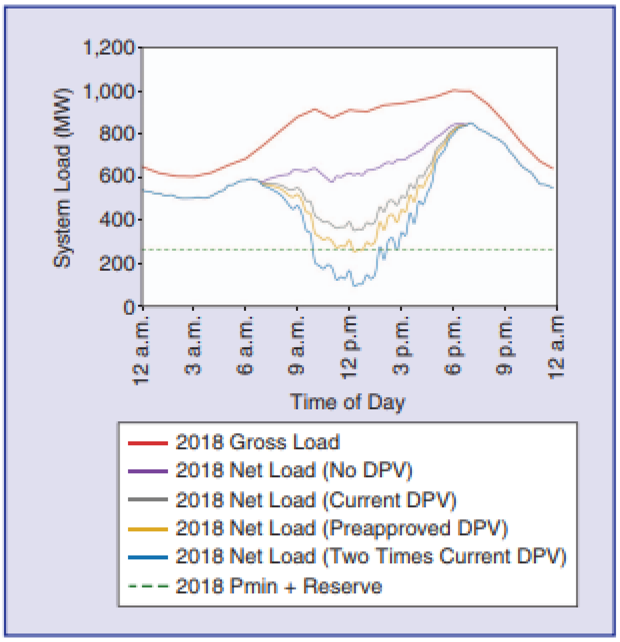


Figure 2.1: Oahu load profile with increasing distributed photovoltaics (DPV) projections [20]

While Hawaii is the most extreme example in the U.S., California's energy production also exceeds demand several times per year. In 2017 there were more than 30 days when the price of electricity went negative, meaning that the system operator was actually forced to pay utilities to take energy [6].

Another challenge presented by rooftop PV installations is that a significant portion of generation becomes distributed, rather then centralized. If an area is disconnected from the rest of the grid, whatever power that area was generating is also removed. This may happen unexpectedly due to faults, or the utility may intentionally shed load areas for a variety of reasons. If a major generator goes offline, load-shed blocks may be disconnected to stabilize the system frequency. However, if those blocks were also producing significant amounts of power, it could actually make the disturbance worse. An example of this happening after a trip of the largest generator on Oahu is shown in Figure 2.2.

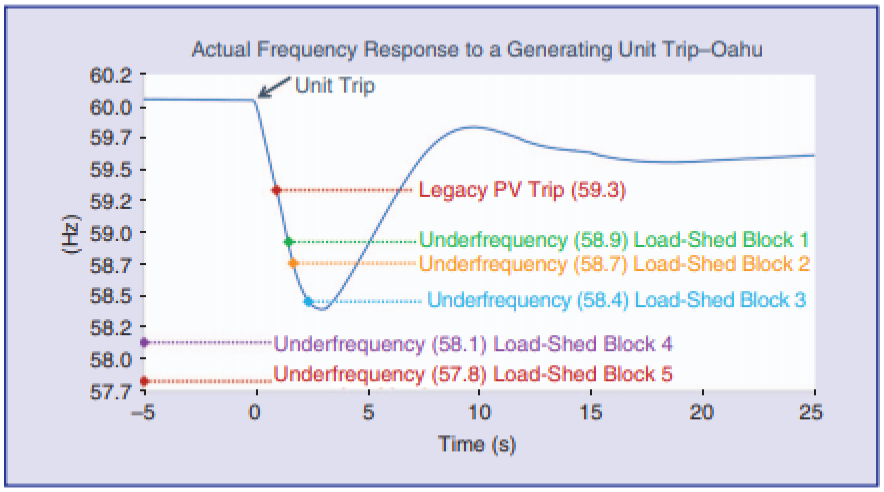
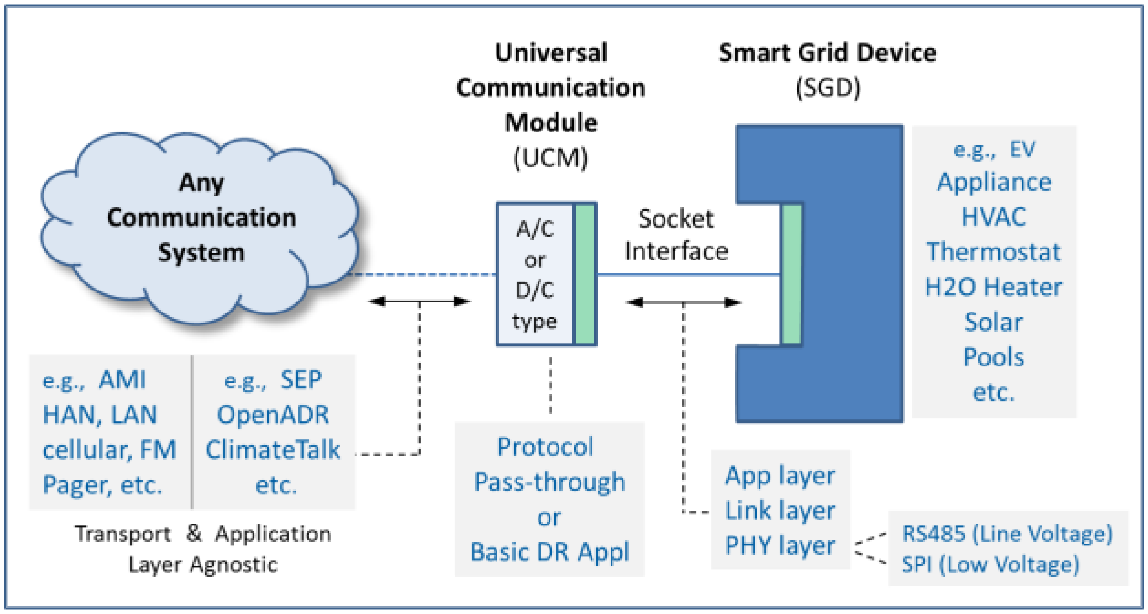


Figure 2.2: Frequency disturbance from generator loss and subsequent load-shedding [20]

2.2 The CTA-2045 Communications Specification

The CTA-2045 specification was developed from 2008 through 2012 by the Electric Power Research Institute (EPRI) and the Smart Grid Interoperability Panel (SGIP), with the goal of creating a universal demand response communications standard for a wide variety of devices and manufacturers [6]. The standard also defines a socket interface so that devices come ready for energy management functions directly from the manufacturer. The interface can communicate with other devices using one of two serial protocols: low voltage appliances are typically installed with interfaces for SPI, and larger devices use RS-485 [10]. Utilities can design and sell universal communication modules (UCMs) that plug into the CTA-2045 interface. The utility is then able to exchange telemetry with the UCM using whatever communication system they desire, such as Wi-Fi, radio, cellular networks, or even systems that haven't yet been invented [13], and the UCM translates that telemetry to and from the CTA-2045 standard. This flexibility is critical for appliances with long service lives, because major developments and changes to communication networks can take place over the device's lifespan. A generic block diagram demonstrating this chain of communication is shown in Figure 2.3.



Universal Communication Module (UCM)
Smart Grid Device (SGD)
Any Communication System
A/C or D/C type
Socket Interface
e.g., EV Appliance HVAC Thermostat H2O Heater Solar Pools etc.
e.g., AMI HAN, LAN cellular, FM Pager, etc.
e.g., SEP OpenADR ClimateTalk etc.
Transport & Application Layer Agnostic
Protocol Pass-through or Basic DR Appl
App layer Link layer PHY layer
RS485 (Line Voltage) SPI (Low Voltage)

Figure 2.3: Chain of communication from utility to asset [10]

The cost of equipping devices with UCMs is expected to decline steeply as CTA-2045 becomes more widespread. A study by the Bonneville Power Administration (BPA) has projected the cost of equipping a single water heater with a UCM dropping to \$25 by 2030 [6]. Beyond the benefit of allowing utilities to design whatever UCM is most practical for them, the modular approach also has cybersecurity advantages. If vulnerabilities in the interface are exposed, the UCM is much cheaper and easier to update or replace than the appliance's built-in hardware.

Any time a smart grid device (SGD) such as a smart water heater is connected to a network through a CTA-2045 interface, it can be queried as to what type of device it is, as well as the model and manufacturer. The standard requires that all devices be able to report a variety of energy parameters, which include but are not limited to the following:

- The SGD's current state of operation, which details if the device is importing or exporting power from the grid, or not receiving any commands at all. This parameter also has values for when the device has received commands but is unable to follow them, for example because the SGD is depleted or cannot import any more.
- The SGD's maximum rated power and the actual power being imported and exported.
- The total amount of energy the SGD is able to absorb, and the actual amount it is able to import and export before becoming completely charged or depleted.
- Many SGDs, including water heaters, must provide the customer with an option to override any outside requests. This override status must also be reported.

While all CTA-2045 devices must support these queries, not all of them are necessarily useful for a given device. For example, even though a water heater cannot export power to the grid, it must still be able to provide a zero if its rated export power is queried. There is also a library of obligatory and optional DR commands. The following commands must be supported by all CTA-2045 water heaters [12]:

- Shed: The water heater shall not operate, unless the water temperature reaches a minimum level required for consumer comfort. This level is determined by the manufacturer and is below the thermostat's normal activation temperature.
- Critical Peak Event: This command is similar to shed, but the water temperature bottom limit is even lower. However, it is only intended for use a few times per year.
- Load Up: The water heater should heat up to the customer's desired temperature (whatever is selected as the thermostat setpoint), and aggressively maintain that level.
- End Shed: This command ends any of the other above controls and returns the device to normal operation.

These commands can be given with a duration of up to twelve hours. The device will return to normal operation when the duration expires or an *end shed* command is received. Additionally, all CTA-2045 interfaces require a communication status update from the network (commonly called a "heartbeat") at least once every 15 minutes. If no heartbeat is received, the device assumes it has been disconnected from the network and returns to autonomous operation.

Designing a UCM or effectively communicating with a CTA-2045 interface from scratch would require a fairly extensive understanding of the standard's electrical requirements and functionality. However, EPRI has provided a C++ library for manufacturers and researchers, so that any controller with the appropriate serial port (SPI or RS-485) can be programmed to make queries and send commands using simple functions [14].

2.3 Use of Water Heaters for Demand Response

Utilities have been making use of DR in the industrial sector for years by engaging the largest energy consumers through dynamic tariffs or incentivized direct load control [26][19]. However, the residential sector's DR resources remain largely untapped. The residential sector constitutes 37% of electricity consumption in the U.S. [29] Water heating makes up a significant portion of this - 17% of the national residential consumption [18], and up to 30% in some regions [11]. Additionally, as shown in Figure 2.4, the times of water heater usage coincide with the peaks in overall electricity demand, meaning that control of water heaters can be a powerful tool for peak shifting.

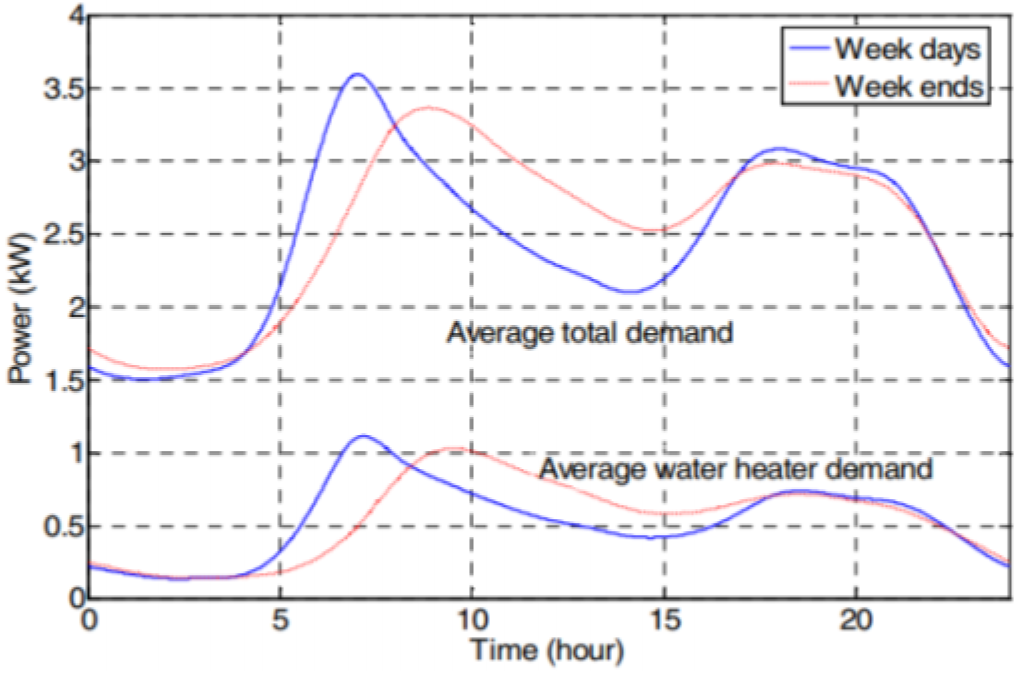


Figure 2.4: Average Daily Residential Load for Total Demand and Water Heating (Pacific Northwest) [5]

Water heaters are valuable DR assets for several more reasons: First, the rated power consumption of an individual device (typically 4500 W) is high, meaning that relatively few devices can quickly accumulate to have a bulk-scale impact. Second, EWHs turn on and off very quickly. The ramp rate from zero to full rated power is almost instantaneous, allowing for more precise control and response to short-term transient events such as frequency disturbances. Third, resistive water heaters are almost purely resistive and do not require reactive power support from the grid (unlike HVAC systems or induction machines, for example). Finally, water heaters serve as a form of energy storage, and the time of hot water use is not directly correlated to the time of electricity consumption.

The usefulness of water heaters for DR increases with the number of available assets, and is therefore directly dependent on customer participation and the market penetration of accessible devices. A pilot study of CTA-2045-equipped water heaters conducted by the BPA projects that, if 26.5% of all electric water heaters in the states of Oregon and Washington were enrolled in a DR program, the combined dispatchable load would be equivalent to the creation of a 301 MW peaking plant. This resource has an estimated long-term value of $106 million, and on a national scale extrapolates to $2 billion [6]. The report claims a benefit-cost ratio of 1.0 even if customer participation is as low as 5%. The BPA also proposes a market plan for utilities and the three major electric water heater manufacturers (AO Smith, Rheem, and Bradford White) to fill the Oregon and Washington markets with CTA-2045 equipped water heaters starting in 2025. Given the expected life-spans of traditional water heaters already in use, the report expects 91% of all water heaters to have CTA-2045 interfaces by 2039, as shown in Figure 2.5.

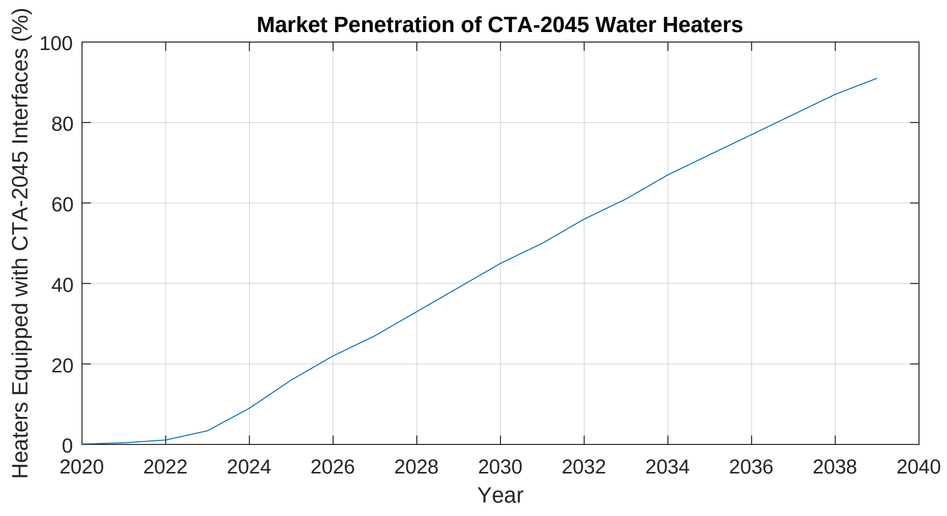


Figure 2.5: Expected Market Penetration of CTA-2045-equipped water heaters in Oregon and Washington [6]

Another challenge to residential DR is recruiting and maintaining a large number of participating customers. The primary concern for water heaters is that shed events will cause customers to run out of hot water, but this was only a minor issue in the BPA pilot study. The BPA enrolled 277 households, and sent approximately 600 DR commands over the course of 220 days. Customers received different compensation based on their local utility, Portland General Electric (PGE) offered customers a total of \$50 for starting and \$100 for completing the pilot [6]. Customer feedback was majority positive; 80% of customers reported that they were very satisfied with the program, and 94% stated that they would be very likely to enroll in similar programs in the future. Only two households left the program due to hot water shortages.

2.4 Heat Pump Water Heaters

Heat pump water heaters (HPWHs) achieve greater efficiency than traditional resistive water heaters by moving heat energy from the surrounding air into the tank. As shown in Figure 2.6, a motor-driven compressor moves ambient heat into a condenser coil, where it is absorbed into the tank. HPWHs are typically evaluated based on their coefficient of performance (COP), the ratio of heat energy transferred to the tank to electrical energy consumed. While an EWH cannot ever reach or exceed a COP of 1, it is normal for HPWHs to have COPs of over 2 [8].

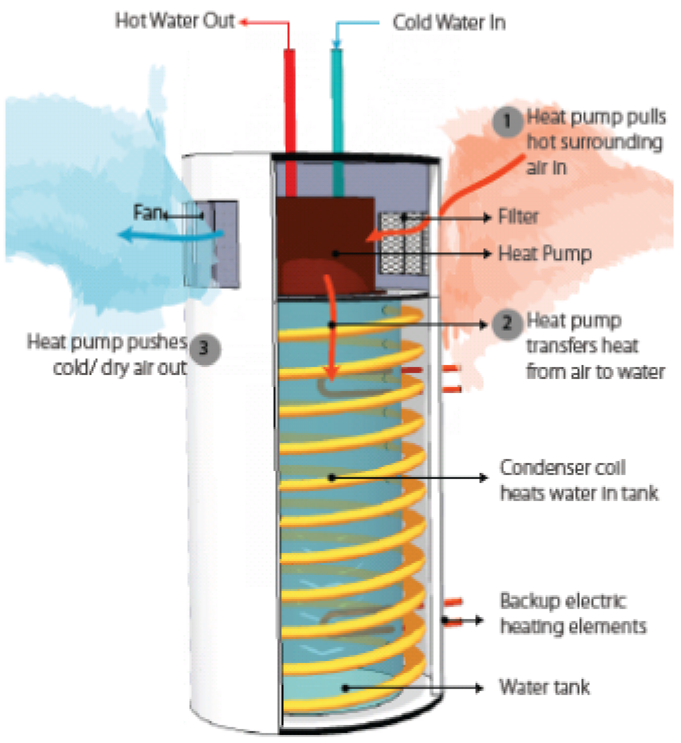

Hot Water Out
Cold Water In
1 Heat pump pulls hot surrounding air in
Fan
Filter
Heat Pump
2 Heat pump transfers heat from air to water
Heat pump pushes cold/ dry air out 3
Condenser coil heats water in tank
Backup electric heating elements
Water tank

Figure 2.6: Diagram of Heat Pump Water Heater [8]

HPWHs are not a new technology - the first patent was made in 1935. HPWHs failed to achieve any significant market penetration in the 20th century due to their high costs, maintenance requirements, and noise. As of 2018, less than 2% of water heaters are equipped with heat pumps. However, advances in technology have made them more affordable and convenient, and new legislation has either incentivized or required them in many parts of the country. All water heaters with tank sizes over 55 gallons manufactured since 2015 are effectively required to have heat pumps by federal law [8]. The BPA predicts the market penetration of HPWHs to increase steadily over the coming decades, reaching 31% by 2039, as shown in Figure 2.7.

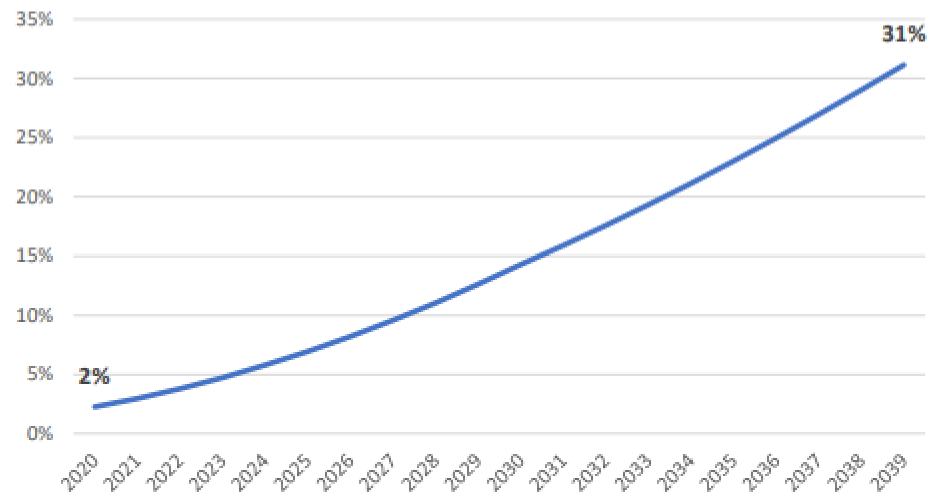


Figure 2.7: HPWH Market Penetration Forecast [6]

Although HPWHs are more energy-efficient than resistive EWHs, they are less useful from a DR point of view. They aren't capable of absorbing as much energy from the grid, they take much longer to turn on, and the rated power of a heat pump is usually much less than the rated power of a resistive heating element. The BPA points out that, even though EWHs will only make up 69% of water heaters in 2019, they will still represent 80% of the utility benefits. Additionally, because HPWHs reheat more slowly, utilities must be more conservative when shedding them to avoid hot water shortages [6].

2.5 Frequency Response

The North American power system is divided into four interconnections: Eastern, Western, Texas, and Quebec. Within any given interconnection, the electrical frequency is the same everywhere, and nearly all of the coupled generators are rotating in synchronism. Every machine stores kinetic energy in its inertial mass. Because there must always be a balance between power generated and power consumed in the interconnection, sudden changes in either load or supply manifest as changes in frequency. These changes typically happen following the accidental or intentional disconnection of a generator or load from the grid. If a generator is disconnected, other generators inject some of their stored kinetic energy into the grid to compensate for the lost generation capacity, resulting in slower rotation. Likewise, if a load is lost the generators speed up, retaining their kinetic energy [22].

Severe changes to the grid frequency can result in power outages and damage to large steam turbines. There are three stages of frequency control used to arrest the change in frequency and return it to its normal steady-state level. These stages are divided into overlapping windows of time over the course of a frequency disturbance and the following recovery. An example of a frequency disturbance displayed side-by-side with the three control stages is shown in Figure 2.8.

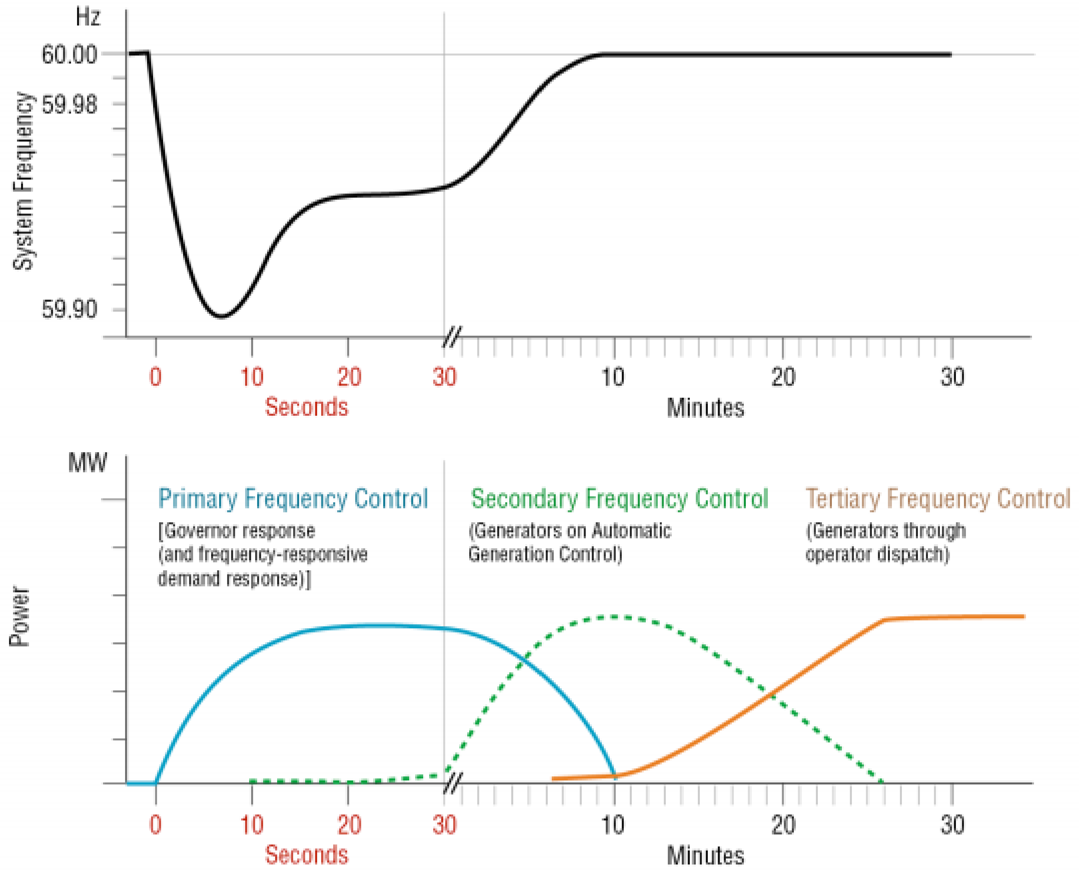


Figure 2.8: Stages of Frequency Control during a Disturbance [9]

Primary Frequency Control, more commonly known as frequency response, is the fastest-acting stage and ramps up within the first few seconds of a frequency disturbance. Traditionally, frequency response is provided by two sources: governor and load reactions. Generator governors respond automatically and continuously to changes in speed by adjusting the generator's mechanical energy input, such as by moving the wicket gates to increase water flow to a hydropower turbine, or increasing the fuel consumption of a coal-fired power plant. Some loads are equipped with underfrequency relays that disconnect them if the grid frequency drops too far for too long. Additionally, the motors that make up a large portion of most grids' total load consume less power as the frequency decreases, and more power as the frequency increases, providing a natural buffer against frequency change. DR resources that can be dispatched within seconds of a frequency disturbance, such as EWHs, would belong to the Primary Frequency Control. An interconnection's total frequency response capacity can be characterized as MW/0.1 Hz. For example, the Western Interconnection has a frequency response of roughly 1,500 MW/0.1 Hz, so the loss of a 1,000 MW generator would result in a frequency drop of about 0.067 Hz [23].

Once a frequency change has been arrested, the governors continue to respond to the difference in power with some time delay. The frequency only partially recovers and oscillates around an intermediate value for a transient period that lasts about 20 seconds after the initial disturbance. An individual frequency disturbance and the associated response are characterized by the North American Electric Reliability Corporation (NERC) using three values, referred to as A, B, and C. The A-value is the average pre-disturbance frequency. The B-value is the average frequency during the post-disturbance transient period. The C-value is the nadir frequency for a negative disturbance, or the zenith for a positive disturbance. The time windows for averaging the A-value and B-value have not been standardized and are decided on by the regional Balancing Authorities (BAs). The A-value period is typically about 16 seconds, and the B-value period is between 20 and 52 seconds. All BAs in an interconnection must use the same averaging periods to provide consistent results [24]. A comprehensive plot showing the frequency response to the loss of a 1000 MW generator can be seen in Figure 2.9. At the moment of loss, the power deficit goes directly to 1,000 MW, but then decreases as the load response lowers the total demand. The balancing inertia tracks the power deficit perfectly until the governor response takes over by increasing the other generators' total power output after a brief time delay. At the end of the transient period the balancing inertia settles back to zero while the governor response tracks the power deficit.

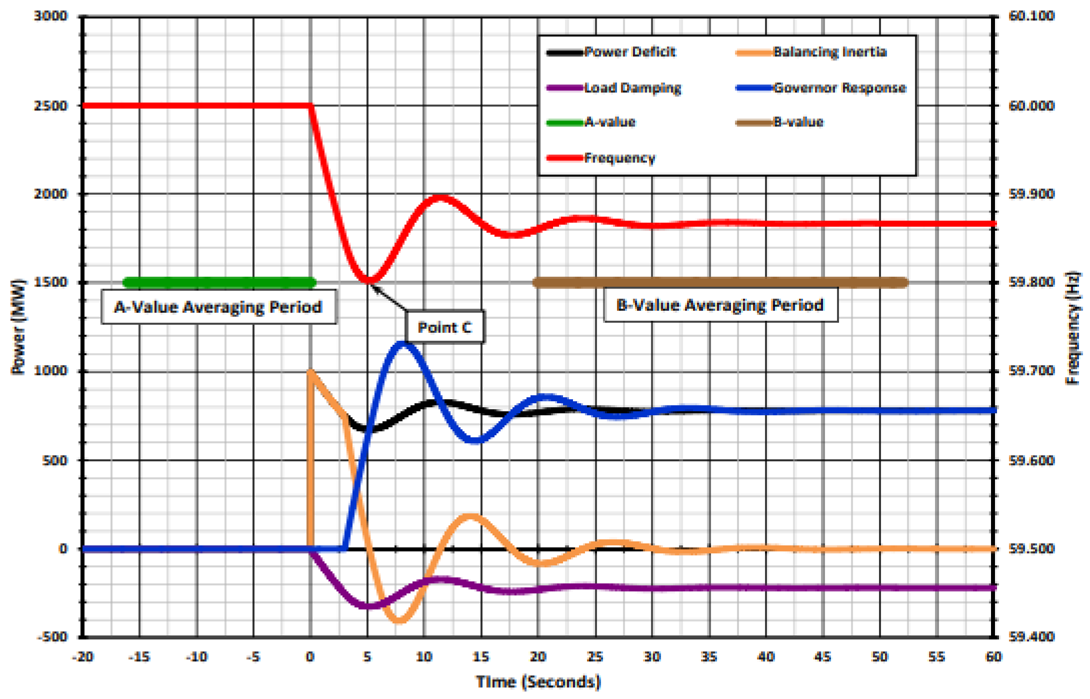


Figure 2.9: Components and metrics of traditional frequency response [24]

The ongoing retirements of synchronous generators around the country and the increase in power provided by sources with little or no mechanical inertia erode system frequency response. While NERC's 2018 long-term reliability assessment predicts that the national grid frequency response resources are expected to remain adequate through 2022, the large-scale transitions to renewable resources may put the system in serious jeopardy in the following decades. Type 1, 2, and 3 wind turbines use induction generators and are not mechanically synchronous with the grid electrical frequency, and cannot provide governor response in the same way as more traditional power plants. Type 4 wind turbines, which connect to the grid through electronic AC/AC power conversion, and DC energy sources that require inverters, such as battery systems and PV, do not naturally provide any inertia to the grid.

Many generation sources can emulate inertia with controllers that monitor grid frequency and modify the power output in response to disturbances. This practice is known as fast frequency response (FFR). A 2018 order by the Federal Energy Regulatory Commission (FERC) mandates all new generating facilities to provide some form of frequency response capability. While this does not apply to distributed sources such as residential rooftop PV installations, it does set the precedent that even renewable energy providers are obligated to maintain this ancillary service [25].

Frequency response capacity has a high monetary value. The Salem Smart Power Center (SSPC) is a PGE R&D project located in Salem, Oregon, and is home to a 5 MW, 1.25 MWh battery energy storage system (BESS). One of the goals of the SSPC is to determine the value of different ancillary services, including frequency response. While the SSPC only engaged in frequency response for a total of 17 hours in a full year, it was still found to be more valuable than all other services combined, as shown in Table 2.1.

Service	20-Year Benefits (SSPC only)
Arbitrage	$746,299
Demand Response	$428,155
Regulation Up	$374,609
Regulation Down	$656,706
Primary Frequency Response	$3,568,826
Spin Reserve	$100,622
Non-Spin Reserve	$46,124
Volt-VAR / CVR	$393,619
Total	$5,865,846

Table 2.1: Monetary value of different SSPC services, projected over 20 years [27]

Figure 2.10 shows that, while frequency response and demand response have high monetary value compared to other services, they only make up a small portion of the SSPC's annual application hours, making them watt-for-watt the most valuable ancillary services by a very wide margin.

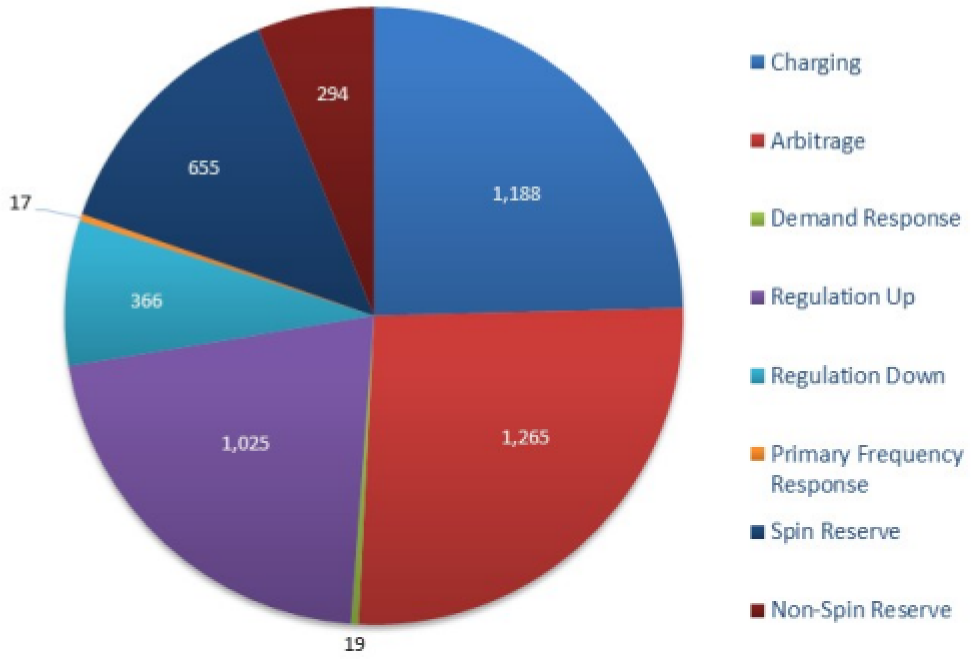


Figure 2.10: Hours required for different SSPC services in one year [27]

While NERC requires utilities to provide frequency response services, they do not specify when an event is occurring, or exactly how the response must be executed. The SSPC's event detection and response algorithms were created by PGE, and only react to negative frequency deviations. As soon as an event is detected, the battery output ramps up as quickly as possible to nearly full power. The power remains at this level for 3 minutes, before slowly ramping down as the Secondary and Tertiary frequency control resources take over. 300 kWh of the SSPC battery's storage capacity is reserved for frequency response at all times, which is roughly equal to the total energy discharged during a response. An example of the SSPC responding to a real frequency disturbance is shown in Figure 2.11.

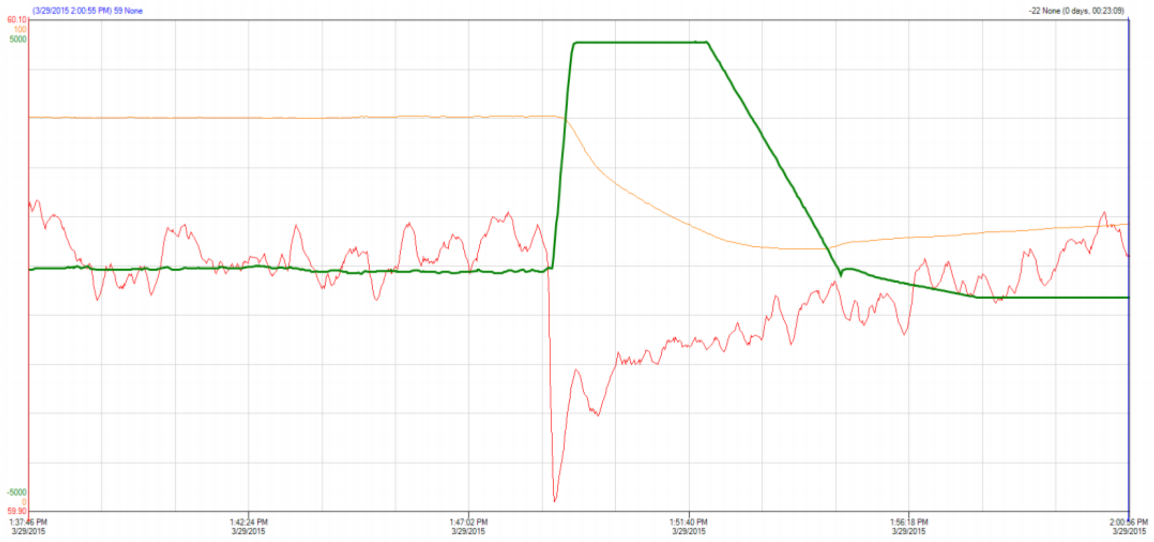


Figure 2.11: SSPC response to a frequency disturbance. The grid frequency is shown in red, the SSPC power output in green, and the battery's state of charge in yellow. [27]

Battery inverter systems are one of the best options for providing frequency response to negative frequency deviations because they can maintain an energy reserve to discharge to the grid whenever needed. Water heaters cannot generate power, but they can change the overall load. In the event of a negative frequency deviation, water heaters would need to turn off in order to provide frequency response. However, because water heaters are already off for most of the time [4], a very large number of devices would be required to provide a significant response to a frequency decline. However, EWHs are an ideal resource for responding to upward frequency disturbances, such as the one shown in Figure 2.12. Caused by a sudden loss of load or spike in generation, these disturbances are less common but pose the same threats to grid reliability as downward disturbances. Arresting the positive frequency change requires a decrease in generation or increase in load. Because they ramp up to their full power almost instantaneously, EWHs are capable of responding to these events even faster than battery inverter systems. Additionally, if this service is delegated to EWHs, battery inverter systems on the same network won't need to maintain any charging headroom.

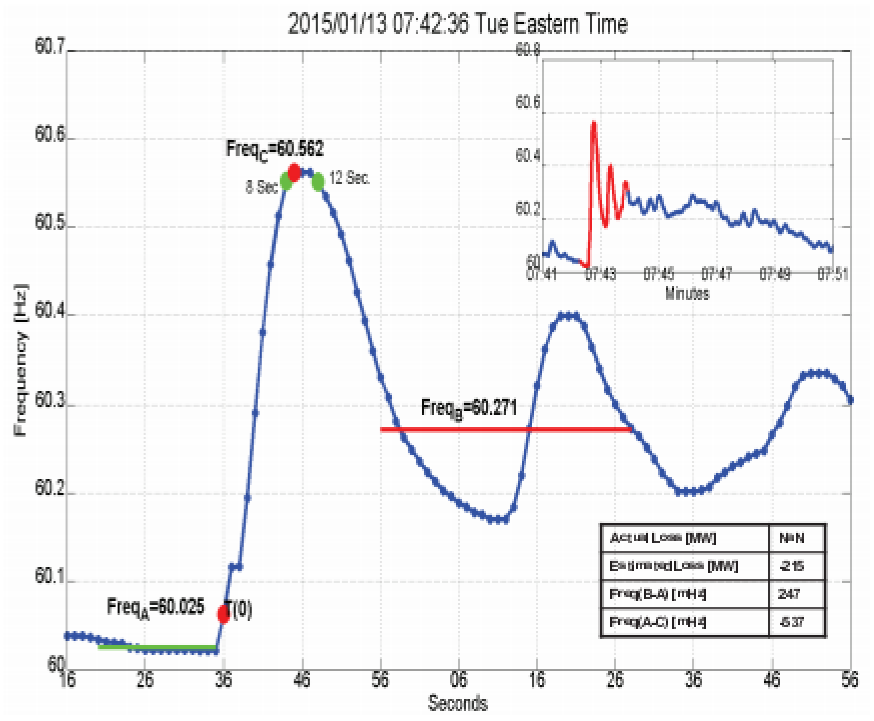


Figure 2.12: Real Example of Upward Frequency Disturbance Event [1]

Frequency disturbance events are fairly recognizable after they have occurred. However, effective frequency response requires that events be detected within the first few seconds. This requires service providers to create some automated event detection algorithm. Because the grid's generation/load balance is in a constant state of change, a detection algorithm that is too sensitive may generate many false positives. For example, the SSPC successfully responded to 15 out of 18 registered frequency disturbances over the course of 10 months in 2016, equating to an 83.3% detection rate, but also triggered erroneously nearly eight times as often [27]. Responses to these false positives are both a waste of resources and an additional stress on the system. A balance must be struck in order to minimize both false negatives and false positives. NERC provides little guidance for detecting frequency events in real time, but does establish a minimum stable frequency for each Interconnection. If the grid frequency does not drop below this "floor frequency" there is no need for a frequency response. In the Western Interconnection, this frequency is 59.976 Hz. Frequency response measures should be able to detect and arrest a frequency decrease somewhere between 59.976 Hz and 59.5 Hz, where the last-resort measure of under-frequency load shedding (UFLS) begins [24]. NERC generally refers to frequency changes as +/- deviations from 60 Hz, so equivalent thresholds can be applied to upward frequency disturbances.

3 Methods

3.1 Methodology

This work seeks to evaluate the usefulness of an aggregation of water heaters in providing demand response and frequency response services. By nature, a large number of devices is required. Conducting a study such as the BPA pilot project requires the participation of utilities and a large number of their customers, which far exceeds the means and scope of this research. Additionally, the internet-of-things network that is used to exchange telemetry between the aggregator and devices is a novel approach that has not yet expanded beyond a local area network (LAN). Obtaining, installing, and operating a large number of water heaters would also be very expensive and labor intensive, require an extravagant amount of space, and consume a large amount of water. For these reasons, virtual devices are used primarily here, rather than physical devices. Developing these virtual device models does require the characterization of physical devices, and a test station was built for that purpose. A thermal model for a small electric water heater was developed by a previous study [3]. A significant part of the work presented here builds on that original model and expands it to larger resistive and heat-pump water heaters. Evaluation of the CTA-2045 interface is also necessary, both as a means of controlling and monitoring the devices, and because the exchange of telemetry is limited by the interface capabilities and functionality. Once emulators have been developed that adequately reflect the characteristics and behavior of the physical devices, a large number can be run and connected to the aggregator simultaneously. The aggregator is unaware that these devices are virtual, and they send the same information that a CTA-2045 interface would provide. The starting energy condition and usage schedule of each virtual device is randomized to simulate the stochastic conditions of real appliances. The net power consumption and energy capacity of the virtual devices can then be used to evaluate the effects of demand response and frequency response efforts by the aggregator. In this study, the goal for demand response is to shift power consumption away from when it would normally occur. The goal for frequency response is to detect both upward and downward frequency disturbances and react accordingly by changing the net load.

3.2 Water Heater Test Station

The test station built for this study has two 50-gallon water heaters, one resistive (EWH) and one with a heat pump (HPWH). Each water heater is controlled by a Raspberry Pi 3 Model B single-board computer, which is also referred to as the distributed control system (DCS). The DCS communicates serially through the water heater CTA-2045 interface and is also connected to the lab LAN via Wi-Fi. The DCS serves as the UCM by connecting to the aggregator IoT network, receiving controls from the aggregator and converting them into appropriate CTA-2045 commands for the water heater, and querying the water heater for information. Users can also control and request information from the water heater directly through the DCS using its command line interface. The DCS also measures and records data that is not available through the CTA-2045 interface, such as water temperature and electric current. The DCS also simulates usage of the water heater by executing automated water draw schedules.

3.2.1 Water Heaters

The test station EWH is an A. O. Smith model PXGT-50 100, with two heating elements (upper and lower) each rated at 4500 W if supplied by a 240 V service. Only one element can be energized at a time, so the full rated power is 4500 W. Using the front panel, users can adjust the temperature setpoint, enable or disable remote control through the CTA-2045 interface, or put the heater into vacation mode. If remote control is disabled, information such as available import energy can still be queried through the CTA-2045 interface, but all commands will be ignored. In vacation mode, the temperature setpoint drops to 60°F to minimize energy consumption.

The station HPWH is an A. O. Smith model HP10-50H45DV 120. The heat pump compressor rated current is 1.7 A, which yields a rated power of 408 W for a 240 V supply. In addition to the heat pump, it has two 4500 W resistive elements. Enabling remote access through the CTA-2045 interface is not as simple as with the EWH. According to the manual, remote access can be enabled by holding down a button until the letters "r.a." appear on the screen. However, due to what appears to be a manufacturer's error, the logic is reversed: remote access is disabled when "r.a." is on the screen, and enabled when it is not.

The HPWH front panel provides more options: in addition to changing the temperature setpoint, the user can switch between four different modes. In 'Hybrid' mode, the heat pump is primarily used for heating, but the upper resistive element is also used if the temperature drops below a threshold. In ‘Efficiency’ mode, only the heat pump is used. In ‘Electric’ mode, the heat pump is not used at all, and the device behaves like a traditional EWH. In ‘Vacation’ mode, the temperature setpoint drops to 60°F. Because these modes change the HPWH behavior and characteristics, it would be desirable to include all of them in the model. However, the device does not report what mode it is in over the CTA-2045 interface, even though message structures are provided for this in the standard [10]. It is therefore impossible for various devices in a network to inform the aggregator what modes they are using, and the aggregator must treat them all the same. No data is available on the probability of each mode being used, so for the HPWH model only the default ‘Hybrid’ mode was considered.

3.2.2 Temperature Sensors

All water temperature measurements for the study used Maxim DS18B20 waterproof digital sensors. The DS18B20 can communicate with the DCS using the Raspberry Pi dedicated One-Wire GPIO pin. Because each sensor has a hard-coded unique ID number, the DCS can operate multiple sensors simultaneously using the same pin and signal line. A free Python package allows easy measurement using a function that only requires the ID number of the desired sensor. Temperature sensors were only placed in the EWH. The anode rod, normally used to prevent corrosion of the tank, was removed and replaced with a series of sensors spaced along a pole. The anode rod of the HPWH is inaccessible, so no temperature measurements of its tank were made.

3.2.3 Valves and Flow Meters

To allow for automated water draws, each DCS controls an electric ball valve on the water heater outlet. The valves operate on a 120 V supply, so the DCS uses its 3.3 V GPIO output to control a solid state relay that in turn provides 120 V to the valve.

A flow meter records usage data. The flow meter is an Omega FPR303. Water flowing through the meter turns a propellor. A magnet embedded in one of the propellor blades passes by a Hall effect sensor once with every rotation, generating a voltage pulse. The number of rotations per gallon of water flow is dependant on the individual meter and is printed on the housing. During a draw event, the Python program that controls the valve calculates the total flow by counting the rising edges of the flow meter pulses. The flow meter runs on a separate 24 V power supply, and the output is divided down to 3 V to avoid overvoltaging the GPIO pin. The flow meter's sensor is current sinking, so a pull-up resistor at the GPIO pin is also required.

3.2.4 Current Transducers

Although all CTA-2045 devices are obligated to report their power consumption, validating these measurements falls within the scope of this study. The water heaters in the test station are powered by a 240 V split-phase service. No neutral line is connected, so the current flowing through each hot line must be equal. To measure power consumption, a current transducer is placed on one hot line of each water heater. The current transducers have adjustable input current ranges, and convert the RMS current to a proportional DC voltage between 0 and 5 V. With the input range set to 0-50 A, the output voltage must simply be multiplied by 10 to determine the current. The calculating program assumes a stable RMS voltage of 240 V, and multiplies the measured current by this to obtain the power consumption.

Because the Raspberry Pi does not have any analog inputs, an analog to digital converter is required. The Microchip Technology MCP3008 is a cheap ADC with 10-bit resolution that can communicate with the Raspberry Pi's SPI interface using a popular free C++ library called WiringPi. Initial measurements with the MCP3008 were inaccurate, but this was corrected by adding capacitors at the input and power supply pins. The capacitor at the input pin allows the MCP3008 sample-and-hold capacitor to charge more quickly, and the capacitor at the power supply smooths out fluctuations in the supply voltage, which is also used as a reference.

3.2.5 Serial Communications

The CTA-2045 interface for water heaters requires serial communications via the RS-485 standard, which the Raspberry Pi is not capable of by itself. The first attempted solution was to use a separate signal conditioning circuitboard to interface between RS-485 and the Raspberry Pi's UART pins. While the Raspberry Pi and EWH were able to exchange some messages, there was a high rate of error and unintelligible transmissions. A USB to RS-485 converter functioned much better, and required less work. Creating and interpreting CTA-2045 messages manually was also very difficult and tedious, which was exacerbated by mistakes in the CTA-2045 manual (one typo gave the wrong value for the communications status update, which made all commands return override messages). Establishing the communications functionality required for this study would have been a massive undertaking of its own if not for the CTA-2045 C++ library published by EPRI. Once this library was implemented, most of the communications became automated. Erroneous reads still occur occasionally, but are infrequent, with typically less than one occurrence during a full week of constant operation.

3.3 Device Responses to CTA-2045 Commands

One of the purposes of the DCS is to convert instructions from the aggregator into the most appropriate CTA-2045 command. As far as the aggregator is concerned, water heaters are assets that can be either importing their full rated power (on) or not (off). Relevant instructions for water heaters therefore come as a request to either turn on or turn off. The asset's ability to follow these instructions is dependent on its current energy state and the temperature thresholds set by the manufacturer. Each water heater has a default regulation range bounded by upper and lower thresholds, as shown in Figure 3.1. The stored energy bounces back and forth between the thresholds as the tank is heated and water is drawn.

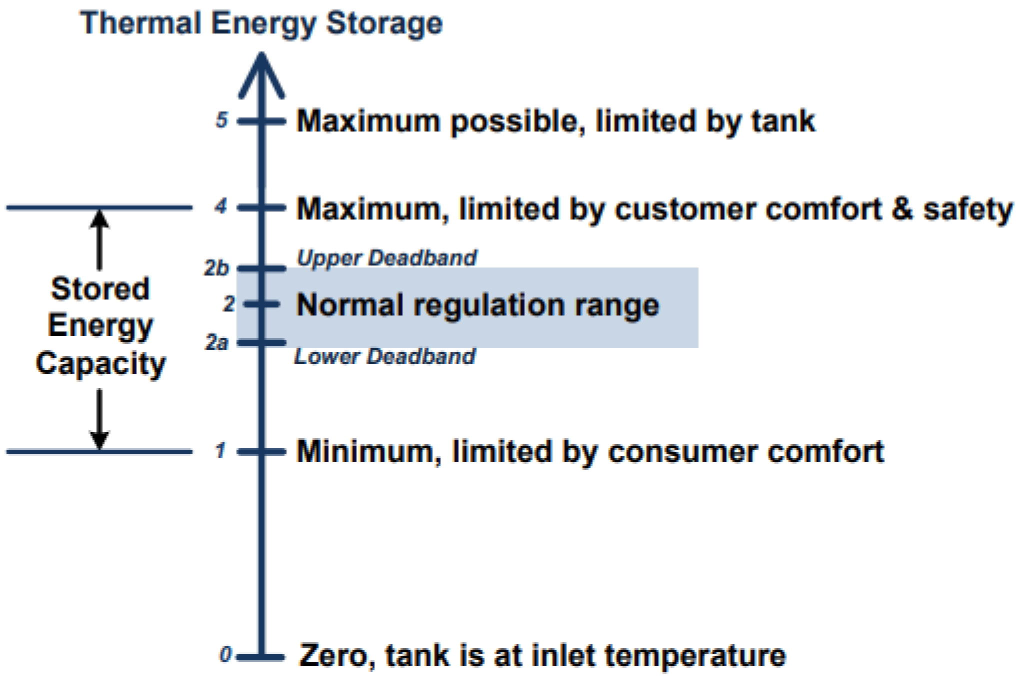

Thermal Energy Storage
5
Maximum possible, limited by tank
4
Maximum, limited by customer comfort & safety
2b
Upper Deadband
2
Normal regulation range
2a
Lower Deadband
Stored Energy Capacity
1
Minimum, limited by consumer comfort
0
Zero, tank is at inlet temperature

Figure 3.1: Energy Diagram of EWH [12]

CTA-2045 commands effectively switch the water heater between different sets of thresholds, therefore shifting and/or changing the size of the regulation range. Because the water heater will always try to keep its stored energy inside of the regulation range, adjusting the regulation range can result in the device turning on or off. Ideally, it would be possible to set very narrow regulation ranges at either extreme of the allowable energy range, which would allow the most precise control of the device. The total range over which it is possible to turn the device on and off freely is referred to as the 'dispatchable' range from this point on. This dispatchable range is limited by the absolute limits on acceptable water temperature set by the manufacturer, and by the thresholds of the different regulation ranges available through CTA-2045 commands. Keeping the dispatchable range as wide as possible maximizes the aggregator's utility by increasing the number of devices that it is able to control at any given time. Additionally, determining the thresholds for the different CTA-2045 commands is necessary in order to properly model device behavior.

3.3.1 Baseline Operation

The thresholds for each CTA-2045 command can be observed by logging the available import energy (how much more energy the device can import before being fully heated, from here on referred to simply as ‘import energy’) over a period of time while drawing hot water occasionally. These thresholds were observed to be the same regardless of the temperature setpoint, so all tests were conducted with a setpoint of 120 °F. To avoid possible correlations between usage pattern and water heater behavior, randomized draw schedules were used. The first test was to see how the water heaters behaved with no active CTA-2045 commands, or the default regulation range. The EWH baseline behavior is shown in Figure 3.2. The water heaters don’t report changes in energy lower than 75 Wh, and only report in 75 Wh increments.

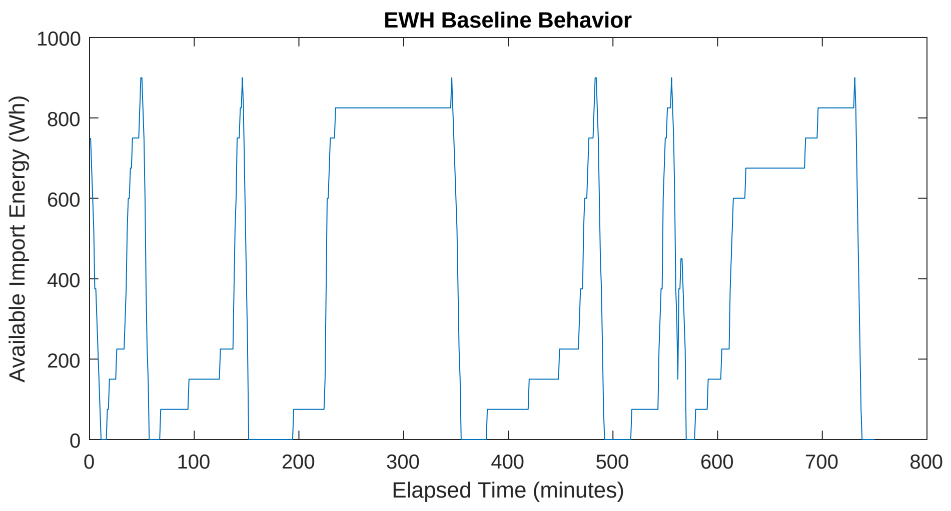


Figure 3.2: Baseline Behavior of EWH

Figure 3.2 shows that the lower baseline threshold is 0 Wh, and the upper threshold is 900 Wh. Note that as the import energy increases, the temperature decreases. Large, sudden increases in import energy are due to large water draws, while smaller changes may be due to small draws or to convection losses. The sampling rate is one sample per minute. Figure 3.3 shows the baseline behavior of the HPWH.

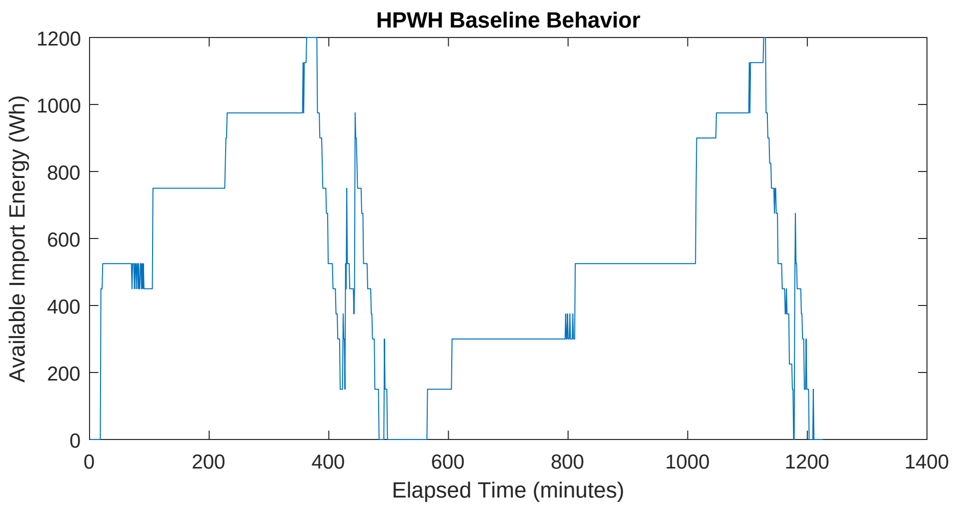


Figure 3.3: Baseline Behavior of HPWH

The HPWH's lower baseline threshold is 0 Wh, and the upper threshold is at 1200 Wh, making the normal regulation 33% larger than that of the EWH.

3.3.2 Shed Operation

The CTA-2045 definition of the *shed* command states that the device should not consume power until the lower energy threshold is reached. Ideally, the entire regulation range is shifted down, so that the water heater can be turned off as long as the current energy is above the upper threshold. The EWH's behavior while receiving a *shed* command is shown in Figure 3.4.

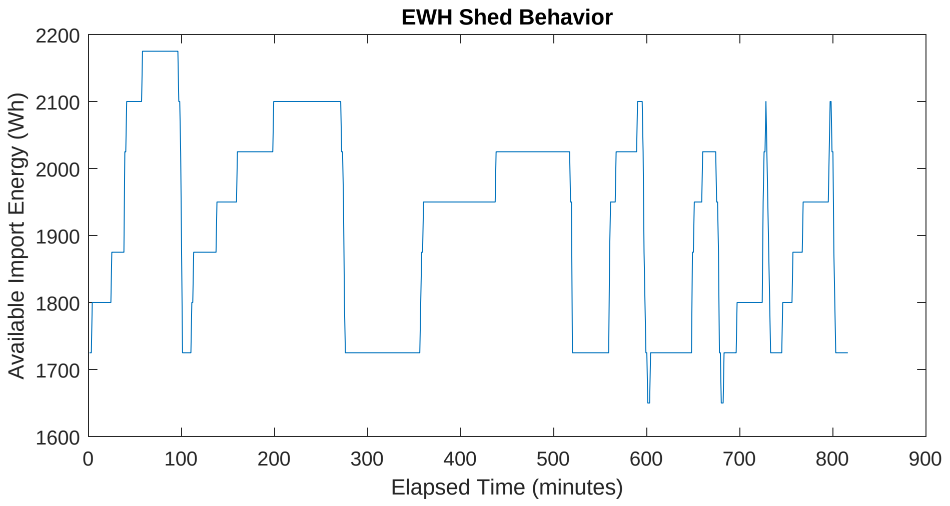


Figure 3.4: Shed Behavior of EWH

The EWH turns on when the import energy reaches 2100 Wh, and turns off as soon as it has decreased to 1725 Wh. This shows that the shedding regulation range has both been compressed to a width of about 375 Wh, and shifted up by about 750 Wh from its normal upper limit. The *shed* behavior of the HPWH is shown in Figure 3.5.

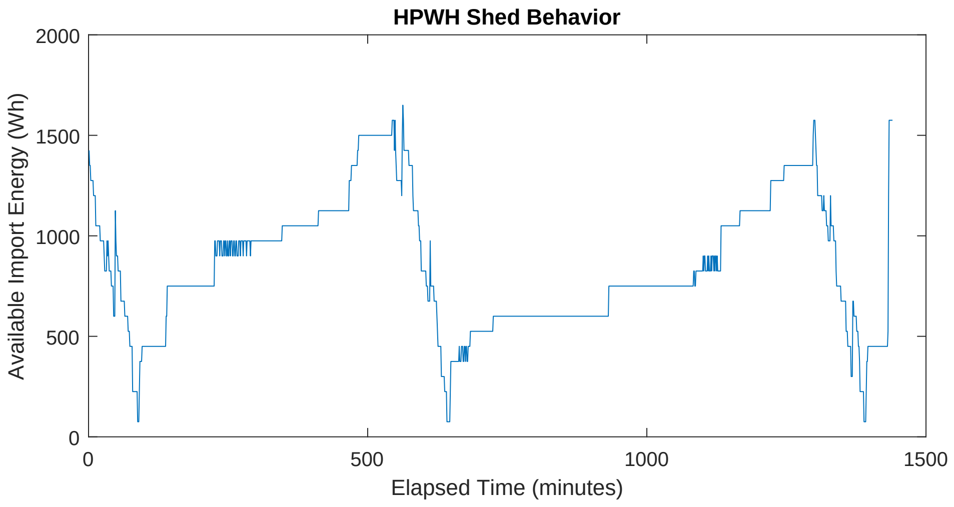


Figure 3.5: Shed Behavior of HPWH

While shedding, the HPWH moves the upper threshold to 1575 Wh. If the available import energy is greater than 1425 Wh, the upper element is used together with the heat pump. However, unlike with the EWH, the regulation range actually gets wider. While the water is allowed to get colder, once the heat pump comes on it does not turn off again until the tank is almost fully recharged. This is a disadvantage for the aggregator for two reasons. First, it means that the *shed* command is unable to turn the HPWH off if it is already on. Second, because the energy the device can import before turning off is a valuable resource, it is preferable to ensure that a large amount of import energy is always available. EPRI also pointed out this design flaw, noting that it reduced the HPWH's usefulness for demand response [15].

3.3.3 Critical Peak Event Operation

The *critical peak event* command functions similarly to a *shed* command, but is intended for non-routine usage and lowers the regulation range more aggressively. The EWH behavior under a *critical peak event* command is shown in Figure 3.6.

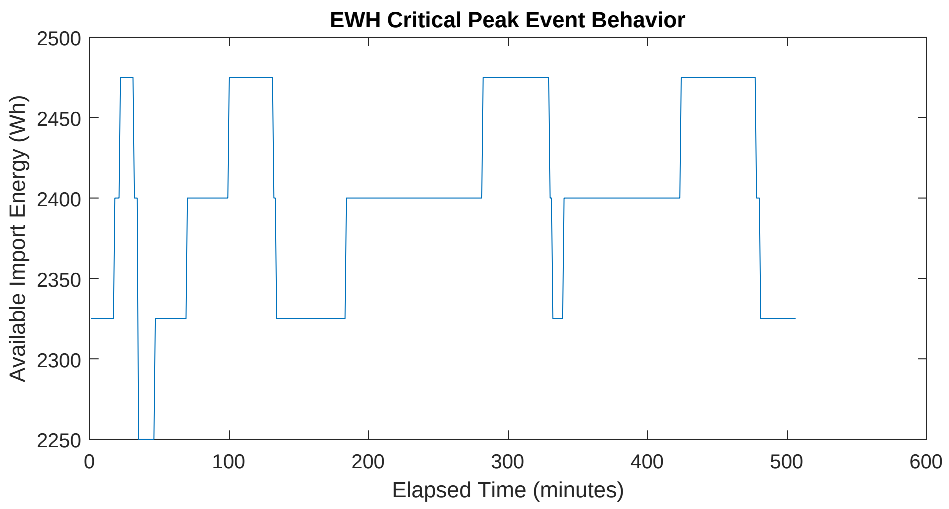


Figure 3.6: Critical Peak Event Behavior of EWH

In the case of the EWH, the *critical peak event* command functions as described in the CTA-2045 manual. The regulation range is narrowed considerably and shifted further down, with an upper threshold of 2475 Wh and a lower threshold of 2325 Wh. This is useful, because it means the EWH can be turned off any time the available import energy is less than 2325 Wh. However, with this much available import energy in the tank, the water temperature is significantly lower than the setpoint. Because roughly 2.44 Wh is required to heat one gallon of water by one degree Fahrenheit, the average temperature of a 50 gallon tank with 2400 Wh of available import energy is about 19.7 °F colder than the setpoint. This is almost certainly enough to cause consumer discomfort, so the use of this command with EWHs should be avoided. The HPWH's behavior while receiving a *critical peak event* command is shown in Figure 3.7.

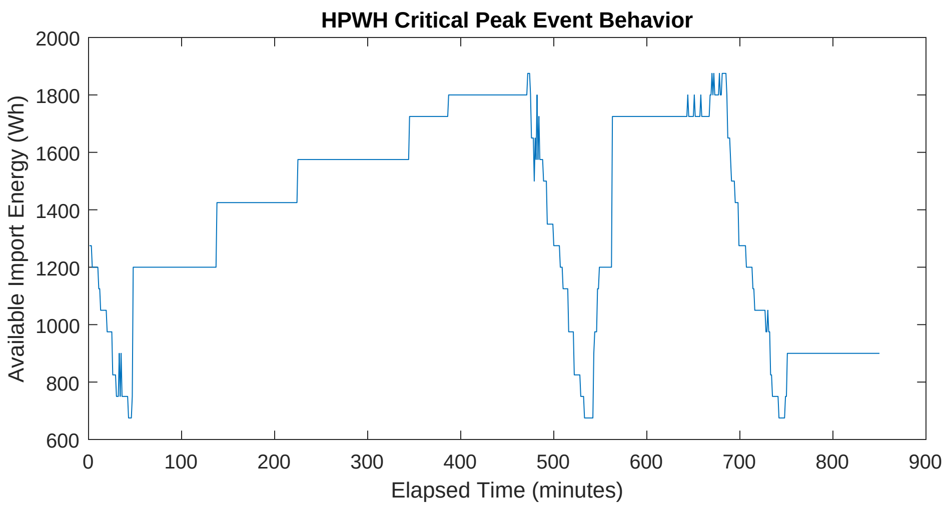


Figure 3.7: Critical Peak Event Behavior of HPWH

The HPWH's *critical peak event* behavior is different in two ways that make it significantly more useful for demand response. First, the upper threshold is much less aggressive, at only 1800 Wh (equivalent to an average temperature decrease of about 14.75 °F. Second, unlike the *shed* command, this command actually narrows the regulation range by decreasing the lower threshold to 675 Wh. This makes the *critical peak event* the only command that is able to actually turn of the heat pump if it is already on. For these reasons, it is a preferable alternative to shedding in all situations. The DCS for the HPWH uses the *critical peak event* command any time it receives an instruction from the aggregator to turn off.

3.3.4 Load Up Operation

The *load up* command instructs the device to absorb as much energy as possible, and is the only CTA-2045 command that is explicitly intended to turn devices on. This is shown for the EWH in Figure 3.8. The device was receiving a *load up* command for the entire plotted period.

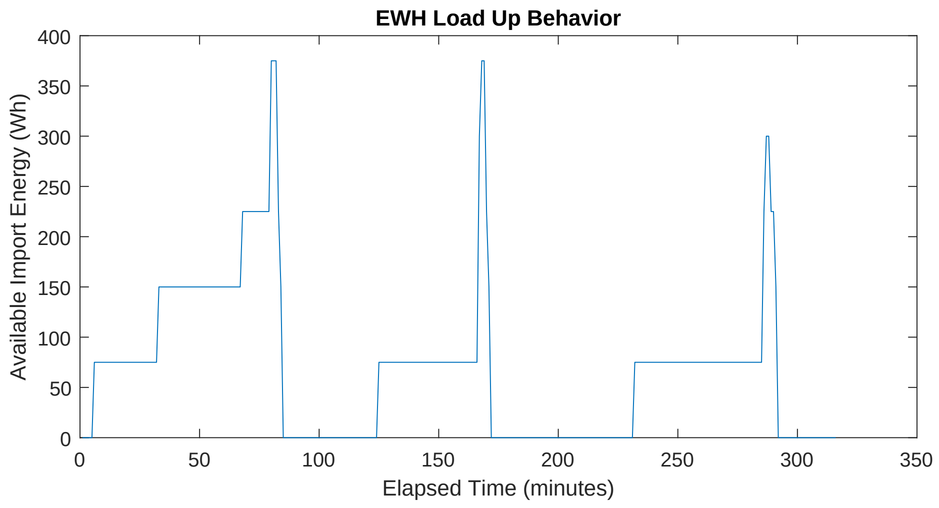


Figure 3.8: Load Up Behavior of EWH

The *load up* command causes the EWH to heat up until the available import energy reaches zero. Any time the available import energy goes above 300 Wh, the EWH comes back on and reheats completely. Because this work treats water heaters as an upward dispatchable resource (the assets are normally off, but can be turned on quickly to add load), the *load up* command is useful for both load shifting and responding to positive frequency disturbances. The HPWH behavior under a *load up* command is shown in Figure 3.9.

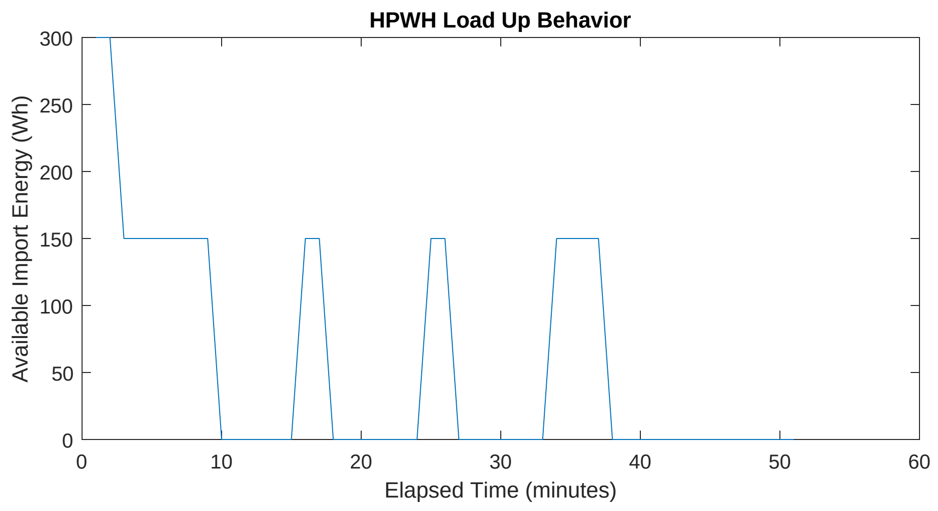


Figure 3.9: Load Up Behavior of HPWH

The HPWH is even more aggressive than the EWH at keeping the stored energy maximized. Any time the available import energy is above 150 Wh, the heat pump comes on and fully reheats the tank.

3.3.5 Summary of Energy Thresholds

The thresholds observed for the different CTA-2045 commands are summarized in Table 3.1. These values were used in programming the emulators to make them respond to commands the same way that physical devices would. They might also be used in future versions of the aggregator to better calculate the total dispatchable load. Note that the regulation ranges of the HPWH are generally wider than those of the EWH, meaning that the aggregator control of the HPWH is much more limited.

	EWH		HPWH	
	Low	High	Low	High
Baseline	0	900	0	1200
Shed	1725	2100	75	1575
Critical Peak Event	2325	2475	675	1800
Load Up	0	300	0	150

Table 3.1: Available Import Energy Thresholds (Wh) under different CTA-2045 Commands

3.3.6 Speed of Response

Physical devices do not respond to commands instantaneously. There is some delay between the time a command is sent, the time the device actually comes on, and the time that it reports that it is on. Figure 3.10 shows the differences between these times for the EWH responding to *load up* and *shed* commands.

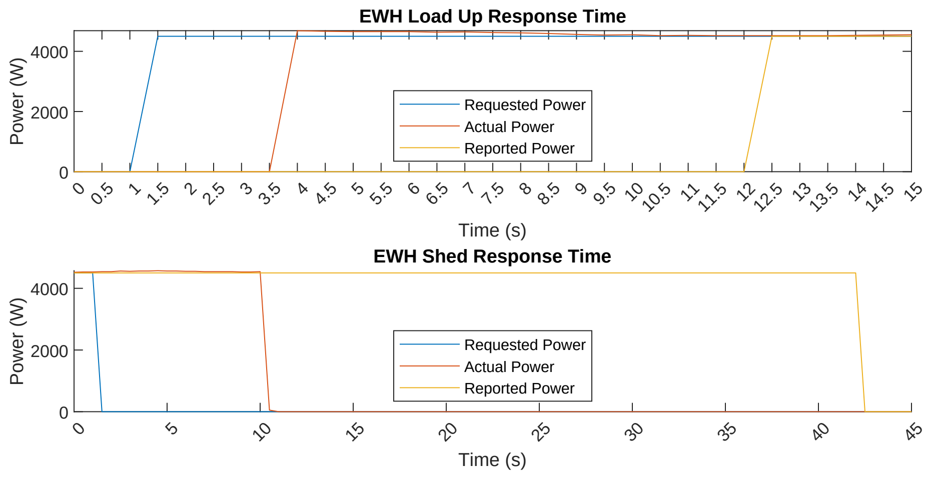


Figure 3.10: Response Times of EWH

The EWH takes 2.5 seconds to turn on after receiving a *load up* command, and 9 seconds to turn off after receiving a *shed* command. Both of these times are short enough to qualify for primary frequency control. There is a considerably longer delay before the EWH actually reports to the CTA-2045 interface that it has changed power consumption. This is the aggregator's only way of knowing how much power its assets are consuming, but the aggregator can also assume that devices are doing what it asked of them, so the delay is only an inconvenience that doesn't change the end result. The HPWH response times are shown in Figure 3.11. A *critical peak event* command is used instead of *shed*, because that is the only effective way to turn off the heat pump.

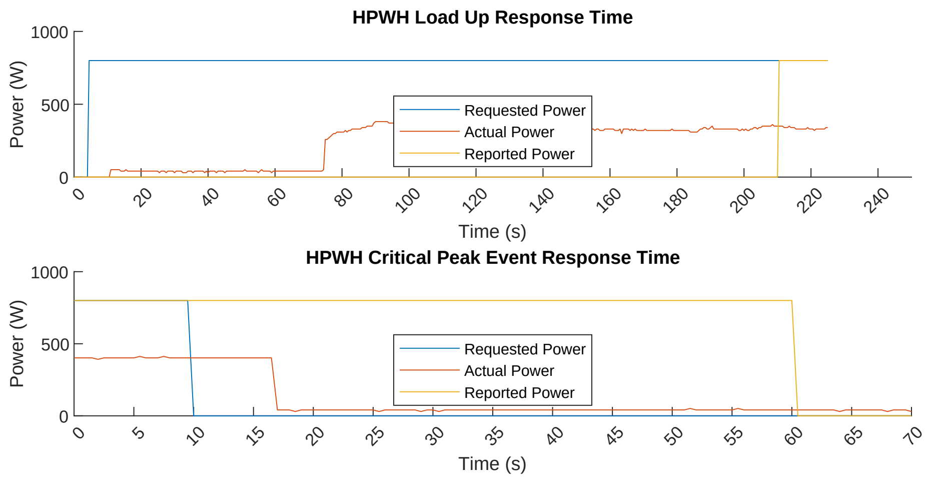


Figure 3.11: Response Times of HPWH

The heat pump takes much longer to come on than the resistive elements of the EWH. The fan must run for more than a minute before the compressor turns on. The delay in reporting is also considerably longer. In this case, the heat pump came on 71 seconds after the *load up* command, and the HPWH did not report that it was on for another 140 seconds. Also note that the HPWH always reports that the heat pump is consuming 800 W when it is on, although the actual power consumption varies based on the water temperature and is usually closer to 400 W. The heat pump is able to turn off much more quickly, with only a 7 second delay from the command. The reported power goes to zero 43.5 seconds after the heat pump actually turns off. The turn-on delay of the HPWH prevents it from being a valid frequency response asset. However, for routine peak shifting, the delay is short enough that it is hardly noticeable.

3.4 Emulators

Programming emulators that faithfully represent the physical devices requires modeling of four main dynamics:

1. Convection losses
2. Change in stored energy from usage events
3. Power consumption and heating
4. Device behavior (when the resistive elements and/or heat pump turn on and off)

The original plan for this work was to build on the thermal model developed in [3], which included these dynamics based on traditional thermodynamic formulae and observation of a non-CTA-2045 15 gallon water heater. However, some assumptions made by this model, such as neglecting the temperature stratification of water in the tank, were dependent on the small tank size. Additionally, because the CTA-2045 interface does not report temperature, the measurements and calculations performed internally to calculate the import energy are opaque to users. Because only the information reported through the interface is available to the aggregator, the model developed here is based on observations of those reported values rather than thermal calculations. Some pieces of the original thermal model were used as starting points, and then adjusted to improve accuracy.

3.4.1 Thermal Model and Temperature Stratification

A side-by-side comparison of the original thermal model and temperature measurements from the 15-gallon EWH it was based on are shown in Figure 3.12.

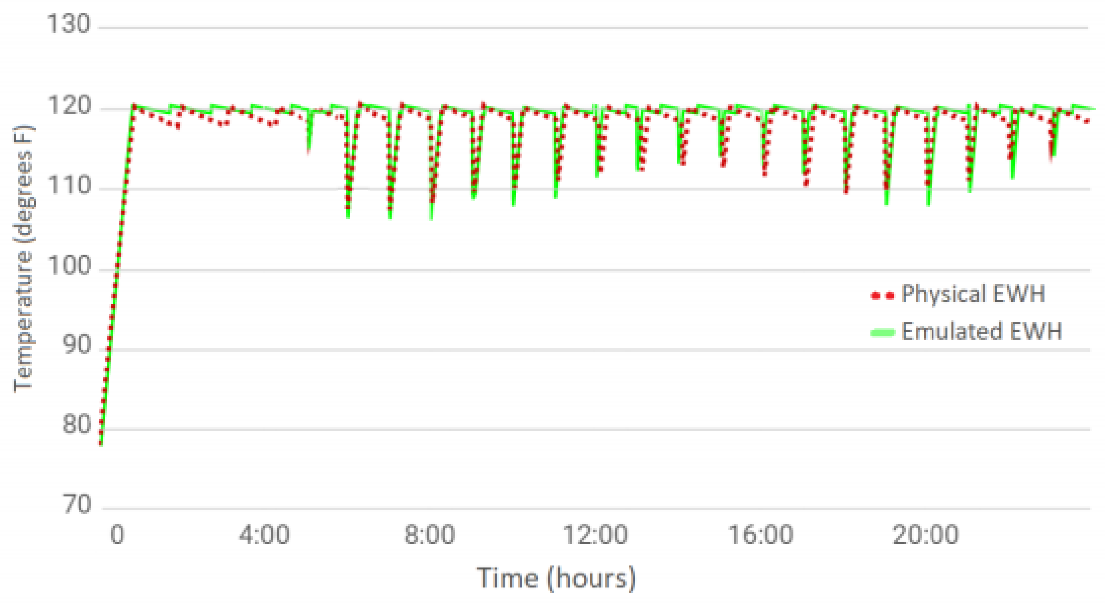


Figure 3.12: Validation of Emulator Using Thermal Model [3]

In [3] the water temperature at several points in the tank was found to be roughly equal, so the outlet temperature was used in calculations as the average temperature for the whole tank. The emulator does a good job of tracking changes from usage events, but the convection loss rate (which was assumed to be linear) is only valid in a very limited range, and only a single, very narrow regulation range is used. Temperature measurements in the 50-gallon EWH used in this work show that very significant stratification occurs in the tank, as shown in Figure 3.13. The temperature sensors are numbered from top to bottom.

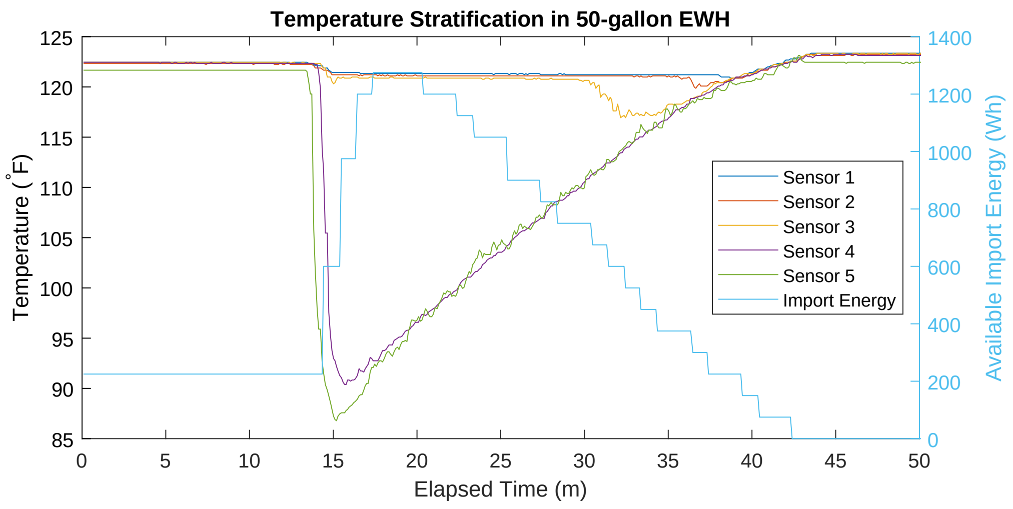


Figure 3.13: Temperature and Import Energy in 50-gallon EWH

A large draw event barely changes the temperature at the upper two sensors, but causes the temperature at the lower two to drop by over 30 °F. The EWH has two built-in temperature sensors at unknown locations in the tank, and uses an unknown process to calculate the import energy. For this research, it was therefore assumed that the import energy reported by the CTA-2045 interface is representative of the average tank temperature.

3.4.2 Convection Loss Rates

The increase of import energy over time in the EWH due exclusively to convection losses is shown in Figure 3.14. The tank was heated completely and then allowed to cool over the course of 36 hours.

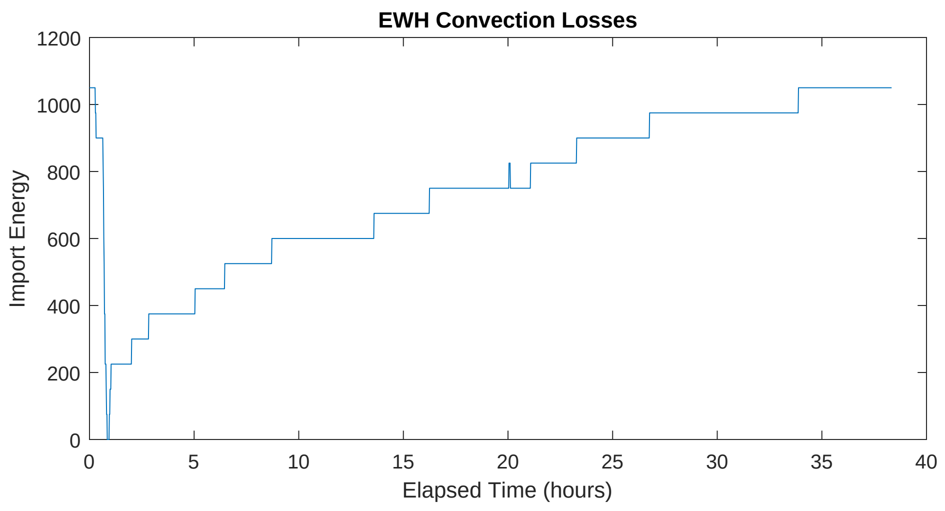


Figure 3.14: EWH Energy Change due to Convection Losses

As expected from basic Newtonian cooling, the rate of change is proportional to the amount of energy still in the tank. To create a function relating convective loss rate to import energy for the emulator, the data in Figure 3.14 were fitted to a curve in MATLAB. This function was then differentiated with respect to time, and the resulting power values were plotted against the original array of import energy values. A new curve was fit to this relationship to determine the convection loss function. This curve is shown in Figure 3.15.

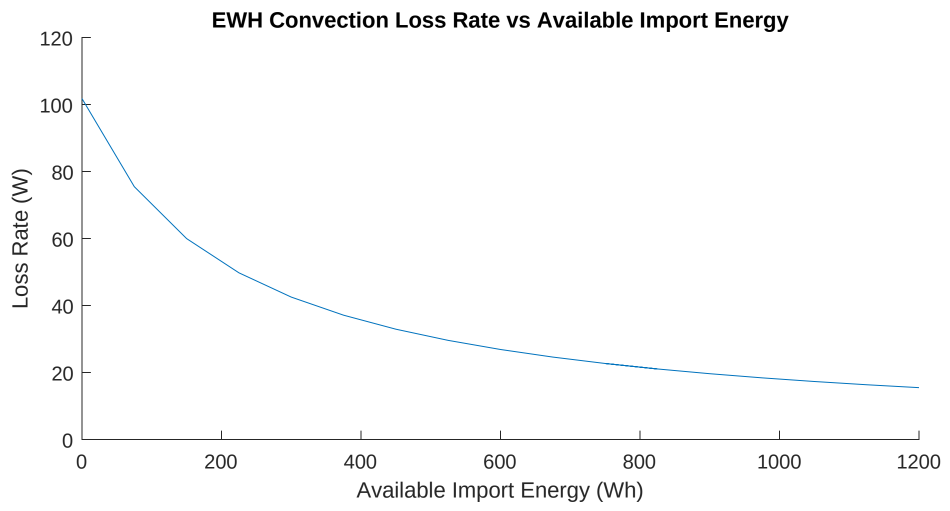


Figure 3.15: EWH Convection Loss Curve for Emulator

The equation for this curve is:

$$\frac{dE}{dt}(E) = \frac{22000}{E + 215} \tag{3.1}$$

The convection losses for the HPWH are significantly less than for the EWH. This is most likely because residual heat in the condenser coils wrapped around the tank provides an extra layer of insulation. The same process of determining the loss rate formula was repeated for the HPWH, resulting in the curve shown in Figure 3.16.

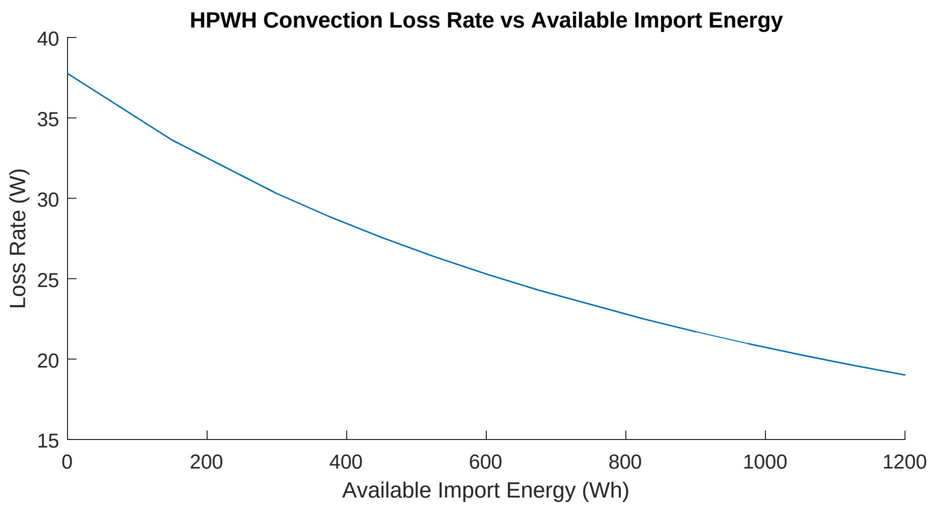


Figure 3.16: HPWH Convection Loss Curve for Emulator

The equation for this curve is:

$$\frac{dE}{dt}(E) = \frac{46000}{E + 1200} \tag{3.2}$$

The HPWH loses less energy than the EWH when it is fully charged, but the two convection loss rates even out as the import energy goes over 1000 Wh.

3.4.3 Heat Pump COP

The power consumed by the heat pump is related to the thermal energy added to the tank by the heat pump COP. The COP is dependent on the temperature gradient, and therefore the amount of energy in the tank. As the water in the tank warms, the heat pump must work harder to keep up the same rate of heating, as shown in Figure 3.17.

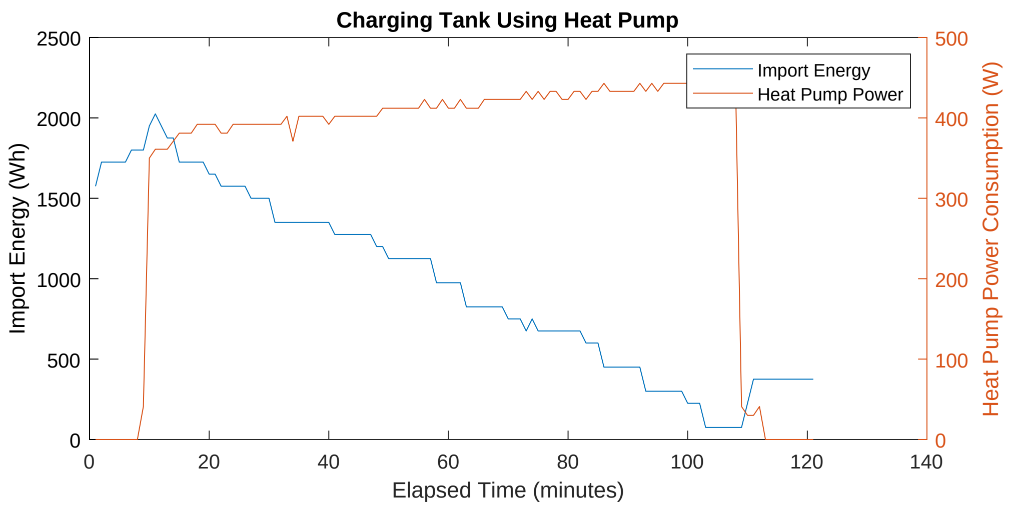


Figure 3.17: Change in Heat Pump Power as Import Energy decreases

The change in import energy appears to be linear with respect to time. Fitting a line to the data in MATLAB, and then converting the slope to watts results in a constant heating rate of 1083 W. The emulators still required a function for electric power consumption, so the observed power consumption was plotted against import energy and fit to another linear equation. The resulting equation is used in the HPWH emulator:

$$P(E) = 454 - 0.0414E \tag{3.3}$$

The rate of heating and observed power consumption were then used to calculate the COP with respect to import energy. All three curves are shown in Figure 3.18.

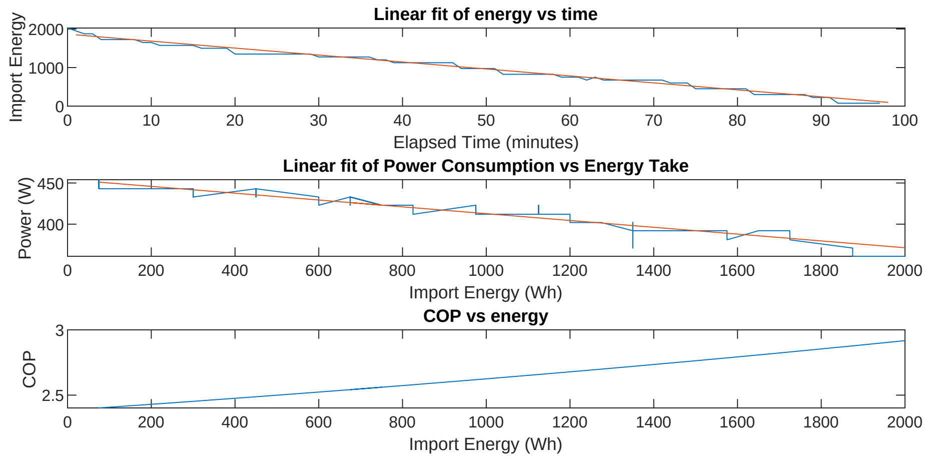


Figure 3.18: HPWH Power, Import Energy, and COP

The COP curve reaches from about 2.4 to 2.9 over the HPWH's typical energy range. This is fairly close to the average COPs reported for similar HPWH models [8].

3.4.4 EWH Emulator Validation

Each emulator program runs a main loop that first checks if any command has been received, and checks the current import energy to determine if the device should be on or not. The loop then calls nested functions to calculate the power consumption, heating of the tank, idle losses, and losses due to hot water usage. The power consumption and rate of heating of the EWH were assumed to each be 4500 W. To calculate usage losses, the emulator approximates the average tank temperature from the import energy, and multiplies the difference between the average temperature and the water mains temperature by the volume of the draw event and the conversion factor of $2.44\frac{Wh}{gal \times {}^\circ F}$. The mains temperature can be set in the emulator configuration file. Because usage losses do not happen instantaneously, the emulator spreads them out over a period of time using a thermal ramp rate, which is also set in the configuration file. The value for this thermal ramp was determined by adjustment over several experiments.

For validation, the emulator and physical device were run with the same schedule of usage events and CTA-2045 commands. The usage schedule was obtained using a probabilistic tool for generating realistic water heater usage schedules created by NREL [4]. The CTA-2045 commands follow a generic peak-shifting dispatch schedule summarized in Table 3.2.

Time Window	Command
Midnight - 6AM	Load up
6AM - 11AM	Shed
11AM - 4PM	Load up
4PM - Midnight	Shed

Table 3.2: Peak Shifting Schedule Used for Emulator and Demand Response Validations

A side-by-side comparison of the emulator and physical EWH is shown in Figure 3.19.

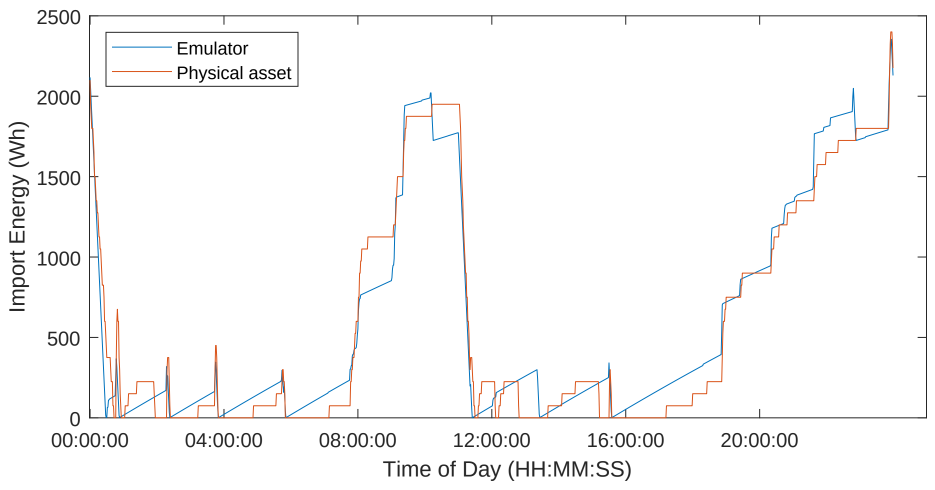


Figure 3.19: EWH Emulator Validation

The emulator is close, but not a perfect match for the physical device. The goodness of fit, determined in MATLAB using a normalized root mean square error (NRMSE) cost function, is 78.66%. While this match is less than ideal, particularly in the period between noon and 4PM, the physical device is also not particularly consistent from day to day. Figure 3.20 shows the physical EWH on seven consecutive days. The day-to-day difference in import energy can be as great as 600 Wh, and there is no clear trend between noon and 4PM. Because this leaves some room for variation, the emulator was determined to be good enough to substitute for physical devices for these experiments.

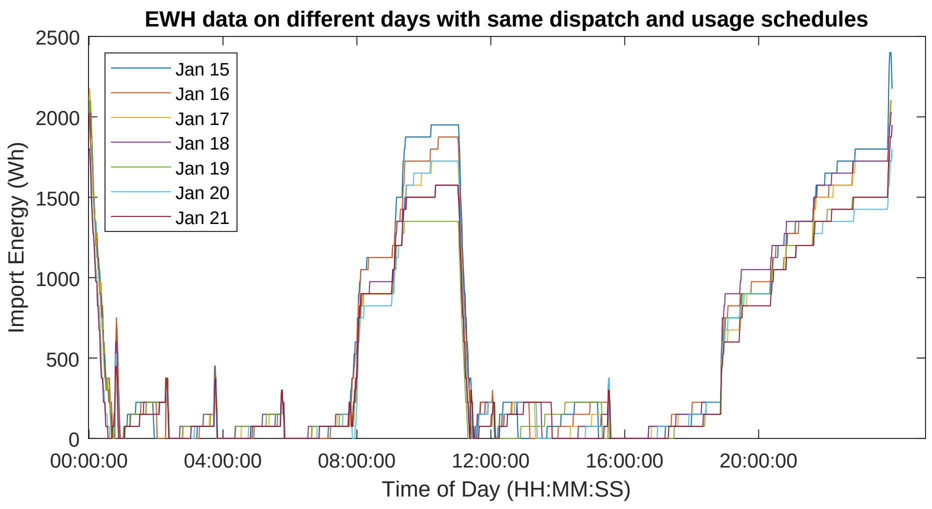


Figure 3.20: Day-to-Day Comparison of physical EWH

3.4.5 HPWH Emulator Validation

Similarly to how the HPWH has lower convection losses than the EWH, the energy losses from usage are also less. Because of this, the first draft of the HPWH emulator was very inaccurate. To correct this, the usage losses in the HPWH emulator are multiplied by an adjustment factor that was determined by repeated testing. The validation test results of the HPWH emulator are shown in Figure 3.21. The emulator matches the main dynamics fairly closely. However, MATLAB calculates a goodness of fit of only 62.24%. The poor match is primarily due to an anomaly between 4AM and 6AM. Although the devices are receiving *load up* commands, the physical HPWH fails to reheat after a usage event. This is most likely because the heat pump is programmed with a maximum number of cycles within some window of time. Because this is only a short-term difference, does not occur frequently, and reverts to normal as soon as the import energy goes over 500 Wh, this is an acceptable difference and the HPWH emulator is also good enough for its role in this research.

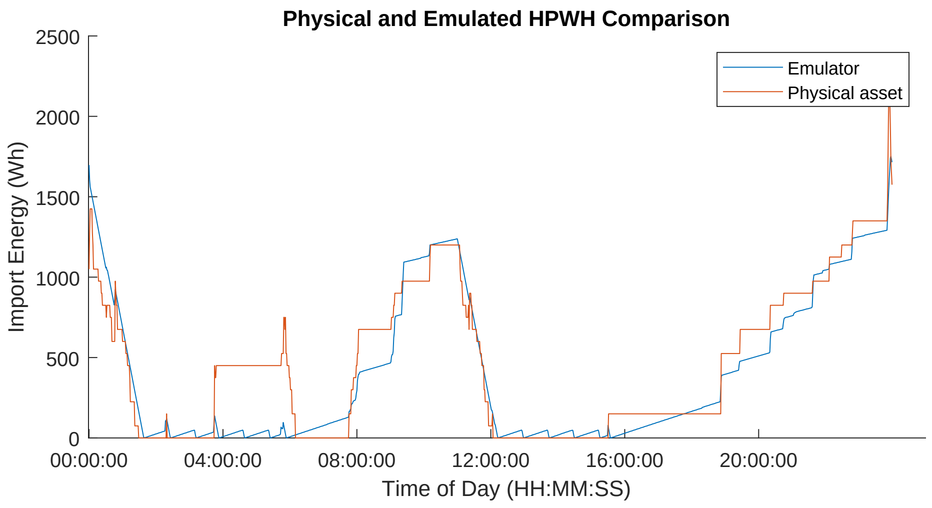


Figure 3.21: HPWH Emulator Validation

3.4.6 Device Randomization

The emulator programs used for aggregate testing take several measures to provide the stochastic differences in usage expected from real customers. The first measure is accounting for household size. The NREL water heater profile generation tool was used to create separate schedules for houses with one to five bedrooms. The U.S. Census reports that, as of 2017, 13% of households had no bedrooms or one bedroom, 26.3% had two bedrooms, 39.6% had three bedrooms, 16.5% had four bedrooms, and 4.4% had five or more bedrooms [7]. Each time an emulated device is created, it receives a schedule based on the probabilities listed here using a random number generator. The second measure is randomization of the starting condition. The initial import energy of a new device is determined using a normal distribution with a mean of 50% and standard deviation of 30% of the rated value. The third measure is randomization of the usage schedule. Both the times and volumes of use are spread over normal distributions. For each usage event, the mean for the time distribution is the base value, and the standard deviation is one half-hour. For the volumes, the mean is the base value, and the standard deviation is 30% of that.

3.5 Aggregator

DERAS, the aggregator used in this work, was first developed and presented in [28]. DERAS is a C++ program that uses Alljoyn, an open source software framework for creating networks of different devices. The user can query information on the total aggregate or on individual devices, directly request the aggregate to begin importing or exporting a specific amount of power, or begin different services. A service for scheduled dispatch commands was already included, but the frequency response service was created as a part of this research.

3.5.1 Digital Twins

In order to include a large number of assets on the network, it is necessary to limit the amount of network traffic created by each device. To accomplish this, DERAS uses digital twins to represent devices in its network. Each time a device is added to the network, DERAS creates a digital twin for it locally, which serves as a placeholder for the asset's energy and power properties. For example, if an EWH joins the network, DERAS queries its rated power and energy capacity as well as its current state, and stores all of these values in a new digital twin. If the EWH is instructed to be on, DERAS automatically begins updating the energy value stored in the digital twin using the rated power. Digital twins also include an idle loss rate. The digital twin is not intended to accurately track the state of the asset over a long period of time, but rather only to reduce the necessary frequency of metrology requests. For this research, the DCSs were programmed to send property updates once every hour, or any time the device import energy changed by more than 10% of its rated value. In this way, any sudden changes to the device energy state, such as a large usage event, are soon reported to the digital twin. However, for most of the time, when very little is happening, the device only needs to send an update once every hour. This dramatically reduces the total network traffic necessary for a fairly accurate real-time assessment of the aggregate. Part of the work presented here is to validate that the digital twins track the devices sufficiently.

Using the available machines and software, it was possible to include about 100 assets in the DERAS Alljoyn network. Adding more assets made the system increasingly unstable, resulting in communication timeouts and disconnections. Therefore, all tests for this research were performed using 100 assets.

3.5.2 Peak Shifting Service

To test the load-shifting capacity of an aggregate of water heaters, the DERAS peak shifting service was used. This experiment used the same generic 24-hour dispatch schedule used in the emulator validation tests and shown in Table 3.2, with the goal of shifting power consumption away from the traditional peak hours in the morning and evening. First, a baseline test was run to show the pattern of the aggregate if no commands were sent. The aggregator was then programmed to send the same scheduled commands to all devices in its network simultaneously. This test was first done exclusively with EWHs, and then with a mix of EWHs and HPWHs. Because [6] predicts a 31% market penetration of HPWHs by the time that CTA-2045 interfaces are essentially ubiquitous, the mixed test was done with an aggregate of 69 EWHs and 31 HPWHs.

3.5.3 Frequency Response Service

The DERAS frequency response service was created with two algorithms: one for detecting positive and negative frequency disturbance events, and one for responding to them. In actual use, DERAS would receive second-by-second updates of the grid frequency in order to detect events. For this research, DERAS was provided with a data file containing actual frequency data from previous disturbances provided by PGE. PGE also provided the programmable logic controller (PLC) program used for frequency event detection by the SSPC, which was used as a template for this work.

3.5.3.1 Frequency Event Detection

The foundation for the DERAS event detection algorithm was a translation of the SSPC program from ladder logic (a diagram-based programming language that loops multiple processes simultaneously) to procedural C++ code. However, precisely converting ladder logic to C++ is a tricky process, and some of the algorithm's functionality was changed in translation. Because frequency events are characterized both by the magnitude of their deviation from the base frequency and their slew rate, these two parameters were left as arguments for the event detection function so that its sensitivity could be adjusted. To determine the most appropriate settings, a parametric sweep of these parameters was run using data from 18 different frequency responses from the SSPC in 2018 and 2017. For each combination of settings, the program's event detection was scored based on its similarity to the SSPC's detection. The surface plot in Figure 3.22 shows the results of the parametric sweep.

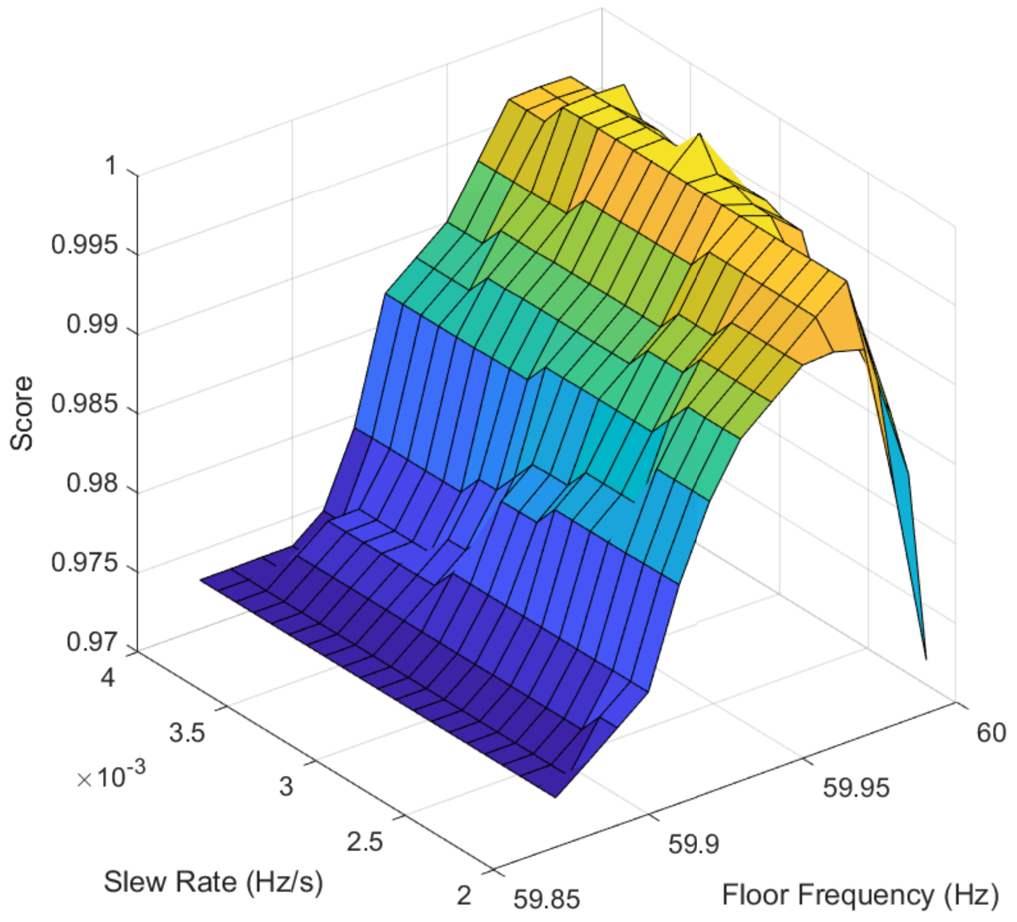


Figure 3.22: Scoring a Parametric Sweep of Sensitivity Settings for Event Detection

The sweep reveals a best match with a floor frequency of 59.98 Hz (equivalent to a deviation absolute value of 0.02 Hz), and a slew rate of 0.0031 $\frac{Hz}{s}$. Using these parameters, the DERAS detection algorithm output is a 99.98% match with the SSPC. The 0.02% difference is just due to some events being detected a second earlier or later.

The SSPC algorithm only detects downward frequency deviations. The algorithm written for DERAS checks for both upward and downward deviations by reflecting the floor frequency and minimum slew rate across the base frequency of 60 Hz. However, adding responses for both types of event leads to the possibility of both happening within the time window of a single response. A real example of this happening is shown in Figure 3.23.

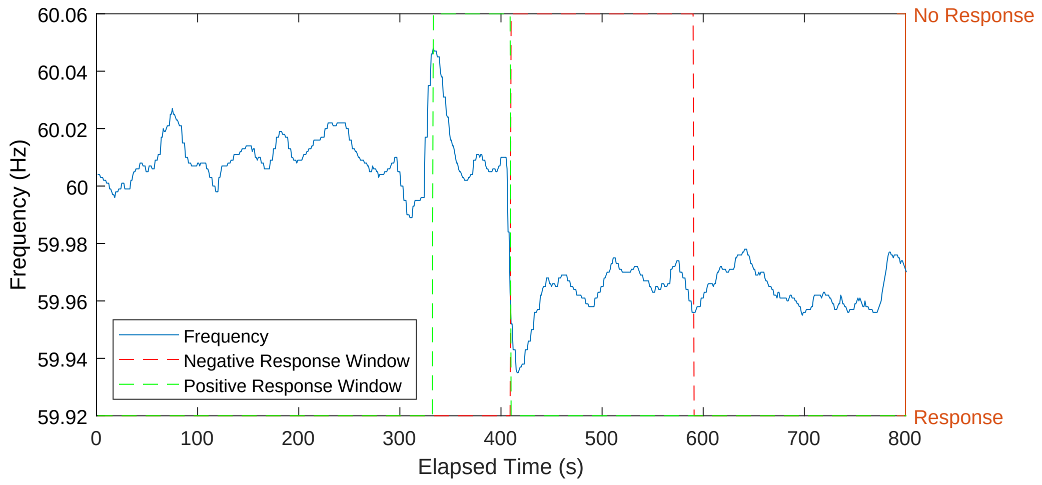


Figure 3.23: Adjacent Positive and Negative Frequency Disturbances

In this case, a positive disturbance happens, but then a negative disturbance occurs shortly afterward, while the system would normally still be responding to the first event. A continued response to the positive disturbance would actually exacerbate the situation by continuing to pull down the frequency with added load. Because downward events are more common, and PGE's current protocol is to respond to them, DERAS was programmed to always prioritize downward responses. In a situation such as that shown in Figure 3.23, DERAS would immediately end its response to the first event and only respond to the second.

3.5.3.2 Event Response

This work treats water heaters as upward dispatchable resources, which are normally idle but can be turned on. The response algorithm for positive frequency disturbances is more detailed than the response for negative disturbances, because the aggregator's capacity to decrease load is much lower. Because frequency response is a high-value and time-critical service, the aggregator initially responds to both kinds of events with its full capacity. All possible devices will be turned off or on as soon as possible, depending on the type of event. The positive event response algorithm was created based on the SSPC's response shown in Figure 2.11. After three minutes of all devices in the aggregate importing as much power as they can, the aggregator's requested power ramps down to zero over the course of another three minutes. For negative event responses, there is no ramp. Because the capacity for downward dispatch is less, the aggregator simply tells all devices not to turn on for a full six minutes.

4.1 Digital Twin Validation

The first goal is to validate that DERAS's digital twins do a sufficient job of representing the devices they are assigned. A side-by-side comparison of data logged locally by a real EWH and the digital twin created by DERAS for that device is shown in Figure 4.1.

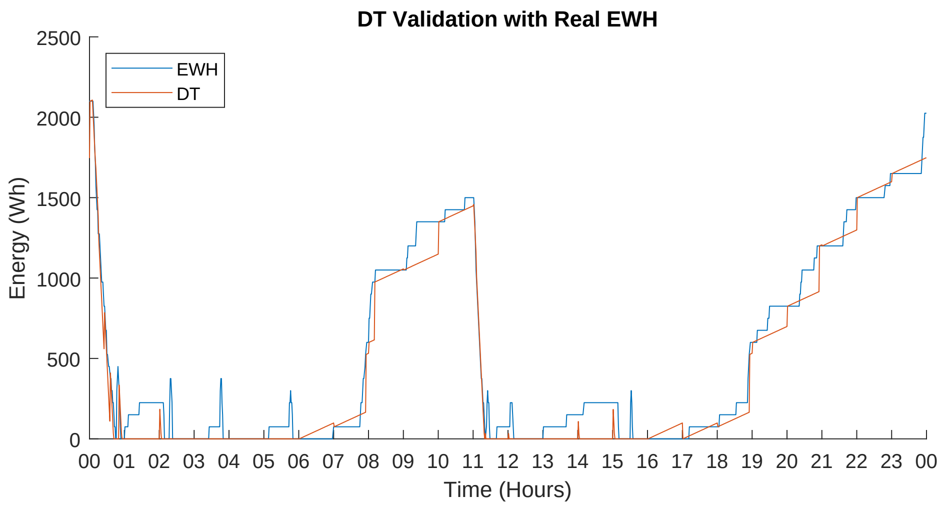


Figure 4.1: Comparison of Real EWH Data and Digital Twin

The digital twin tracks the EWH fairly accurately. When the import energy changes quickly, such as following the import commands at midnight and 11 AM, or the large usage events around 8 AM, the EWH and its digital twin are almost perfectly overlapping. There is some difference during the hours of importing, because the EWH energy never changes by more than 400 Wh, so the updates are less frequent. However, because there is very little of interest happening during these low-usage hours, and the aggregator already knows that the devices are importing whenever possible, the digital twin accuracy is not very important here. The results of the same test, but this time comparing an emulated device to its digital twin, is shown in 4.2. Note that the physical and emulated device DT validation tests were done with different randomized usage schedules, which is why Figures 4.1 and 4.2 don't match.

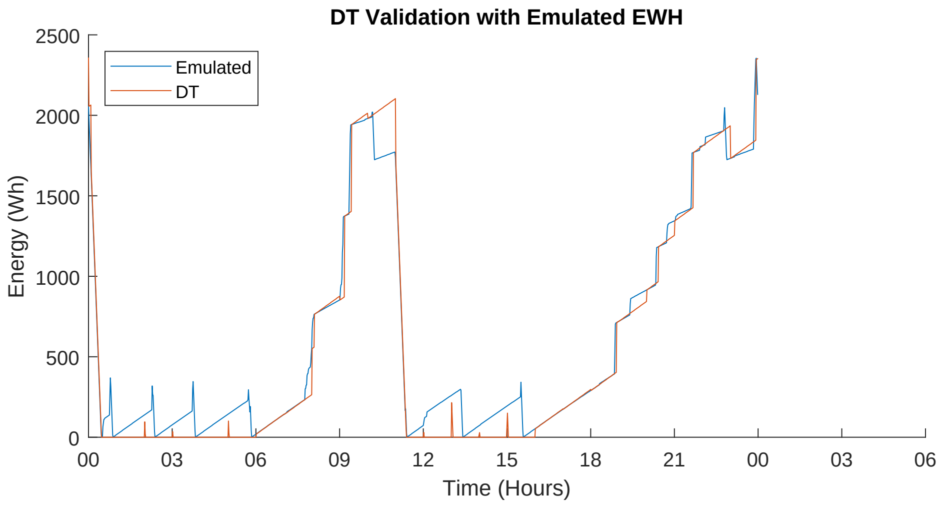


Figure 4.2: Comparison of Emulated EWH Data and Digital Twin

As with the real EWH, the digital twin does a sufficient job of representing the emulated device. Once again the accuracy decreases during the less eventful times, but is also less important here because the total variation in energy remains very small. A group of digital twins representing a real EWH and six different emulated devices is shown in Figure 4.3. Note that the real EWH is practically indistinguishable from the emulated devices.

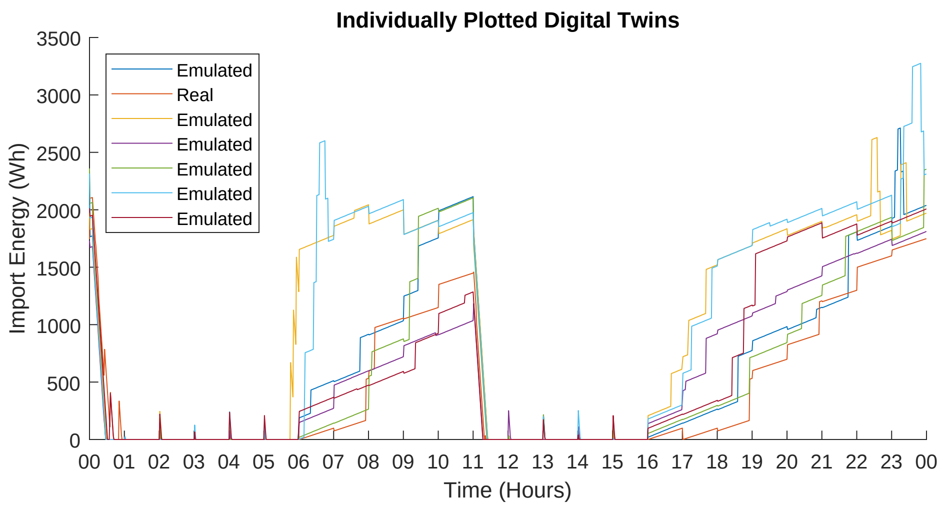


Figure 4.3: Digital Twins Representing Both Real and Emulated Devices

The accuracy of the digital twins could be improved by increasing the update frequency. However, the accuracy shown here is good enough that it would actually be more sensible to decrease the update frequency in order to decrease the amount of network traffic even further. The most critical times for digital twin accuracy are periods of high usage and large changes in import energy, and the results shown here are more than adequate.

4.2 Peak Shifting Service

To demonstrate the utility of DERAS for peak shifting using EWHs, an aggregate of 100 EWH emulators was first run with no CTA-2045 commands to show the power and energy trends with no outside influence. The baseline power consumption is shown in Figure 4.4.

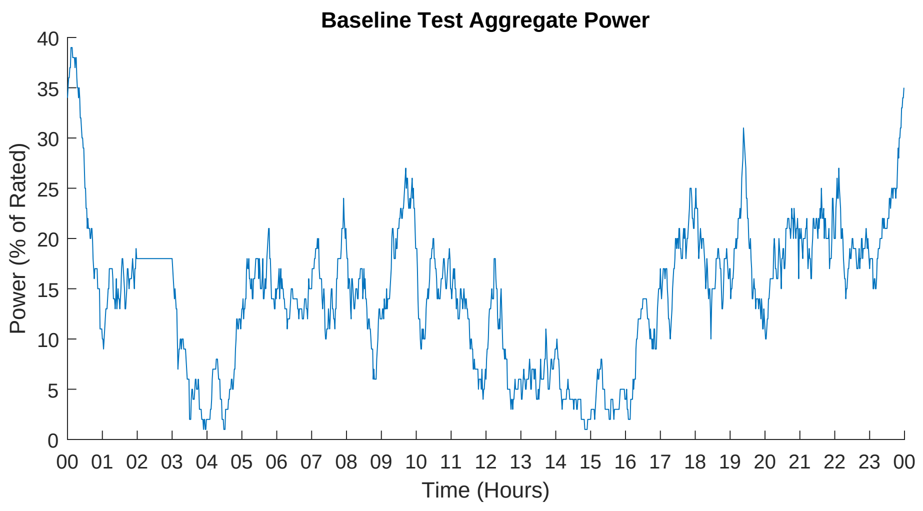


Figure 4.4: Power Consumption of Aggregate without any DERAS Commands

The typical residential load curve is visible in the plot, with very high consumption in the morning and evening, and very low consumption in the afternoon and at night. The greatest consumption is around midnight, when nearly 40% of the aggregate's devices are on. This spike at midnight is anomalous and most likely due to the randomization of the usage schedules. The import energy for the same aggregate is shown in Figure 4.5.

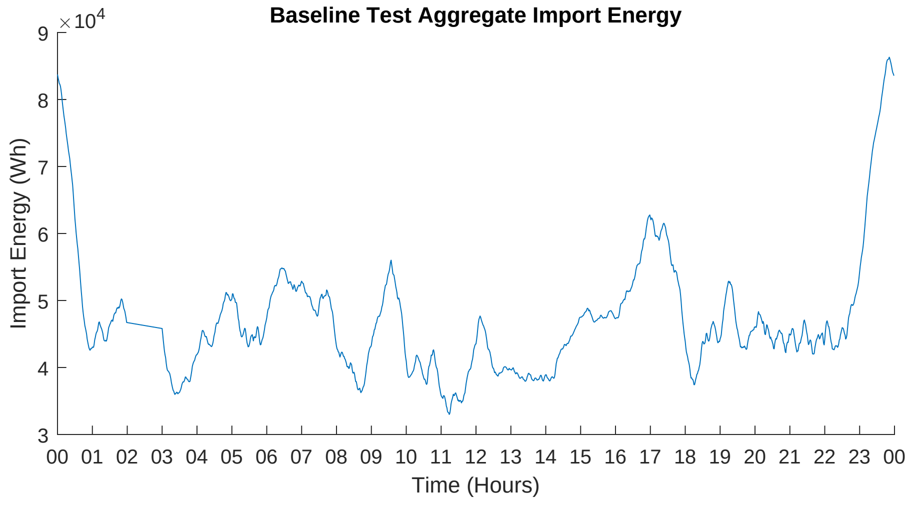


Figure 4.5: Import Energy of Aggregate without any DERAS Commands

Aside from a peak during the biggest usage hour leading up to midnight, the baseline import energy does not follow any particular pattern, but generally stays within a band between 40 and 50 kWh. This meets expectations, because the devices are all staying within the same regulation range for the entire test. The baseline regulation range is from 0 to 900 Wh, which for 100 devices aggregates to a range of 0 to 90,000 Wh. If the aggregate import power averages to around 45,000 Wh, this puts it right in the middle of the regulation range.

In the next test, DERAS controlled 100 emulated EWHs over 24 hours using the generic dispatch schedule described in Table 3.2 with its peak shifting service. The resulting aggregate power plot is shown in Figure 4.6. Load up times are shaded red, and shed times are shaded blue.

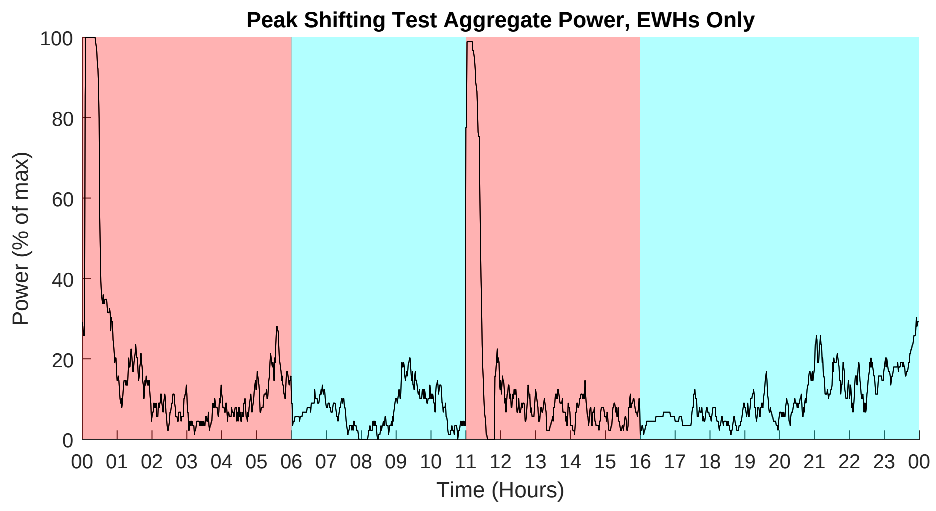


Figure 4.6: Aggregate Power Consumption of 100 EWHs with Peak-Shifting Schedule

At midnight and 11AM, when the two load up periods begin, nearly 100% of the devices turn on. This peak does not last long because all devices are fully heated within an hour and turn back off. Such a concentration of power consumption might be undesirable, but this could be fixed by limiting the import power request sent by DERAS so that fewer devices turn on at once. Outlying usage events during these load up periods cause some temporary spikes, such as shortly before 6AM, because the devices turn on in response to any large decrease in import energy. More importantly, there is a clear reduction in power consumption during the shed hours. Particularly at the start of the peak usage hours, from 6AM to 9AM and from 4PM to 9PM, there is a drastic reduction in power consumption because the devices begin with a large amount of energy and widen their regulation ranges. The effectiveness of the evening shed period does gradually expire as it gets closer to midnight, as the aggregate's amount of stored energy becomes exhausted and more devices turn back on. The maximum effective length of a shed period therefore appears to be around 5 hours, although this will always depend on usage patterns. The effects of the aggregator commands are also clearly visible in the aggregate energy plot shown in Figure 4.7.

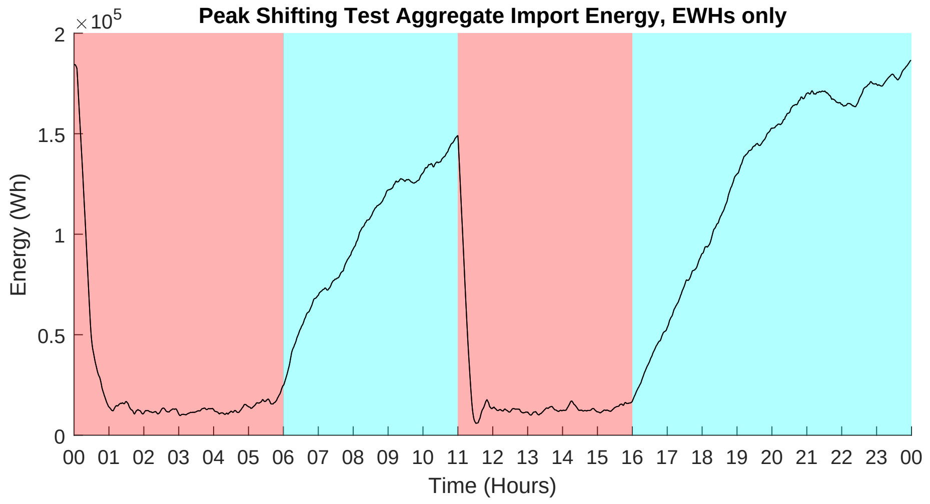


Figure 4.7: Aggregate Import Energy of 100 EWHs with Peak-Shifting Schedule

The shed and load up periods are more distinct in the energy plot. The aggregate successfully maintains its stored energy at near the limit for load up periods, and steadily releases that energy during shed periods. The upper threshold of the regulation for EWHs receiving a *shed* command was determined to be 2100 Wh, so it can be seen that by midnight that nearly every device has met or exceeded that limit. This means that the water heaters' energy storage utility has been maximized.

The peak shifting test was repeated with a mix of 69 EWH and 31 HPWH emulators. The power plot is shown in Figure 4.8.

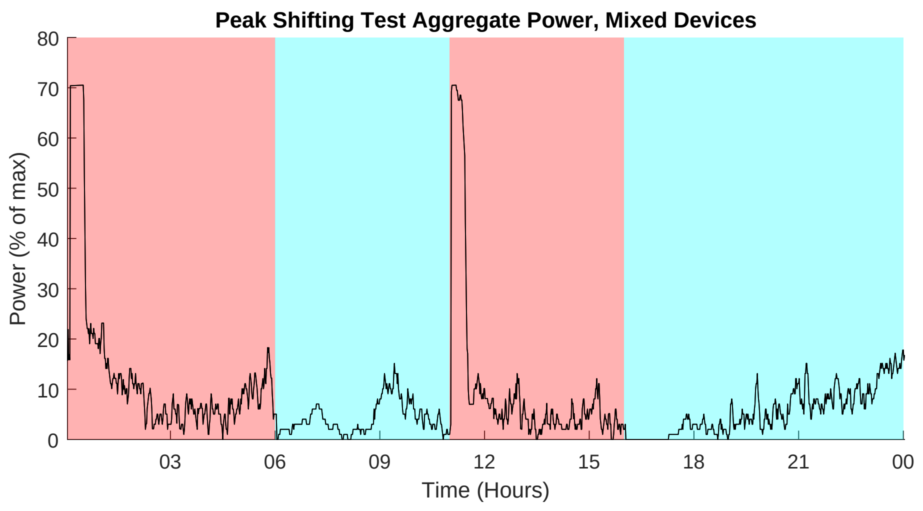


Figure 4.8: Aggregate Power Consumption of Mixed Devices with Peak-Shifting Schedule

The peak shifting test with mixed devices was an even greater success. The reductions in power consumption at the beginning of the shed events at 6AM and 4PM are clearly visible, and the peak hours seen in the baseline power plot are almost completely eliminated. The spikes in power consumption at the beginning of the load up periods at midnight and 11AM are also significantly lower. This is because the max power consumption was calculated considering that HPWHs can use both their elements and heat pumps simultaneously. However, this only occurs when the device has a very high import energy, so most of the time only the heat pump is used and the devices operate at well below their full rated power. Although HPWHs offer DERAS less controllable resource per device, they also act as a natural form of peak shifting because due to their lower power and heating rates there is a greater separation between the time of use and the time of power consumption. The energy plot for this test is shown in Figure 4.9.

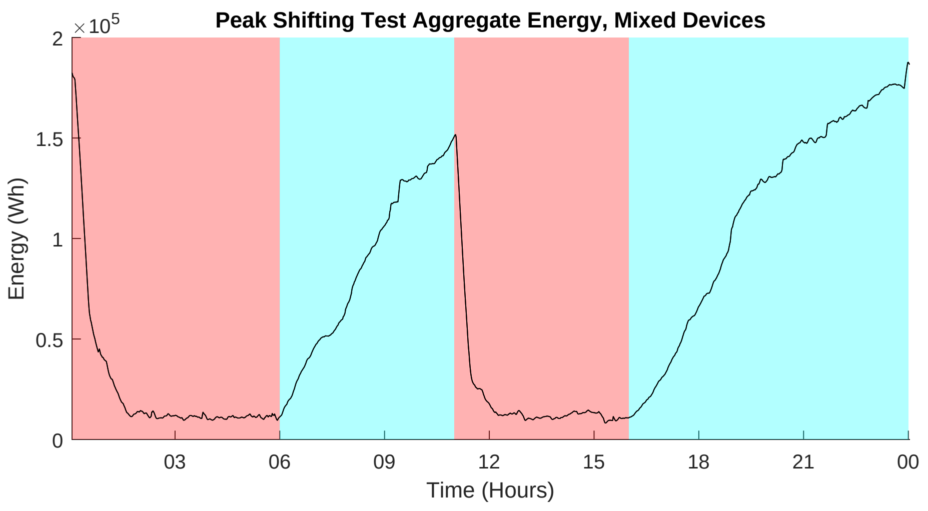


Figure 4.9: Aggregate Import Energy of Mixed Devices with Peak-Shifting Schedule

The energy plot for the mixed device peak shifting test is almost identical to the plot from the test with only EWHs. This makes sense, since usage losses are fairly similar for both device types.

To mitigate the large spikes in power consumption at the beginning of import periods, a limit was placed on the amount of power DERAS could request at once, effectively staggering the devices by only sending the *load up* command to at most a third of them at a time. The peak shifting test was then run a final time with the same mixture of 31 HPWHs and 69 EWHs. The resulting power and energy plots are shown in Figures 4.10 and 4.11.

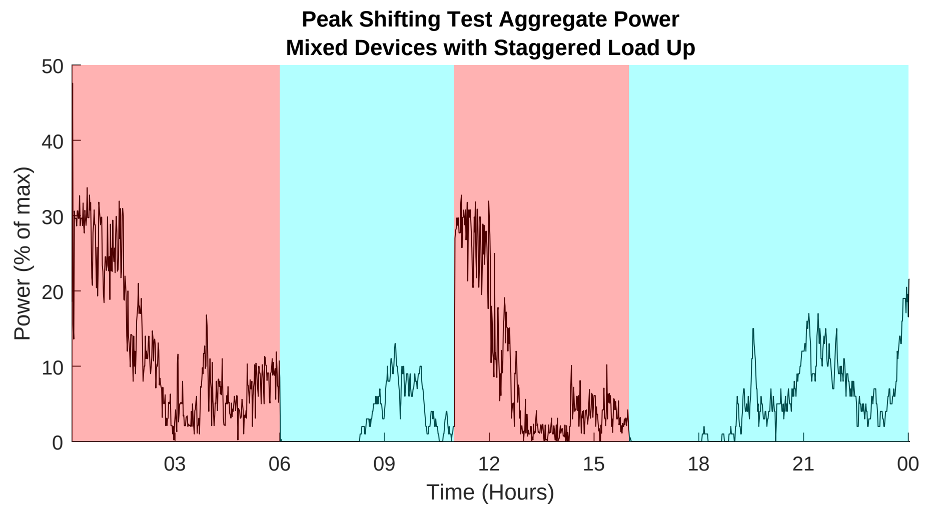


Figure 4.10: Aggregate Power of Mixed Devices with Staggered Peak-Shifting Schedule

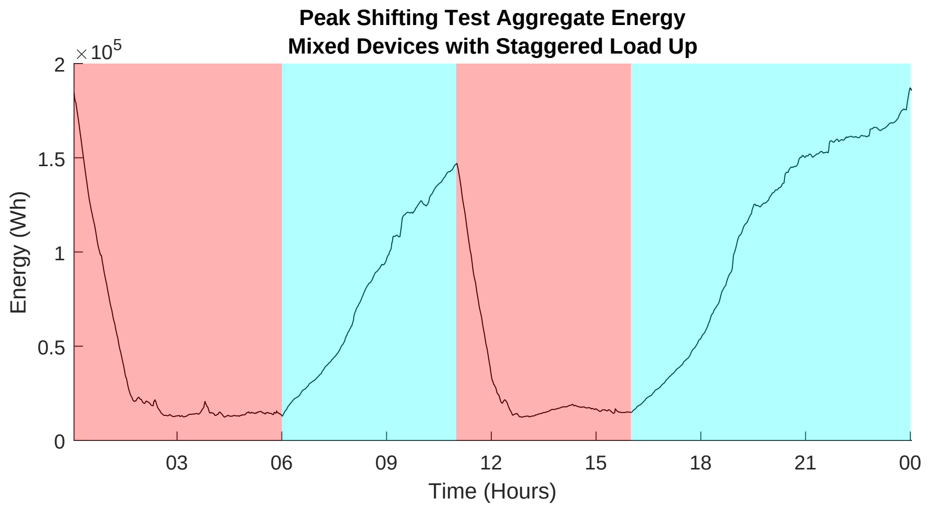


Figure 4.11: Aggregate Import Energy of Mixed Devices with Staggered Peak-Shifting Schedule

This step lowered the power consumption spikes from 70% to about 30% without negatively impacting the aggregator's peak shifting capacity. Roughly the same amount of import energy is available at the start of each loading period. DERAS automatically prioritizes devices with a larger amount of import energy. The ramp down is choppy because devices are constantly heating up, turning off, and then being replaced. The energy plot is nearly identical to those from the previous peak shifting experiments. Figures 4.10 and 4.11 satisfactorily demonstrate DERAS's capacity for peak shifting using an aggregate of water heaters, therefore meeting one of this work's major objectives.

4.3 Frequency Response

To test DERAS's capacity to detect and respond to positive frequency disturbances, frequency data containing an event were applied to an aggregate of 100 emulated EWHs. The response to this event is shown in Figure 4.12.

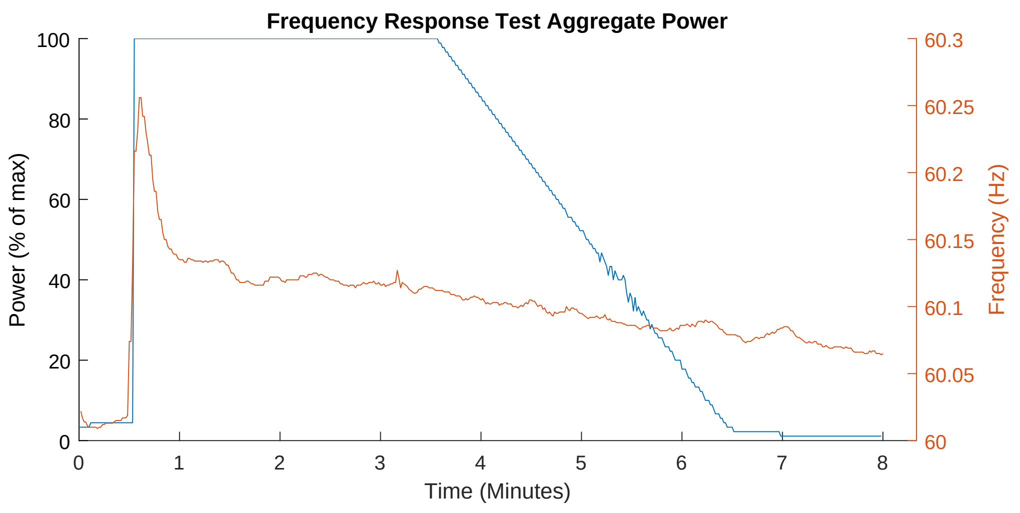


Figure 4.12: Aggregate Frequency Response to Upward Disturbance

The event is detected perfectly, and the response is almost instantaneous. Only 4 of the devices were on before the event, but all 100 were on within seconds of the event beginning.

Given the fast load up response time for EWHs shown in Figure 3.10, this test indicates that a very large number of EWHs could be dispatched in the first few seconds of a frequency disturbance, even considering the delay between the DERAS command and the physical response. All 100 devices are able to remain on for the desired 3 minutes, and then ramp down almost perfectly back to pre-disturbance conditions at the end of the following 3 minutes. The response demonstrated here is a successful recreation of the SSPC's frequency response shown in Figure 2.11, modified upward frequency disturbances. The results of this test met all expectations, and suffice to validate the untapped potential of aggregated EWHs as a resource for frequency response. Note that the frequency data shown here is an example from a previously recorded disturbance, and the devices in this work had nothing to do with the actual recovery. This work does not suggest that water heaters alone are sufficient for frequency response. The Western Interconnection is a massive system and no single asset is capable of such an impact on the grid frequency. This work only proposes that aggregated water heaters can contribute to the system's net resilience to upward frequency disturbances. The effect of this response on the aggregate's available import energy is shown in Figure 4.13.

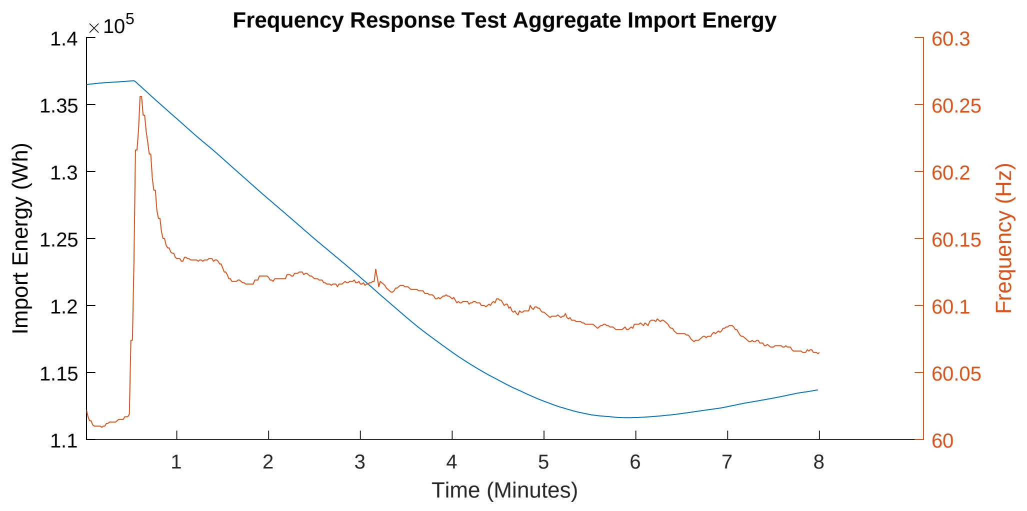


Figure 4.13: Aggregate Import Energy during Frequency Response

Over the course of the response, the aggregate's import energy was decreased roughly from 136 kWh to 112 kWh, or by about 24 kWh. An aggregate of 100 EWHs will almost always have this much available import energy unless it is already loading up. Because the power consumption is also proportional to the number of devices in the aggregate, it is safe to say that an EWH aggregate of any size can be expected to respond to upward frequency disturbances with its full capacity most of the time. The success of the frequency response test completes the second major objective of this research.

The frequency response experiment was repeated with a mixture of 31 HPWHs and 69 EWHs to demonstrate that HPWHs are a less useful asset than EWHs for frequency response due to their lower power consumption and longer response times. The results of this test are shown in Figure 4.14.

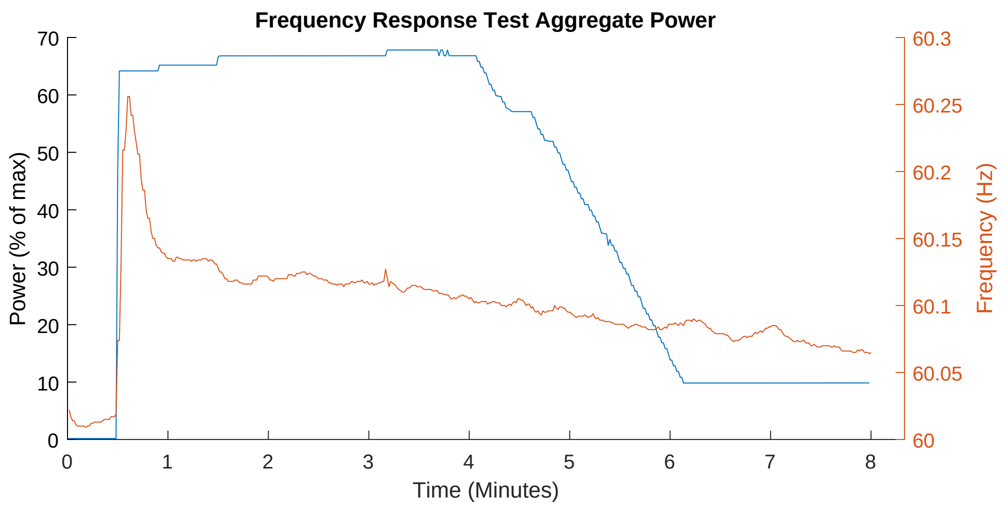


Figure 4.14: Aggregate Power during Frequency Response with Mixed Devices

Replacing some of the EWHs with HPWHs has clearly visible effects on the frequency response. The heat pump emulator was programmed with a one-minute turn-on delay based on the observations made in Figure 3.11. Therefore, the initial spike in power at the beginning of the disturbance consists only of the EWHs and possibly the resistive elements of some HPWHs which had very high import energy (HPWHs only respond to *load up* commands with their resistive elements if they have more than 1425 Wh of import energy). 2 heat pumps are on before the event begins, and 64 elements are energized in the initial response. This means that not all of the 69 EWHs are able to turn on, which is most likely to them already being too warm to respond to the *load up* command. The heat pumps turn on one minute later, and slowly ramp up their power consumption over the course of the response. However, because the heat pumps consume less than 10% of a resistive heating element's power, even when all 31 heat pumps are on this only makes a small difference to the total response power. The ramp down is also less even because the different devices are consuming different amounts of energy. Some usage events occurred during the response, so at the end 9 devices are unable to turn off (this is anomalous and could also have happened in the test with only EWHs). In general, replacing EWHs with HPWHs reduces the total power available for frequency response as well as the precision of DERAS's control.

5 Discussion

5.1 Summary of Results

The peak shifting and frequency response capabilities of aggregated water heaters demonstrated in this research met all of the initial goals and expectations. Satisfactory emulators were created for CTA-2045-equipped EWHs and HPWHs, and groups of 100 emulated devices were controlled using the PSU-built DERAS aggregation system. The tests were conducted both exclusively with EWHs and with the expected mixture of EWHs and HPWHs 20 years in the future.

The peak shifting tests demonstrated that it is possible to displace significant power consumption during periods of high usage, although the duration of this displacement is typically limited to 4 or 5 hours. In the first experiments, the control signal to turn on at the end of a displacement period was sent to all devices simultaneously, resulting in a sudden, severe spike in consumption. However, this was corrected by adding a limit to the percentage of devices in the network that received the command at once. DERAS prioritizes devices with more available energy, so the coldest heaters come on first.

The frequency response tests demonstrated both that the aggregator's frequency disturbance detection algorithm is capable of quickly triggering a response to disturbances, and that EWHs have great potential to respond to upward frequency disturbances due to their very fast activation times, ramp rates, and high power consumption. An aggregate of EWHs was able to provide a frequency response nearly identical to examples from the SSPC. HPWHs were shown to be less useful for frequency response because of their low power consumption and the necessary turn-on delay of heat pumps. Frequency response is a high-value ancillary service, and dedicating water heater assets to responding to upward frequency disturbances could be extremely beneficial to utilities while minimally impacting customers.

5.2 Potential Sources of Error

The experimental design had several flaws. One shortcoming is the small number of devices used in the aggregate tests. Aggregate sizes were limited to 100 devices because the Alljoyn network would otherwise become unstable and begin losing devices. Alljoyn has already been absorbed by a larger project and lost most of its support, so the only solution for this problem is to switch to a new, more robust framework.

Another weakness in the experimental design is the small group of profiles used to simulate stochastic customer usage. Only one original profile was made for each household size, and then the emulated devices created unique profiles by randomizing the original ones. This led to a lack of overall variety in the aggregate usage patterns as well as some anomalous results such as the large peak at midnight seen clearly in Figure 4.4. Diversifying the group of original usage profiles would be a fairly simple improvement to the experimental design.

Three steps could be taken to improve the emulator accuracy. First, the device response times were based off of a single observation. These response times are not uniform, and a more representative delay time could be determined by making several observations and then observing any patterns and perhaps taking the average value. However, the key point of this observation was only that EWHs can turn on very quickly, while heat pumps take so long that they are not useful for frequency response, so this is not a critical error.

Second, the heat pump has some limit on how many cycles it can run within a given amount of time, which caused a discrepancy between the physical and emulated devices as shown in Figure 3.21. The number of cycles and the length of the time window are unknown. Determining these values and including them in the HPWH emulator would improve its accuracy.

Finally, the HPWH has several different operation modes that can be chosen by the customer using buttons on the front panel. These modes have significant effects on the behavior of the device, but for this research only the 'hybrid' mode was observed and replicated. A more robust emulator would include functionality to switch between modes.

5.3 Suggestions for Manufacturers

There are five ways that water heater manufacturers could greatly increase the effectiveness of their devices for utility services that also would have made this research much easier.

- First, devices such as the HPWH, which has multiple modes of operation, should report what mode they are using through the CTA-2045 interface.

- Second, HPWHs should allow separate control of their resistive elements and heat pumps. If DERAS was able to selectively tell HPWHs to turn on their resistive elements rather than their heat pumps, they would be just as useful as EWHs for frequency response.

- Third, devices should report their current power consumption more accurately. The HPWH always reports that the heat pump is consuming 800 W any time that it is on. However, the power consumption actually changes based on the temperatures of the tank and the surrounding air, and is typically closer to 400 W.

- Fourth, devices should be faster to report changes in import energy and power. The delay between changes in actual and reported states can exceed one minute, which lowers the precision of the aggregator's calculations and controls.

- Finally, devices should allow the utility to directly control their temperature setpoints through the CTA-2045 interface. Manufacturers can still allow limits for customer comfort and safety, but allowing direct control of the thermostat within those limits would be preferable to using the commands such as *shed*, *load up*, etc. CTA-2045 already provides message structures for directly adjusting the temperature setpoint, but they are not supported by the devices used in this research.

6 Conclusion

This work successfully demonstrated the utility of DERAS for peak shifting and frequency response using aggregates of water heaters. These services have high monetary value to the current grid, and will increase in value as renewable energy sources contribute more of the total generation. Turning large numbers of water heaters off at peak hours can help even out the balance of supply and demand and reduce the ramping requirements of the worsening duck curve on the rest of the system. Turning water heaters on in response to upward frequency disturbances can help replace the mechanical inertia of traditional generators as these machines are replaced with inverter-based generation.

This research leaves several opportunities for future work. One possible development is the use of machine learning for frequency disturbance detection. As demonstrated by both the SSPC detection algorithm and the algorithm developed for DERAS, accurately detecting frequency disturbances without responding to false positives is very difficult. Because a large corpus of frequency data with many identified disturbances already exists, this is an excellent application for guided machine learning.

Increasing the number of devices that can be included in the aggregator's network would improve the reliability of experiments and is also a necessary step for expanding this work into a real-world application with thousands of customers. Therefore, an immediate and pressing task is the transfer of the aggregator and device controllers to a new communication framework. The creator of DERAS currently plans to switch to the IEEE 2030.5 standard.

For further experiments using emulators, more reliable results will be achieved with a more robust probabilistic model for hot water usage. However, at some point this project will need to expand to real-world applications. This will require the DCS to be refined into a sturdy and packaged device that is easy to install in customer homes. The experiments will need to switch from emulated to real devices for validation. The quality of the results and the experimental costs will both increase with the number of included households, but this is the only pathway for this project to eventually become realized.

Bibliography

[1] NERC Interconnections Monthly Frequency Events. Technical report, Consortium for Electric Reliability Technology Solutions, January 2015.

[2] U.S. Energy Information Administration. What is u.s. electricity generation by energy source? Web, March 2019. https://www.eia.gov/tools/faqs/faq.php?id=427t=3.

[3] Annie Clarke. Electric Water Heater Modeling for CTA-2045 Demand Response Testing. Master's thesis, Portland State University, May 2018.

[4] B. Hendron and J. Burch and G. Barker. Tool for Generating Realistic Residential Hot Water Event Schedules. Technical report, National Renewable Energy Laboratory, August 2010.

[5] Bonneville Power Administration. Description of Electric Energy Use in Single-Family Residences in the Pacific Northwest. Technical report.

[6] Bonneville Power Administration. CTA-2045 Water Heater Demonstration Report. Technical report, November 2018.

[7] United States Census Bureau. 2013-2017 american community survey 5-year estimates: Dp04 - selected housing characteristics. Web, 2018. https://factfinder.census.gov/faces/tableservices/jsf/pages/productview.xhtml?src=bkmk.

[8] C. Shapiro and S. Puttagunta. Field Performance of Heat Pump Water Heaters in the Northeast. Technical report, U.S. Department of Energy, February 2016.

[9] California ISO. Frequency Response - Issue Paper. Technical report, August 2015.

[10] Consumer Electronics Association. ANSI/CEA Standard Modular Communications Interface for Energy Management: ANSI/CEA-2045. Technical report, February 2013.

[11] R. Diao, S. Lu, M. Elizondo, E. Mayhorn, Y. Zhang, and N. Samaan. Electric water heater modeling and control strategies for demand response. In *2012 IEEE Power and Energy Society General Meeting*, pages 1–8, July 2012.

[12] Electric Power Research Insitute. Demand Response-Ready Domestic Water Heater Specification: Preliminary Requirements for CEA-2045 Field Demonstration. Technical report, December 2014.

[13] Electric Power Research Insitute. Introduction to the ANSI/CEA-2045 Standard. Technical report, May 2014.

[14] Electric Power Research Insitute. CTA-2045 Simulator User's Manual. Technical report, May 2017.

[15] Electric Power Research Insitute. Performance Test Results: CTA-2045 Water Heater. Technical report, November 2017.

[16] Eurostat. Share of renewable energy in the eu up to 17.5% in 2017. Web, February 2019. https://ec.europa.eu/eurostat/documents/2995521/9571695/8-12022019-AP-EN.pdf/b7d237c1-ccea-4adc-a0ba-45e13602b428.

[17] Galen L. Barbose. U.S. Renewables Portfolio Standards: 2017 Annual Status Report. Technical report, July 2017. https://emp.lbl.gov/publications/us-renewables-portfolio-standards-0.

[18] J.J. Guo, J.Y. Wu, and R.Z. Wang. A new approach to energy consumption prediction of domestic heat pump water heater based on grey system theory. *Energy and Buildings*, 43(6):1273 – 1279, 2011.

[19] Elta Koliou, Cherrelle Eid, José Pablo Chaves-Ávila, and Rudi A. Hakvoort. Demand response in liberalized electricity markets: Analysis of aggregated load participation in the german balancing mechanism. *Energy*, 71(Supplement C):245 – 254, 2014.

[20] D. Lew, M. Asano, J. Boemer, C. Ching, U. Focken, R. Hydzik, M. Lange, and A. Motley. The power of small: The effects of distributed energy resources on system reliability. *IEEE Power & Energy Magazine*, pages 50 – 60, October 2017.

[21] National Renewable Energy Laboratory. Wind and Solar Energy Curtailment: Experience and Practices in the United States. Technical report, March 2014. https://www.nrel.gov/docs/fy14osti/60983.pdf.

[22] North American Electric Reliability Corporation. Balancing and Frequency Control. Technical report, January 2011.

[23] North American Electric Reliability Corporation. Frequency Response Initiative Report: the Reliability Role of Frequency Response. Technical report, October 2012.

[24] North American Electric Reliability Corporation. Frequency Response Standard Background Document. Technical report, November 2012.

[25] North American Electric Reliability Corporation. 2018 Long-Term Reliability Assessment. Technical report, December 2018.

[26] Oak Ridge National Laboratory. Assessment of Industrial Load for Demand Response across U.S. Regions of the Western Interconnect. Technical report, September 2013.

[27] Pacific Northwest National Laboratory. The Salem Smart Power Center: An Assessment of Battery Performance and Economic Potential. Technical report, September 2017.

[28] Tylor Slay. Adoption of an Internet of Things Framework for Distributed Energy Resourc Coordination and Control. Master's thesis, Portland State University, May 2018.

[29] U.S. Energy Information Administration. Electric Power Annual 2017. Technical report, December 2018.

Appendix A: Code

A.1 DERAS

A.1.1 main.cpp

```
/******************************************************************************
 *    Copyright (c) Open Connectivity Foundation (OCF), AllJoyn Open Source
 *    Project (AJOSP) Contributors and others.
 *
 *    SPDX-License-Identifier: Apache-2.0
 *
 *    All rights reserved. This program and the accompanying materials are
 *    made available under the terms of the Apache License, Version 2.0
 *    which accompanies this distribution, and is available at
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 *    Copyright (c) Open Connectivity Foundation and Contributors to AllSeen
 *    Alliance. All rights reserved.
 *
 *    Copyright (c) V2 Systems, LLC.  All rights reserved.
 *
 *    Permission to use, copy, modify, and/or distribute this software for
 *    any purpose with or without fee is hereby granted, provided that the
 *    above copyright notice and this permission notice appear in all
 *    copies.
 *
 *    THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
 *    WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED
 *    WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE
 *    AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL
 *    DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
 *    PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
 *    TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
 *    PERFORMANCE OF THIS SOFTWARE.
******************************************************************************/

// INCLUDE
#include <iostream>     // cout, cin
#include <thread>       // thread, join
#include <chrono>       // now, duration
#include <map>
#include <string>
#include <vector>

#include "include/tsu.h"
#include "include/aj_utility.h"
#include "include/Aggregator.h"
#include "include/CommandLineInterface.h"
#include "include/ClientListener.h"
#include "include/SmartGridDevice.h"
#include "include/Operator.h"

// NAMESPACES
using namespace std;
using namespace ajn;

// GLOBALS
bool done = false;      // signal program to stop

// Program Help
// - command line interface arguments during run, [] items have default values
static void ProgramHelp (const string& name) {
    cout << "\n[Usage] > " << name << " -c <file path> [-o <y/n>] -h\n"
        "\t[] means it has a default value\n"
        "\t -h \t help\n"
        "\t -c \t configuration filename"
        "\t -o \t enable operator"  << endl;
}  // end Program Help

// Argument Parser
// - method to parse program initial parameters
static std::map <std::string, std::string> ArgumentParser (int argc,
                                                           char** argv) {
    string name = argv[0];
    // parse tokens
    map <string, string> parameters;
    string token, argument;

    for (int i = 1; i < argc; i = i+2){
        token = argv[i];

        // check to see if the is an argument for the program control token
        if (argc <= i+1) {
            cout << "[ERROR] : Invalid program argument: " << token << endl;
            ProgramHelp(name);
            exit(EXIT_FAILURE);
        } else {
            argument = argv[i+1];
        }


        if ((token == "-h")) {
            ProgramHelp(name);
            exit(EXIT_FAILURE);
        } else if ((token == "-c")) {
            parameters["config"] = argument;
        } else {
            cout << "[ERROR] : Invalid parameter: " << token << endl;
            ProgramHelp(name);
            exit(EXIT_FAILURE);
        }
    }
    return parameters;
}  // end Argument Parser

// THREADS
// -------

// Resource Loop
// - this loop runs the aggregator control loop at the desired frequency
// - it subtracks processing time of the Loop () function to make the frequency
// - more consistant
void AggregatorLoop (unsigned int sleep, Aggregator* vpp_ptr) {
    unsigned int time_remaining, time_past;
    unsigned int time_wait = sleep;
    auto time_start = chrono::high_resolution_clock::now();
    auto time_end = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> time_elapsed;

    while (!done) {
        time_start = chrono::high_resolution_clock::now();
            // time since last control call;
            time_elapsed = time_start - time_end;
            time_past = time_elapsed.count();
            vpp_ptr->Loop(time_past);
        time_end = chrono::high_resolution_clock::now();
        time_elapsed = time_end - time_start;

        // determine sleep duration after deducting process time
        if (time_wait - time_elapsed.count() > 0) {
            time_remaining = time_wait - time_elapsed.count();
            this_thread::sleep_for (chrono::milliseconds (time_remaining));
        }
    }
}  // end Aggregator Loop

// Operator Loop
// - this loop runs the resource control loop at the desired frequency
// - it subtracks processing time of the Loop () function to make the frequency
// - more consistant
void OperatorLoop (unsigned int sleep, Operator* oper_ptr) {
    unsigned int time_remaining;
    unsigned int time_wait = sleep;
    auto time_start = chrono::high_resolution_clock::now();
    auto time_end = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> time_elapsed;

    while (!done) {
        time_start = chrono::high_resolution_clock::now();
        oper_ptr->Loop();
        time_end = chrono::high_resolution_clock::now();
        time_elapsed = time_end - time_start;

        // determine sleep duration after deducting process time
        if (time_wait - time_elapsed.count() > 0) {
            time_remaining = time_wait - time_elapsed.count();
            this_thread::sleep_for (chrono::milliseconds (time_remaining));
        }
    }
}  // end Resource Loop

// Smart Grid Device Loop
// - this loop runs the resource control loop at the desired frequency
// - it subtracks processing time of the Loop () function to make the frequency
// - more consistant
void SmartGridDeviceLoop (unsigned int sleep, SmartGridDevice* sgd_ptr) {
    unsigned int time_remaining;
    unsigned int time_wait = sleep;
    auto time_start = chrono::high_resolution_clock::now();
    auto time_end = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> time_elapsed;

    while (!done) {
        time_start = chrono::high_resolution_clock::now();
        sgd_ptr->Loop();
        time_end = chrono::high_resolution_clock::now();
        time_elapsed = time_end - time_start;

        // determine sleep duration after deducting process time
        if (time_wait - time_elapsed.count() > 0) {
            time_remaining = time_wait - time_elapsed.count();
            this_thread::sleep_for (chrono::milliseconds (time_remaining));
        }
    }
}  // end Resource Loop

// Main
// ----
int main (int argc, char** argv) {
cout
        << "\n*******************************************************"
        << "\n*** Distributed Energy Resource Aggregation System ***"
        << "\n*******************************************************\n";

    cout << "Initialization...\n";
    // if the config file is not passed to the program then exit
   if (argc == 1) {
        string name = argv[0];
        ProgramHelp(name);
        return EXIT_FAILURE;
    }
    map <string, string> arguments = ArgumentParser(argc, argv);

    // read config file for program configurations and object attributes
    tsu::config_map configs = tsu::MapConfigFile (arguments["config"]);

    cout << "\tCreating virtual power plant\n";
    // ~ reference Aggregator
    Aggregator* vpp_ptr = new Aggregator (configs);

    cout << "\tCreating Operator\n";
    Operator* oper_ptr = new Operator(configs["Operator"], vpp_ptr);

    cout << "\tCreating Command Line Interface\n";
    // ~ reference CommandLineInterface.h
    CommandLineInterface CLI(vpp_ptr, oper_ptr);

    cout << "\tCreating AllJoyn Message Bus\n";
    try {
        cout << "\t\tInitializing AllJoyn...\n";
        // Must be called before any AllJoyn functionality
        AllJoynInit();
    } catch (exception &e) {
        cout << "[ERROR]: " << e.what() << endl;
        return EXIT_FAILURE;
    }

    #ifdef ROUTER
        try {
            cout << "\t\tInitializing AllJoyn Router...\n";
            // Must be called before any AllJoyn routing functionality
            AllJoynRouterInit();
        } catch (exception &e) {
            cout << "[ERROR]: " << e.what() << endl;
            return EXIT_FAILURE;
        }
    #endif // ROUTER

    cout << "\tCreating AllJoyn Bus Attachment\n";
    // BusAttachment is the top-level object responsible for connecting to and
    // optionally managing a message bus.
    // ~ AllJoyn Docs
    string app = configs["AllJoyn"]["app"];
    bool allow_remote = true;
    BusAttachment* bus_ptr = new BusAttachment(app.c_str(), allow_remote);

    cout << "\tCreating AllJoyn About Data\n";
    // The AboutObj class is responsible for transmitting information about the
    // interfaces that are available for other applications to use.
    // ~ AllJoyn Docs
    AboutData about_data("en");
    AboutObj* about_ptr = new AboutObj(*bus_ptr);

    cout << "\tCreating AllJoyn Session Port\n";
    // inform users of session related events
    // ~ AllJoyn Docs
    aj_utility::SessionPortListener SPL;
    SessionPort port = stoul(configs["AllJoyn"]["port"]);

    cout << "\tSetting up AllJoyn Bus Attachment...\n";
    // ~ reference aj_utility.cpp
    QStatus status = aj_utility::SetupBusAttachment (configs,
                                                     port,
                                                     SPL,
                                                     bus_ptr,
                                                     &about_data);

    cout << "\tCreating AllJoyn Observer\n";
    // takes care of discovery, session management and ProxyBusObject creation
    // for bus objects.
    // ~ AllJoyn Docs
    const char* client_name = configs["AllJoyn"]["client_interface"].c_str();
    Observer* obs_ptr = new Observer(*bus_ptr, &client_name, 1);

    cout << "\tCreating AllJoyn Client Listener\n";
    // ~ reference ClientListener.cpp
    ClientListener* listner_ptr = new ClientListener(bus_ptr,
                                                      obs_ptr,
                                                      vpp_ptr,
                                                      client_name);
    obs_ptr->RegisterListener(*listner_ptr);

    cout << "\tCreating AllJoyn Smart Grid Device\n";
    // ~ reference SmartGridDevice.cpp
    const char* device_name = configs["AllJoyn"]["server_interface"].c_str();
    const char* path = configs["AllJoyn"]["path"].c_str();
    SmartGridDevice* sgd_ptr = new SmartGridDevice(bus_ptr,
                                                    vpp_ptr,
                                                    device_name,
                                                    path);

    cout << "\t\tRegistering AllJoyn Smart Grid Device\n";
    if (ER_OK != bus_ptr->RegisterBusObject(*sgd_ptr)){
        cout << "\t\t[ERROR]: Failed Registration!\n";
        return EXIT_FAILURE;
    }
    about_ptr->Announce(port, about_data);

    // most objects will have a dedicated thread, but not all
    cout << "\tSpawning threads...\n";
    thread VPP (AggregatorLoop, stoul(configs["Threads"]["sleep"]), vpp_ptr);

    thread OPER (
        OperatorLoop, stoul(configs["Threads"]["sleep"]), oper_ptr
    );

    thread SGD (
        SmartGridDeviceLoop, stoul(configs["Threads"]["sleep"]), sgd_ptr
    );

    // the CLI will control the program and can signal the program to stop
    cout << "Initialization complete...\n";
    CLI.Help ();
    string input;
    while (!done) {
        getline(cin, input);
        done = CLI.Control (input);
    }

    // when done = true, the program begins the shutdown process
    // TODO (TS): AllJoyn still leaves alot of errors when closing. The docs
    // - dont really explain the shutdown procedure for lots of alljoyn objects
    cout << "Closing program...\n";

    // First join all active threads to main thread
    cout << "\tJoining threads\n";
    VPP.join ();
    OPER.join ();
    SGD.join ();

    // Then delete all pointers that were created using "new" since they do not
    // automaticall deconstruct at the end of the program.
    cout << "\nDeleting pointers...\n";
    delete sgd_ptr;
    delete listner_ptr;
    delete obs_ptr;
    delete about_ptr;
    delete bus_ptr;
    delete oper_ptr;
    delete vpp_ptr;

    #ifdef ROUTER
        cout << "\tShutting down AllJoyn Router\n";
        status = AllJoynRouterShutdown ();
    #endif // ROUTER

    cout << "\tShutting down AllJoyn\n";
    status = AllJoynShutdown ();

    // return exit status
    return EXIT_SUCCESS;
} // end main
```

A.1.2 Aggregator.cpp

```
#include <iostream>
#include <algorithm>
#include <memory>
#include <vector>
#include <string>
```

```
#include <ctime>
#include <map>
#include <alljoyn/ProxyBusObject.h>
#include <alljoyn/Status.h>
#include "include/Aggregator.h"
#include "include/logger.h"

// constructor
// - simply initialize member properties
Aggregator::Aggregator (tsu::config_map &init) :
        config_(init),
        last_log_(0),
        log_inc_(stoul(init["Logger"]["increment"])),
        log_path_(init["Logger"]["path"]),
        total_export_energy_(0),
        total_export_power_(0),
        total_import_energy_(0),
        total_import_power_(0),
        export_watts_(0),
        import_watts_(0),
        price_(0),
        time_(0),
        temperature_(0) {
        // do nothing
}  // end constructor

Aggregator::~Aggregator () {
        // do nothing
}  // end destructor

// Set Targets
// - store the new target arguments and then filter the current resources
void Aggregator::SetTargets (const std::vector <std::string> &targets) {
        targets_ = targets;
        Aggregator::FilterResources ();
}  // end Set Targets

// Set Export Watts
// - set the dispatch watts for DER to export to the grid
void Aggregator::SetExportWatts (unsigned int power) {
        import_watts_ = 0;
        if (power > total_export_power_) {
                export_watts_ = total_export_power_;
        } else {
                export_watts_ = power;
        }
}  // end Set Export Watts

// Set Import Watts
// - set the dispatch watts for DER to import from the grid
void Aggregator::SetImportWatts (unsigned int power) {
        export_watts_ = 0;
        if (power > total_import_power_) {
                import_watts_ = total_import_power_;
        } else {
                import_watts_ = power;
        }
}  // end Set Import Watts

// Set Price
// - the tenths of a cent per kWh
void Aggregator::SetPrice (int price) {
        price_ = price;
}  // end Set Price

// Set Temperature
// - Set the local temperature in F
void Aggregator::SetTemperature (int temperature) {
        temperature_ = temperature;
}  // end Set Temperature

// Set Time
// - the UTC time as seconds from epoch
void Aggregator::SetTime () {
        unsigned int utc = time (nullptr);
        bool half_hour = utc % (60*30) == 0;
        // update time every hour
        if (half_hour) {
                time_ = utc;
        }
}  // end Set Time

// Get Total Export Energy
// - return the filtered resource Watt-hours available to export to the grid
unsigned int Aggregator::GetTotalExportEnergy () {
    return total_export_energy_;
}  // end Get Total Export Energy

// Get Total Export Power
// - return the filtered resource Watts available to export to the grid
unsigned int Aggregator::GetTotalExportPower () {
    return total_export_power_;
}  // end Get Total Export Power

// Get Total Import Energy
// - return the filtered resource Watt-hours available to import from the grid
```

```
unsigned int Aggregator::GetTotalImportEnergy () {
    return total_import_energy_;
}  // end Get Total Import Energy

// Get Total Import Power
// - return the filtered resource Watts available to import from the grid
unsigned int Aggregator::GetTotalImportPower () {
    return total_import_power_;
}  // end Get Total Import Power

// Get Price
// - return tenths of a cent per kWh for electricity
int Aggregator::GetPrice () {
        return price_;
}  // end Get Price

// Get Time
// - return the UTC time
unsigned int Aggregator::GetTime () {
        return time_;
}  // end Get Time

// Get Temperature
// - return the local temperature in F
int Aggregator::GetTemperature () {
        return temperature_;
}  // end Get Temperature

// Add Resource
// - This is used by the Client Listener class to add newly discovered DER.
// - it also passes the AllJoyn Proxy Bus Object that will be used to control
// - individual DER when desired
void Aggregator::AddResource (
        std::map <std::string, unsigned int>& init,
        ajn::ProxyBusObject &proxy) {
        std::string interface = config_["AllJoyn"]["client_interface"];
        std::shared_ptr <DistributedEnergyResource>
                der (new DistributedEnergyResource (init, proxy, interface));
        Logger("Property", log_path_)
                << der->GetUID () << '\t'
                << der->GetPath () << '\t'
                << der->GetExportRamp () << '\t'
                << der->GetRatedExportPower () << '\t'
                << der->GetRatedExportEnergy () << '\t'
                << der->GetImportRamp () << '\t'
                << der->GetRatedImportPower () << '\t'
                << der->GetRatedImportEnergy () << '\t'
                << der->GetIdleLosses ();
        der->RemoteImportPower (0);
        resources_.push_back (std::move (der));
        Aggregator::FilterResources ();
}  // end Add Resource

// Update Resource
// - when the Client Listener gets a property update signal it will look for the
// - correct resource and update it's properties
void Aggregator::UpdateResource (std::map <std::string, unsigned int>& init,
                                                                  const std::string& uid)
                                                                   ↪  {
        bool found = false;
        for (auto& resource : resources_) {
                if (resource->GetUID() == uid) {
                        resource->SetRatedExportEnergy (init["rated_export_energy"]);
                        resource->SetRatedExportPower (init["rated_export_power"]);
                        resource->SetExportEnergy (init["export_energy"]);
                        resource->SetExportPower (init["export_power"]);
                        resource->SetExportRamp (init["export_ramp"]);
                        resource->SetRatedImportEnergy (init["rated_import_energy"]);
                        resource->SetRatedImportPower (init["rated_import_power"]);
                        resource->SetImportEnergy (init["import_energy"]);
                        resource->SetImportPower (init["import_power"]);
                        resource->SetImportRamp (init["import_ramp"]);
                        resource->SetIdleLosses (init["idle_losses"]);
                        found = true;
                }
        }

        if (found) {
                // do nothing
        } else {
                std::cout
                        << "Property update signal recieved from unknown resource!"
                        << std::endl;
        }
}  // end Update Resource

// Remove Resource
// - if the Client Listener recieves a object loss signal then it will remove
// - it from the resource list
void Aggregator::RemoveResource (const std::string& uid) {
    // (TS): unlike a normal for loop, this loops iterator must be
    //       incremented only if the element is not deleted or you get
    //       memory leaks.
    for (auto it = resources_.begin(); it != resources_.end();) {
        if ((*it)->GetUID ().find(uid) != std::string::npos) {
            it = resources_.erase (it);
        } else {
            it++;
        }
    }
    Aggregator::FilterResources ();
}  // end Remove Resource

// Loop
// - check the import and export watts to disptach remote devices and
// - digital twins. also call the log function to log all discovered DER
// - properties
void Aggregator::Loop (float delta_time) {
    // update all resources
    for (auto &resource : resources_) {
                resource->Loop (delta_time);
    }
        Aggregator::UpdateTotals ();
        Aggregator::ExportPower ();
        Aggregator::ImportPower ();
        Aggregator::Log ();
        Aggregator::SetTime ();
}  // end Loop

// Log
// - log all resource properties based on the set log increment.
// - TODO (TS): I should write this function to write to file once instead of
// - for each resource, but we will see how much of performance loss we have.
void Aggregator::Log () {
    // log resources based on elapsed time
    unsigned int utc = time(0);
    if (utc != last_log_ && utc % log_inc_ == 0) {
            for (const auto &resource : resources_) {
                        Logger("Data", log_path_)
                                << resource->GetUID () << '\t'
                                << resource->GetPath () << '\t'
                                << resource->GetExportWatts () << '\t'
                                << resource->GetExportPower () << '\t'
                                << resource->GetExportEnergy () << '\t'
                                << resource->GetImportWatts () << '\t'
                                << resource->GetImportPower () << '\t'
                                << resource->GetImportEnergy ();
                    }
                        last_log_ = utc;
            }
}

void Aggregator::DisplayAllResources () {
    std::cout << "\nAll Resources:" << std::endl;
    for (const auto &resource : resources_) {
        resource->Print ();
    }
}

void Aggregator::DisplayTargetResources () {
    std::cout << "\nTarget Resources:" << std::endl;
    for (const auto &resource : sub_resources_) {
        resource->Print ();
    }
}

void Aggregator::UpdateTotals () {
        total_export_energy_ = 0;
        total_export_power_ = 0;
        total_import_energy_ = 0;
        total_import_power_ = 0;
    for (const auto &resource : sub_resources_) {
                total_export_energy_ += resource->GetExportEnergy ();
                total_export_power_ += resource->GetRatedExportPower ();
                total_import_energy_ += resource->GetImportEnergy ();
                total_import_power_ += resource->GetRatedImportPower ();
    }
}

void Aggregator::DisplaySummary () {
    std::cout << "\nAggregated Properties:"
    << "\n\tResource Count = " << resources_.size()
                << "\n\tTotal Export Energy = " << total_export_energy_
                << "\n\tTotal Export Power = " << total_export_power_
                << "\n\tTotal Import Energy = " << total_import_energy_
                << "\n\tTotal Import Power = " << total_import_power_ << std::endl;
}

// Filter Resources
// - filter resources by target arguments.
// - if target arguments is empty, then default to all resources
void Aggregator::FilterResources () {
    sub_resources_.clear();
    if (targets_.size() == 0) {
            sub_resources_ = resources_;
    } else {
            for (const auto &resource : resources_) {
                bool found;
                for (unsigned int i = 0; i < targets_.size(); i++) {
                    if (resource->GetPath().find(targets_[i])!= std::string::npos) {
```

```
                    found = true;
                } else {
                    found = false;
                    break;
                }
            }
            if (found) {
                sub_resources_.push_back(resource);
            }
        }
    }
}
// Export Power
// - loop through target resources and send export signal based on greatest
// - export energy available. The signal sets both the "digital twin" and the
// - remote devices control watts;
void Aggregator::ExportPower () {
        // sort by ramp first then export energy
    std::sort(
            sub_resources_.begin(),sub_resources_.end(), [] (
        const std::shared_ptr <DistributedEnergyResource> lhs,
        const std::shared_ptr <DistributedEnergyResource> rhs) {
                // ramp rate check
                if (lhs->GetExportRamp () != rhs->GetExportRamp ()) {
                        return (lhs->GetExportRamp () > rhs->GetExportRamp ());
                }

                // energy check
                return (lhs->GetExportEnergy () > rhs->GetExportEnergy ());
        });

    unsigned int dispatch_power = export_watts_;
    unsigned int power = 0;
    for (auto &resource : sub_resources_) {
                if (dispatch_power > 0) {
                        power = resource->GetRatedExportPower ();
                    if (resource->GetExportPower () == 0
                            && resource->GetExportWatts () == 0) {
                                // AllJoyn Method Call and digital twin
                            resource->RemoteExportPower (power);
                    }
                    // subtract resources power from dispatch power
                    if (dispatch_power > power) {
                            dispatch_power -= power;
                    } else {
                                dispatch_power = 0;
                    }
                // once dispatch has been met tell other resources to stop exporting
                } else {
                    if (resource->GetExportWatts () != 0) {
                            // AllJoyn Method Call
                            resource->RemoteExportPower (0);
                    }
                }
    }
}  // end Export Power

// Import Power
// - loop through target resources and send import signal based on greatest
// - import energy available. The signal sets both the "digital twin" and the
// - remote devices control watts;
void Aggregator::ImportPower () {
        // sort by ramp first then by import power
    std::sort(
            sub_resources_.begin(),sub_resources_.end(), [] (
                const std::shared_ptr <DistributedEnergyResource> lhs,
                const std::shared_ptr <DistributedEnergyResource> rhs) {
                // ramp rate check
                if (lhs->GetImportRamp () != rhs->GetImportRamp ()) {
                        return (lhs->GetImportRamp () > rhs->GetImportRamp ());
                }

                // energy check
                return (lhs->GetImportEnergy () > rhs->GetImportEnergy ());
        });

    unsigned int dispatch_power = import_watts_;
    unsigned int power = 0;
    for (auto &resource : sub_resources_) {
                if (dispatch_power > 0) {
                        power = resource->GetRatedImportPower ();
                    if (resource->GetImportPower () == 0
                            && resource->GetImportWatts () == 0) {
                            // AllJoyn Method Call and digital twin
                            resource->RemoteImportPower (power);
                    }

                    // subtract resources power from dispatch power
                    if (dispatch_power > power) {
                            dispatch_power -= power;
                    } else {
                                dispatch_power = 0;
                    }
                // once dispatch has been met tell other resources to stop importing
                } else {
                    if (resource->GetImportWatts () != 0) {
                            // AllJoyn Method Call
                            resource->RemoteImportPower (0);
                    }
                }
        }
}  // end Import Power
```

A.1.3 ClientListener.cpp

```
#include <iostream>
#include <alljoyn/ProxyBusObject.h>
#include <alljoyn/BusAttachment.h>
#include <alljoyn/Observer.h>
#include "include/ClientListener.h"
#include "include/logger.h"

// (TS): this is the only way I could find to initialize a const char* array.
//       AllJoyn documentation states "NULL" for registering all properties, but
//       that didn't seem to work.
const char* ClientListener::props_[] = {"rated_export_power",
                                        "rated_export_energy",
                                        "export_power",
                                        "export_energy",
                                        "export_ramp",
                                        "rated_import_power",
                                        "rated_import_energy",
                                        "import_power",
                                        "import_energy",
                                        "import_ramp",
                                        "idle_losses"};

ClientListener::ClientListener(
    ajn::BusAttachment* bus,
    ajn::Observer* obs,
    Aggregator* vpp,
    const char* client_name) : bus_ptr_(bus),
                               obs_ptr_(obs),
                               vpp_ptr_(vpp),
                               client_interface_(client_name){
} // end ClientListener

// ObjectDiscovered
// - a DCS has advertised the interface we are looking for so DERAS
// - request and update of it's current properties to ensure the digital twin is
// - up to date.
void ClientListener::ObjectDiscovered (ajn::ProxyBusObject& proxy) {
    std::string path = proxy.GetPath();
    std::string service_name = proxy.GetServiceName();
    std::string name = proxy.GetUniqueName ();

    std::cout << "\n[LISTENER]\n";
    std::cout << "\tPath = " << path << '\n';
    std::cout << "\tService Name = " << service_name << '\n';
    std::cout << "\tUID = " << name << '\n';

    Logger("Resources", "/home/deras/dev/LOGS/")
        << name << '\t'
        << path << '\t'
        << "found";


    bus_ptr_->EnableConcurrentCallbacks();
    proxy.RegisterPropertiesChangedListener(
        client_interface_, props_, 11, *this, NULL
    );

    ajn::MsgArg values;
    proxy.GetAllProperties (client_interface_, values);

    std::map <std::string, unsigned int> init;
    init = ClientListener::MapProperties (values);

    vpp_ptr_->AddResource (init, proxy);
} // end ObjectDiscovered

// ObjectLost
// - the remote device is no longer available
// - RemoveResource from aggregator using path
void ClientListener::ObjectLost (ajn::ProxyBusObject& proxy) {
    std::string name = proxy.GetUniqueName();
    std::string path = proxy.GetPath();

    std::cout << "\n[LISTENER] : " << name << " connection lost\n";
    std::cout << "\tPath : " << path << " no longer exists\n";

    Logger("Resources", "/home/deras/dev/LOGS/")
        << name << '\t'
        << path << '\t'
        << "lost";
    vpp_ptr_->RemoveResource (name);
} // end ObjectLost

// PropertiesChanged
// - callback to recieve property changed event from remote bus object
void ClientListener::PropertiesChanged (ajn::ProxyBusObject& obj,
                                        const char* interface_name,
                                        const ajn::MsgArg& changed,
                                        const ajn::MsgArg& invalidated,
                                        void* context) {
    std::map <std::string, unsigned int> init;
    init = ClientListener::MapProperties (changed);
    vpp_ptr_->UpdateResource (init, obj.GetUniqueName ());
} // end PropertiesChanged

std::map <std::string, unsigned int> ClientListener::MapProperties (
    const ajn::MsgArg& properties) {
    std::map <std::string, unsigned int> init;
    size_t nelem = 0;
    ajn::MsgArg* elems = NULL;
    QStatus status = properties.Get("a{sv}", &nelem, &elems);
    if (status == ER_OK) {
        const char* name;
        unsigned int property;
        ajn::MsgArg* val;
        for (size_t i = 0; i < nelem; i++) {
            status = elems[i].Get("{sv}", &name, &val);
            val->Get("u", &property);
            init[name] = property;
        }
    }
    return init;
}
```

A.1.4 CommandLineInterface.cpp

```
// INCLUDES
#include <iostream>
#include <sstream>
#include <vector>
#include "include/CommandLineInterface.h"

// Constructor
// - pass pointer to aggregator object for control
CommandLineInterface::CommandLineInterface (Aggregator* vpp, Operator* opr)
    : vpp_ptr_(vpp), oper_ptr_(opr) {
}  // end constructor

CommandLineInterface::~CommandLineInterface () {
    // do nothing at this time
}  // end destructor

// Help
// - CLI interface description
void CommandLineInterface::Help () {
    std::cout << "\n\t[Help Menu]\n\n"
        << "> q                 quit\n"
        << "> h                 display help menu\n"
        << "> a                 display all resources\n"
        << "> f                 display filtered resources\n"
        << "> s                 display summary\n"
        << "> t <arg arg...>    targets filter\n"
        << "> i <watts>         import power\n"
        << "> e <watts>         export power\n"
        << "> p <1/10 of cent>  power price\n"
        << "> w <F>             weather (deg. F)\n"
        << "> o <service>       set operator service\n"
        << "                    OFF: disable operator\n"
        << "                    PJMA: Regulation A\n"
        << "                    PJMD: Regulation D\n"
        << "                    EIM: Energy Imbalance Market\n"
        << "                    TOU: Time of Use\n"
        << "                    PDM: Peak Demand Mitigation\n"
        << "                    FER: Frequency Event Response\n" << std::endl;
} // end Help

// Command Line Interface
// - method to allow user controls during program run-time
bool CommandLineInterface::Control (const std::string& input) {
    // check for program argument
    if (input == "") {
        CommandLineInterface::Help ();
        return false;
    }
    char cmd = input[0];

    // deliminate input string to argument parameters
    std::vector <std::string> tokens;
    std::stringstream ss(input);
    std::string token;
    while (ss >> token) {
        tokens.push_back(token);
    }

    switch (cmd) {
        case 'q': {
           return true;
        }

        case 'a': {
            vpp_ptr_->DisplayAllResources ();
            break;
        }
        case 'f': {
            vpp_ptr_->DisplayTargetResources ();
            break;
        }
        case 's': {
            oper_ptr_->Summary ();
            vpp_ptr_->DisplaySummary ();
            break;
        }
        case 't': {
            try {
                tokens.erase (tokens.begin ());  // remove command
                vpp_ptr_->SetTargets (tokens);
                break;
            } catch (...) {
                std::cout << "[ERROR]: Invalid Argument.\n";
                break;
            }
        }
        case 'o': {
            try {
                oper_ptr_->SetService(tokens.at (1));
                break;
            } catch(...) {
                std::cout << "[ERROR]: Invalid Argument.\n";
                break;
            }
        }
        case 'i': {
            try {
                vpp_ptr_->SetImportWatts (stoul (tokens.at (1)));
                break;
            } catch (...) {
                std::cout << "[ERROR]: Invalid Argument.\n";
                break;
            }
        }
        case 'e': {
            try {
                vpp_ptr_->SetExportWatts (stoul (tokens.at (1)));
                break;
            } catch (...) {
                std::cout << "[ERROR]: Invalid Argument.\n";
                break;
            }
        }
        case 'p': {
            try {
                vpp_ptr_->SetPrice (stoul (tokens.at (1)));
                break;
            } catch (...) {
                std::cout << "[ERROR]: Invalid Argument.\n";
                break;
            }
        }
        case 'w': {
            try {
                vpp_ptr_->SetTemperature (stoi (tokens.at (1)));
                break;
            } catch (...) {
                std::cout << "[ERROR]: Invalid Argument.\n";
                break;
            }
        }
        default: {
            CommandLineInterface::Help ();
            break;
        }
    }

    return false;
}  // end Command Line Interface
```

A.1.5 DistributedEnergyResource.cpp

```
#include <iostream>
#include <string>
#include <map>

#include "include/DistributedEnergyResource.h"

#define DEBUG(x) std::cout << x << std::endl

// Constructor
// - this will be used to populate all the DER properties when alljoyn finds
// - a new device advertised
DistributedEnergyResource::DistributedEnergyResource (
    std::map <std::string, unsigned int> &init,
    ajn::ProxyBusObject &proxy,
    std::string interface) :
    proxy_(proxy),
    rated_export_power_(init["rated_export_power"]),
    rated_export_energy_(init["rated_export_energy"]),
    export_ramp_(init["export_ramp"]),
    rated_import_power_(init["rated_import_power"]),
    rated_import_energy_(init["rated_import_energy"]),
    import_ramp_(init["import_ramp"]),
    idle_losses_(init["idle_losses"]),
    export_power_(init["export_power"]),
    export_energy_(init["export_energy"]),
    import_power_(init["import_power"]),
    import_energy_(init["import_energy"]),
    export_watts_(init["export_power"]),
    import_watts_(init["import_power"]),
    delta_time_(0),
    interface_(interface) {
    //ctor
}

DistributedEnergyResource::~DistributedEnergyResource () {
    //dtor
}

// Remote Export Power
// - send control signal to remote device through alljoyn proxy method call
void DistributedEnergyResource::RemoteExportPower (unsigned int power) {
    DistributedEnergyResource::SetExportWatts (power);
    ajn::MsgArg arg("u",export_watts_);
    proxy_.MethodCall(interface_.c_str(),
                      "ExportPower",
                      &arg,
                      1,
                      ajn::ALLJOYN_FLAG_NO_REPLY_EXPECTED
    );
}  // end Remote Export Power

// Remote Import Power
// - send control signal to remote device through alljoyn proxy method call
void DistributedEnergyResource::RemoteImportPower (unsigned int power) {
    DistributedEnergyResource::SetImportWatts (power);
    ajn::MsgArg arg("u",import_watts_);
    proxy_.MethodCall(interface_.c_str(),
                      "ImportPower",
                      &arg,
                      1,
                      ajn::ALLJOYN_FLAG_NO_REPLY_EXPECTED
    );
}  // end Remote Import Power

// Control
// - check state of import / export power properties from main loop on a timer
void DistributedEnergyResource::Loop (float delta_time) {
    delta_time_ = delta_time;

    if (import_watts_ > 0) {
        DistributedEnergyResource::ImportPower ();
    } else if (export_watts_ > 0) {
        DistributedEnergyResource::ExportPower ();
    } else {
        IdleLoss ();
    }
}  // end Control

// Print
// - a method of quickly printing important properties of the DER
void DistributedEnergyResource::Print () {
    std::cout << "\n[DER]: " << proxy_.GetPath() << std::endl;
    std::cout
        << "\tExport Energy:\t" << export_energy_ << '\n'
        << "\tExport Power:\t" << export_power_ << '\n'
        << "\tExport watts:\t" << export_watts_ << '\n'
        << "\tImport Energy:\t" << import_energy_ << '\n'
        << "\tImport Power:\t" << import_power_ << '\n'
        << "\tImport watts:\t" << import_watts_ << '\n'<< std::endl;
}

// Set Export Watts
// - export watts is used as a control setpoint by ExportPower and turns import
// - power off.
```

```
void DistributedEnergyResource::SetExportWatts (unsigned int power) {
    import_watts_ = 0;
    import_power_ = 0;

    if (power > rated_export_power_) {
        export_watts_ = rated_export_power_;
    } else {
        export_watts_ = power;
    }
}  // end Set Export Watts

// Set Rated Export Power
// - set the watt value available to export to the grid
void DistributedEnergyResource::SetRatedExportPower (unsigned int power) {
    rated_export_power_ = power;
}  // end Rated Export Power

// Set Rated Export Energy
// - set the watt-hour value available to export to the grid
void DistributedEnergyResource::SetRatedExportEnergy (unsigned int energy) {
    rated_export_energy_ = energy;
}  // end Set Export Energy

// Set Export Power
// - regulates export power
void DistributedEnergyResource::SetExportPower (float power) {
    if (power > export_watts_) {
        export_power_ = export_watts_;
    } else if (power <= 0) {
        export_power_ = 0;
    } else {
        export_power_ = power;
    }
}  // end Set Export Power

// Set Export Energy
// - regulates export energy
void DistributedEnergyResource::SetExportEnergy (float energy) {
    if (energy > rated_export_energy_) {
        export_energy_ = rated_export_energy_;
    } else if (energy <= 0) {
        export_energy_ = 0;
    } else {
        export_energy_ = energy;
    }
}  // end Set Export Energy

// Set Export Ramp
// - set the watt per second value available to export to the grid
void DistributedEnergyResource::SetExportRamp (unsigned int ramp) {
    export_ramp_ = ramp;
}  // end Set Export Ramp

// Get Rated Export Power
// - get the rated watt value available to export to the grid
unsigned int DistributedEnergyResource::GetRatedExportPower () {
    return rated_export_power_;
}  // end Get Rated Export Power

// Get Rated Export Energy
// - get the watt value available to import from the grid
unsigned int DistributedEnergyResource::GetRatedExportEnergy () {
    return rated_export_energy_;
}  // end Rated Export energy

// Get Export Power
// - get the watt value available to export to the grid
unsigned int DistributedEnergyResource::GetExportPower () {
    unsigned int power = export_power_;
    return power;
}  // end Get Export Power

// Get Export Energy
// - get the watt-hour value available to export to the grid
unsigned int DistributedEnergyResource::GetExportEnergy () {
    unsigned int energy = export_energy_;
    return energy;
}  // end Get Export Energy

// Get Export Ramp
// - get the watt per second value available to export to the grid
unsigned int DistributedEnergyResource::GetExportRamp () {
    return export_ramp_;
}  // end Get Export Ramp

// Get Export Watts
// - get the control watts
unsigned int DistributedEnergyResource::GetExportWatts () {
    return export_watts_;
}  // end Get Export Watts

// Set Import Watts
// - turn off export power and set control setting for ImportPower method
void DistributedEnergyResource::SetImportWatts (unsigned int power) {
    export_watts_ = 0;
    export_power_ = 0;
    if (power > rated_import_power_) {
        import_watts_ = rated_import_power_;
    } else {
        import_watts_ = power;
    }
}  // end Set Import Watts

// Set Rated Import Power
// - set the watt value available to import from the grid
void DistributedEnergyResource::SetRatedImportPower (unsigned int power) {
    rated_import_power_ = power;
}  // end Set Rated Import Power

// Set Rated Import Energy
// - set the watt-hour value available to import from the grid
void DistributedEnergyResource::SetRatedImportEnergy (unsigned int energy) {
    rated_import_energy_ = energy;
}  // end Set Import Energy

// Set Import Power
// - regulates import power
void DistributedEnergyResource::SetImportPower (float power) {
    if (power > import_watts_) {
        import_power_ = import_watts_;
    } else if (power <= 0) {
        import_power_ = 0;
    } else {
        import_power_ = power;
    }
}  // end Set Import Power

// Set Import Energy
// - regulates import energy balance export energy
void DistributedEnergyResource::SetImportEnergy (float energy) {
    if (energy > rated_import_energy_) {
        import_energy_ = rated_import_energy_;
    } else if (energy <= 0) {
        import_energy_ = 0;
    } else {
        import_energy_ = energy;
    }
}  // end Set Import Energy

// Set Import Ramp
// - set the watt per second value available to import from the grid
void DistributedEnergyResource::SetImportRamp (unsigned int ramp) {
    import_ramp_ = ramp;
}  // end Set Import Ramp

// Get Rated Import Power
// - get the rated watt value available to import from the grid
unsigned int DistributedEnergyResource::GetRatedImportPower () {
    return rated_import_power_;
}  // end Rated Import Power

// Get Rated Import Energy
// - get the watt value available to import from the grid
unsigned int DistributedEnergyResource::GetRatedImportEnergy () {
    return rated_import_energy_;
}  // end Rated Import energy

// Get Import Power
// - get the watt value available to import from the grid
unsigned int DistributedEnergyResource::GetImportPower () {
    unsigned int power = import_power_;
    return power;
}  // end Get Import Power

// Get Import Energy
// - get the watt-hour value available to import from the grid
unsigned int DistributedEnergyResource::GetImportEnergy () {
    unsigned int energy = import_energy_;
    return energy;
}  // end Get Import Energy

// Get Import Ramp
// - get the watt per second value available to import from the grid
unsigned int DistributedEnergyResource::GetImportRamp () {
    return import_ramp_;
}  // end Get Import Ramp

// Get Import Watts
// - get the control watts
unsigned int DistributedEnergyResource::GetImportWatts () {
    return import_watts_;
}  // end Get Import Watts

// Set Idle Losses
// - set the watt-hours per hour loss when idle
void DistributedEnergyResource::SetIdleLosses (unsigned int losses) {
    idle_losses_ = losses;
}  // end Set Idle Losses

// Get Idle Losses
// - get the watt-hours per hour loss when idle
unsigned int DistributedEnergyResource::GetIdleLosses () {
    return idle_losses_;
}  // end Get Idle Losses
```

```
// Get Path
// - get the path to the DER
std::string DistributedEnergyResource::GetPath () {
    return proxy_.GetPath ();
}  // end Get Idle Losses

// Get UID
// - get the unique ID to the DER
std::string DistributedEnergyResource::GetUID () {
    return proxy_.GetUniqueName ();
}  // end Get UID

// Import Power
// - calculate power/energy change
// - degrement import energy and increment export energy
void DistributedEnergyResource::ImportPower () {
    // regulate import power
    float seconds = delta_time_ / 1000;
    float ramp_watts = import_ramp_ * seconds;
    if (import_power_ == import_watts_){
        // do nothing
    } else if (import_power_ < import_watts_) {
        DistributedEnergyResource::SetImportPower (import_power_ + ramp_watts);
    } else if (import_power_ > import_watts_) {
        DistributedEnergyResource::SetImportPower (import_watts_ - ramp_watts);
    }

    // regulate energy
    float hours = seconds / (60*60);
    float watt_hours = 0;

    // if the import power didn't reach rated then include ramp in energy calc
    // this function does not account for the cycle where it reaches rated,
    // but the error is negligable comparted to rated energy
    if (import_power_ < rated_import_power_) {
        // area under the linear function
        watt_hours += import_power_ * hours;  // area under triangle
        watt_hours += ramp_watts * hours/2;    // area of triangle
        DistributedEnergyResource::SetImportEnergy(import_energy_ - watt_hours);
        DistributedEnergyResource::SetExportEnergy(export_energy_ + watt_hours);
    } else {
        // area under the linear function excluding ramp.
        // note: this will exclude some energy if it reached peak within bounds
        watt_hours += import_power_ * hours;
        DistributedEnergyResource::SetImportEnergy(import_energy_ - watt_hours);
        DistributedEnergyResource::SetExportEnergy(export_energy_ + watt_hours);
    }
}  // end Import Power

// Export Power
// - calculate power/energy change
// - decrement export energy and increment import energy
void DistributedEnergyResource::ExportPower () {
    float seconds = delta_time_ / 1000;
    float ramp_watts = export_ramp_ * seconds;
    if (export_power_ == export_watts_) {
        // do nothing
    } else if (export_power_ < export_watts_) {
        DistributedEnergyResource::SetExportPower (export_power_ + ramp_watts);
    } else if (export_power_ > export_watts_) {
        DistributedEnergyResource::SetExportPower (export_power_ - ramp_watts);
    }

    // regulate energy
    float hours = seconds / (60*60);
    float watt_hours = 0;

    // if the import power didn't reach rated then include ramp in energy calc
    // this function does not account for the cycle where it reaches rated,
    // but the error is negligable comparted to rated energy
    if (export_power_ < rated_export_power_) {
        // area under the linear function
        watt_hours += export_power_ * hours;  // area under triangle
        watt_hours += ramp_watts * hours/2;    // area of triangle
        DistributedEnergyResource::SetImportEnergy(import_energy_ + watt_hours);
        DistributedEnergyResource::SetExportEnergy(export_energy_ - watt_hours);
    } else {
        // area under the linear function excluding ramp.
        // note: this will exclude some energy if it reached peak within bounds
        watt_hours += export_power_ * hours;
        DistributedEnergyResource::SetImportEnergy(import_energy_ + watt_hours);
        DistributedEnergyResource::SetExportEnergy(export_energy_ - watt_hours);
    }
}  // end Export Power

// Idle Loss
// - update energy available based on energy lost
void DistributedEnergyResource::IdleLoss () {
    // set import/export power to zero
    DistributedEnergyResource::SetImportPower (0);
    DistributedEnergyResource::SetExportPower (0);
    // set import / export idle loss energy changes
    float seconds = delta_time_ / 1000;
    float hours = seconds / (60*60);
    float energy_loss = idle_losses_ * hours;
    DistributedEnergyResource::SetImportEnergy(import_energy_ + energy_loss);
    DistributedEnergyResource::SetExportEnergy(export_energy_ - energy_loss);
}  // end Idle Loss
```

A.1.6 logger.cpp

```
#include <iostream>
#include <fstream>
#include <ctime>
#include "include/logger.h"

// the constructor starts the message with DateTime
Logger::Logger (std::string context, std::string path)
        : context_(context), path_(path) {
        msg_ =  Logger::GetDateTime () + '\t';
}  // end constructor

// becuase the logger object is constructor inline, it is destroyed at the end
// of the line which then writes the message to the log file.
Logger::~Logger () {
        Logger::ToFile ();
}  // end destructor

// To File
// - constructs the filename based on the date and context of the logger
void Logger::ToFile  () {
        std::string file_name = path_ + context_ + "_" + Logger::GetDate ()
                + ".log";
        std::ofstream output_file(file_name, std::ios_base::app);
        if (output_file.is_open()) {
                output_file << msg_ << '\n';
        }
        output_file.close();
}  // end To File

// Get Date
// - gets the local time date for the file name
std::string Logger::GetDate () {
    time_t now = time(0);
    struct tm ts = *localtime(&now);
    char buf[100];
    strftime(buf, sizeof(buf), "%F", &ts);
    return std::string(buf);
} // end Get Date

// Get Date Time
// - gets the local date and time for indexing of the data
std::string Logger::GetDateTime () {
    time_t now = time(0);
    struct tm ts = *localtime(&now);
    char buf[100];
    strftime(buf, sizeof(buf), "%F %T", &ts);
    return std::string(buf);
} // end Get Date Time
```

A.1.7 Operator.cpp

```
#include <iostream>
#include <ctime>
#include <numeric>
#include "include/Operator.h"
#include "include/tsu.h"
#include <cmath>

// constructor
Operator::Operator (std::map <std::string, std::string>& init,
                                        Aggregator* vpp_pointer)
                                        : vpp_ptr_(vpp_pointer),
                                          configs_(init),
                                          service_(""),
                                          tou_tier_(0),
                                          pjm_a_index_(0),
                                          pjm_d_index_(0),
                                          eim_index_(0),
                                          tou_index_(0),
                                          pdm_index_(0),
                                          fer_index_(0),
                                          prev_freqs_(60,60) {
        // do nothing
};

// destructor
Operator::~Operator () {
        // do nothing
};

// Loop
// - used to determine which services is active and call appropriate method
void Operator::Loop () {
        if (service_ == "OFF") {
                // do nothing
        } else if (service_ == "PJMA") {
                Operator::ServicePJMA ();
        } else if (service_ == "PJMD") {
                Operator::ServicePJMD ();
        } else if (service_ == "EIM") {
                Operator::ServiceEIM ();
        } else if (service_ == "TOU") {
                Operator::ServiceTOU ();
        } else if (service_ == "PDM") {
                Operator::ServicePDM ();
        } else if (service_ == "FER") {
                Operator::ServiceFER ();
        }
};  // end Loop

// Get Time
// - return HH:MM:SS formatted time
std::string Operator::GetTime (time_t utc) {
            struct tm ts = *localtime(&utc);
            char buf[100];
            strftime(buf, sizeof(buf), "%T", &ts);
            return std::string(buf);
}

// Set Service
// - mutator for the service variable
void Operator::SetService (std::string service) {
        if (service == "PJMA"
                || service == "PJMD"
                || service == "EIM"
                || service == "TOU"
                || service == "PDM"
                || service == "FER"
                || service == "OFF") {
                service_ = service;
        } else {
                std::cout << "Set Service Error: Invalid service type." << std::endl;
        }
};  // end Set Service

// Summary
// - display the current service and the last control sent
void Operator::Summary () {
        std::string last_time;
        float last_control;
        if (service_ == "OFF") {
                last_time = 1;
                last_control = 0;
        } else if (service_ == "PJMA") {
                RowPJM& row = schedule_pjm_a_.at(pjm_a_index_);
                last_time = row.time;
                last_control = row.normalized_power;
        } else if (service_ == "PJMD") {
                RowPJM& row = schedule_pjm_d_.at(pjm_d_index_);
                last_time = row.time;
                last_control = row.normalized_power;
        } else if (service_ == "EIM") {
                RowEIM& row = schedule_eim_.at(eim_index_);
                last_time = row.time;
                last_control = row.normalized_power;
        } else if (service_ == "TOU") {
            time_t now = time(nullptr);
                last_time = Operator::GetTime (now);
                last_control = tou_tier_;
        } else if (service_ == "PDM") {
                RowPDM& row = schedule_pdm_.at(pdm_index_);
                last_time = row.time;
                last_control = pdm_control_;
        } else if (service_ == "FER") {
                RowFER& row = schedule_fer_.at(fer_index_);
                last_time = row.time;
                last_control = fer_control_;
        }
        std::cout << "\nOperator:"
                << "\n\t service:\t" << service_
                << "\n\t last time:\t" << last_time
                << "\n\t last control:\t" << last_control << std::endl;

};  // end Summary

// Get PJM A
// - read the PJM schedule and format for use
void Operator::GetPJMA () {
        tsu::string_matrix schedule
                = tsu::FileToMatrix (configs_["pjma_filepath"], ',', 2);
        std::string time;
        float percent_power;
        schedule_pjm_a_.reserve (schedule.size ());
        schedule.erase (schedule.begin ());  // remove header
        for (const auto& row : schedule) {
            // http://man7.org/linux/man-pages/man3/strptime.3.html
            time = row.at(0);
                percent_power = stof (row.at (1));
                schedule_pjm_a_.emplace_back(time, percent_power);
        }
};  // end Get PJM A

// Get PJM D
// - read the PJM schedule and format for use
void Operator::GetPJMD () {
        tsu::string_matrix schedule
                = tsu::FileToMatrix (configs_["pjmd_filepath"], ',', 2);
        std::string time;
        float percent_power;
        schedule_pjm_d_.reserve (schedule.size ());
        schedule.erase (schedule.begin ());  // remove header
        for (const auto& row : schedule) {
            time = row.at(0);
                percent_power = stof (row.at (1));
                schedule_pjm_d_.emplace_back(time, percent_power);
        }
};  // end Get PJM D

// Get EIM
// - read the EIM schedule and format for use
void Operator::GetEIM () {
        tsu::string_matrix schedule
                = tsu::FileToMatrix (configs_["eim_filepath"], ',', 2);
        std::string time;
        float percent_power;
        schedule_eim_.reserve (schedule.size ());
        schedule.erase (schedule.begin ());  // remove header
        for (const auto& row : schedule) {
            time = row.at(0);
                percent_power = stof (row.at (1));
                std::cout << time << "\t" << percent_power << '\n';
                schedule_eim_.emplace_back(time, percent_power);
        }
};  // end Get EIM

// Get TOU
// - read the TOU schedule and format for use
void Operator::GetTOU () {
        tsu::string_matrix schedule
                = tsu::FileToMatrix (configs_["tou_filepath"], ',', 3);
        std::string time;
        float day_ahead;
        float real_time;
        schedule_tou_.reserve (schedule.size ());
        schedule.erase (schedule.begin ());  // remove header
        for (const auto& row : schedule) {
            time = row.at(0);
                day_ahead = stof (row.at (1));
                real_time = stof (row.at (2));
                schedule_tou_.emplace_back(time, real_time, day_ahead);
        }
};  // end Get TOU

// Get PDM
// - read the PDM schedule and format for use
void Operator::GetPDM () {
        tsu::string_matrix schedule
                = tsu::FileToMatrix (configs_["pdm_filepath"], ',', 2);
        std::string time;
        int fahrenheit;
        schedule_pdm_.reserve (schedule.size ());
        schedule.erase (schedule.begin ());  // remove header
        for (const auto& row : schedule) {
            time = row.at(0);
                fahrenheit = stoi (row.at (1));
                schedule_pdm_.emplace_back(time, fahrenheit);
        }
};  // end Get PDM

// Get FER
// - read the FER schedule and format for use
void Operator::GetFER () {
        tsu::string_matrix schedule
                = tsu::FileToMatrix (configs_["fer_filepath"], ',', 2);
        std::string time;
        float hertz;
        schedule_fer_.reserve (schedule.size ());
        schedule.erase (schedule.begin ());  // remove header
        for (const auto& row : schedule) {
            time = row.at(0);
                hertz = stof (row.at (1));
                schedule_fer_.emplace_back(time, hertz);
        }
};  // end Get FER
```

```
// Service PJM Reg A
// - Reg A is a normalized power control signal that is meant for traditional
// - regulating resources. Please reference PJM's Manual 12 for more information
// - The services will used the vpp resource info to determine dispatch and call
// - the appropriate control method.
void Operator::ServicePJMA () {
        // if the schedule has not been read, then load it into memory
        if (schedule_pjm_a_.empty()) {
                Operator::GetPJMA ();
        }

    time_t now = time(nullptr);
        std::string time = Operator::GetTime (now);

    // loop through each row of schedule looking for current utc
    for (unsigned int i = pjm_a_index_; i < schedule_pjm_a_.size(); i++) {
        RowPJM& row = schedule_pjm_a_.at (i);

        if ((row.time == time) && pjm_a_index_ != i) {
            // if the time is found then determine dispatch
            if (row.normalized_power > 0) {
                    std::vector <std::string> targets = {""};
                        vpp_ptr_->SetTargets(targets);
                    float available_watts = vpp_ptr_->GetTotalImportPower ();
                    float dispatch_watts = available_watts * row.normalized_power;
                vpp_ptr_->SetImportWatts (dispatch_watts);
            } else if (row.normalized_power < 0) {
                    std::vector <std::string> targets = {""};
                        vpp_ptr_->SetTargets(targets);
                    float available_watts = vpp_ptr_->GetTotalExportPower ();
                    float dispatch_watts = available_watts*(-row.normalized_power);
                vpp_ptr_->SetExportWatts (dispatch_watts);
            } else {
                    std::vector <std::string> targets = {""};
                        vpp_ptr_->SetTargets(targets);
                vpp_ptr_->SetImportWatts (0);
            }

            // log index so multiple controls are not sent and to reduce
            // search time.
            pjm_a_index_ = i;
        }
    }
};  // end Service PJM Reg A

// Service PJM Reg D
// - Reg D is a normalized power control signal that is meant for dynamic
// - regulating resources. Please reference PJM's Manual 12 for more information
// - The services will used the vpp resource info to determine dispatch and call
// - the appropriate control method.
void Operator::ServicePJMD () {
        // if the schedule has not been read, then load it into memory
        if (schedule_pjm_d_.empty()) {
                Operator::GetPJMD ();
        }

    time_t now = time(nullptr);
        std::string time = Operator::GetTime (now);

    // loop through each row of schedule looking for current utc
    for (unsigned int i = pjm_d_index_; i < schedule_pjm_d_.size(); i++) {
        RowPJM& row = schedule_pjm_d_.at (i);

        if ((row.time == time) && pjm_d_index_ != i) {

            // if the time is found then determine dispatch
            if (row.normalized_power > 0) {
                    std::vector <std::string> targets = {""};
                        vpp_ptr_->SetTargets(targets);
                    float available_watts = vpp_ptr_->GetTotalImportPower ();
                    float dispatch_watts = available_watts * row.normalized_power;
                vpp_ptr_->SetImportWatts (dispatch_watts);
            } else if (row.normalized_power < 0) {
                    std::vector <std::string> targets = {""};
                        vpp_ptr_->SetTargets(targets);
                    float available_watts = vpp_ptr_->GetTotalExportPower ();
                    float dispatch_watts = available_watts*(-row.normalized_power);
                vpp_ptr_->SetExportWatts (dispatch_watts);
            } else {
                    std::vector <std::string> targets = {""};
                        vpp_ptr_->SetTargets(targets);
                vpp_ptr_->SetImportWatts (0);
            }

            // store index so multiple control signals are not sent
            pjm_d_index_ = i;
        }
    }
};  // end Service PJM Reg D

// Service EIM
// - EIM is the energy imbalance market and is designed for Balancing Area
// - authorities. Please reference https://www.westerneim.com for more.
// - The services will used the vpp resource info to determine dispatch and call
// - the appropriate control method.
```

```
void Operator::ServiceEIM () {
        // if the schedule has not been read, then load it into memory
        if (schedule_eim_.empty()) {
                Operator::GetEIM ();
        }

    time_t now = time(nullptr);
        std::string time = Operator::GetTime (now);

    // loop through each row of schedule looking for current utc
    for (unsigned int i = eim_index_; i < schedule_eim_.size(); i++) {
        RowEIM& row = schedule_eim_.at (i);

        if ((row.time == time) && eim_index_ != i) {

            // if the time is found then determine dispatch
            if (row.normalized_power > 0) {
                    float available_watts = vpp_ptr_->GetTotalImportPower ();
                    float available_energy = vpp_ptr_->GetTotalImportEnergy ();
                    float import_time = available_energy / available_watts;
                    float distribute_power = available_energy / (4*import_time);  //
                      ↪  distributed over 4 times the time
                    float dispatch_watts = distribute_power * row.normalized_power;
                vpp_ptr_->SetImportWatts (dispatch_watts);
            } else if (row.normalized_power < 0) {
                    float available_watts = vpp_ptr_->GetTotalExportPower ();
                    float available_energy = vpp_ptr_->GetTotalImportEnergy ();
                    float export_time = available_energy / available_watts;
                    float distribute_power = available_energy / (4*export_time);  //
                      ↪  distributed over 4 times the time
                    float dispatch_watts = distribute_power * row.normalized_power;
                vpp_ptr_->SetExportWatts (dispatch_watts);
            } else {
                vpp_ptr_->SetImportWatts (0);
            }
                        // store index so multiple control signals are not sent
                        eim_index_ = i;
        }
    }
        if (eim_index_ == (schedule_eim_.size () - 1)) {
                eim_index_ = 0;
        }
};  // end Service EIM

// Service TOU
// - TOU is the Time of Use and will try to reduce the cost of importing
// - while maximizing the return for exporting to the grid.
// - Reference https://www.portlandgeneral.com for TOU info
void Operator::ServiceTOU () {
        // get current utc and modulo the date info out since it isn't required for
        // our tests.
        unsigned int seconds_per_day = 60*60*24;
        time_t time = std::time(nullptr);
        unsigned int utc = time % seconds_per_day;

        // Every minute determine TOU tier and set import/export accordingly
    if (utc % 60 == 0 && tou_index_ != utc) {
        // check season for tou tiers
        struct tm time_info = *std::localtime (&time);
        int month = time_info.tm_mon + 1;
        int hour = time_info.tm_hour;
        if (month >= MAY && month <= OCT) {
                // SUMMER TOU
                if (hour >= 22 && hour < 6) {
                        tou_tier_ = OFF_PEAK;
                } else if (hour >= 6 && hour < 15) {
                        tou_tier_ = MID_PEAK;
                } else if (hour >= 15 && hour < 20) {
                        tou_tier_ = ON_PEAK;
                } else {
                        tou_tier_ = MID_PEAK;
                }
        } else {
                // WINTER TOU
                if (hour >= 22 && hour < 6) {
                        tou_tier_ = OFF_PEAK;
                } else if (hour >= 6 && hour < 10) {
                        tou_tier_ = ON_PEAK;
                } else if (hour >= 10 && hour < 17) {
                        tou_tier_ = MID_PEAK;
                } else if (hour >= 17 && hour < 20) {
                        tou_tier_ = ON_PEAK;
                } else {
                        tou_tier_ = MID_PEAK;
                }
        }

        if (tou_tier_ == ON_PEAK) {
                // sell as much as possible
                std::vector <std::string> targets = {""};
                vpp_ptr_->SetTargets(targets);
                    float available_watts = vpp_ptr_->GetTotalExportPower ();
                vpp_ptr_->SetExportWatts (available_watts);
        } else if (tou_tier_ == MID_PEAK) {
                // only import on priority loads that cannot be time shifted
                std::vector <std::string> targets = {"buffer"};
                vpp_ptr_->SetTargets(targets);
                float available_watts = vpp_ptr_->GetTotalImportPower ();
                    vpp_ptr_->SetImportWatts (available_watts);
        } else {
                // import as much as possible
                std::vector <std::string> targets = {""};
                vpp_ptr_->SetTargets(targets);
                float available_watts = vpp_ptr_->GetTotalImportPower ();
                    vpp_ptr_->SetImportWatts (available_watts);
        }
        tou_index_ = utc;
    }
};  // end Service TOU

// Service PDM
// - PDM is the Peak Demand Mitigation and will try to reduce peak power under
// - specified conditions
void Operator::ServicePDM () {
        // if the schedule has not been read, then load it into memory
        if (schedule_pdm_.empty()) {
                Operator::GetPDM ();
        }

        // get current utc and modulo the date info out since it isn't required for
        // our tests.
        time_t time = std::time(nullptr);
        std::string f_time = Operator::GetTime (time);

    // loop through each row of schedule looking for current utc
    for (unsigned int i = pdm_index_; i < schedule_pdm_.size(); i++) {
        RowPDM& row = schedule_pdm_.at (i);

        if ((row.time == f_time) && pdm_index_ != i) {
                // get hour info
                struct tm time_info = *std::localtime (&time);
                int hour = time_info.tm_hour;
            // if the time is found then determine dispatch
            if (row.temperature > 85 && hour >= 18 && hour <= 21) {
                    std::vector <std::string> targets = {""};
                        vpp_ptr_->SetTargets(targets);
                vpp_ptr_->SetImportWatts (0);
                            float available_watts = vpp_ptr_->GetTotalExportPower ();
                        vpp_ptr_->SetExportWatts (available_watts);
                        pdm_control_ = true;
            } else if (row.temperature < 39 && hour >= 17 && hour <= 20) {
                    std::vector <std::string> targets = {""};
                        vpp_ptr_->SetTargets(targets);
                vpp_ptr_->SetImportWatts (0);
                            float available_watts = vpp_ptr_->GetTotalExportPower ();
                        vpp_ptr_->SetExportWatts (available_watts);
                        pdm_control_ = true;
            } else {
                    pdm_control_ = false;
            }

            // store index so multiple control signals are not sent
            pdm_index_ = i;
        }
    }
};  // end Service PDM

// Service FER
void Operator::ServiceFER () {
        unsigned int actual_time;
        float actual_hz;
        float delta_hz;
        float moving_avg;
        float floor_freq = 59.975;
        float min_slew_rate = 0.0031;
        float ceiling_freq = 120 - floor_freq;
        float t1 = 180.0/3600;
        float t2 = t1;
        float ramp_down_power;

        if (schedule_fer_.empty()) {
                Operator::GetFER ();
                std::cout << "File loaded" << std::endl;
        }

        time_t time = std::time(nullptr);
        std::string f_time = Operator::GetTime (time);

    // loop through each row of schedule looking for current utc
    for (unsigned int i = fer_index_; i < schedule_fer_.size(); i++) {
            RowFER& row = schedule_fer_.at (i);

        if (row.time == f_time && fer_index_ != i) {
                actual_time = time;
                actual_hz = row.frequency;

                delta_hz = actual_hz - prev_hz_;
                prev_hz_ = actual_hz;
                prev_freqs_.erase(prev_freqs_.begin());
                prev_freqs_.push_back(actual_hz);
```

```
moving_avg = accumulate(prev_freqs_.begin(),prev_freqs_.end(),0.0)
        / prev_freqs_.size();

//std::cout << "time:\t" << actual_time << std::endl;
//std::cout << "moving avg:\t" << moving_avg << std::endl;
//std::cout << "new freq:\t" << actual_hz << std::endl;
std::cout << "delta hz:\t" << delta_hz << std::endl;

if (actual_hz < floor_freq && actual_hz < moving_avg && delta_hz < 0) {
        std::cout << "Negative deviation detected"
                << "\n\t Actual Hz: " << actual_hz
                << "\n\t Moving Avg. : " << moving_avg << std::endl;

        neg_deviation_ = 1;
}

if (actual_hz > ceiling_freq && actual_hz > moving_avg && delta_hz > 0) {
        pos_deviation_ = 1;
}

if (neg_deviation_ == 1) {
        if (oneshot0_ == 1) {
                oneshot0_ = 0;
                event_start_hz_ = abs(delta_hz) + actual_hz;
                event_min_hz_ = 99;
                event_start_time_ = actual_time;
        }

        if (actual_hz < event_min_hz_) {
                event_min_hz_ = actual_hz;
        } else if ((actual_hz - event_min_hz_) > 0.003) {
                neg_deviation_ = 0;
                event_delta_hz_ = 0;
                event_duration_sec_ = 0;
        }
} else {
        oneshot0_ = 1;
}

if (pos_deviation_ == 1) {
        if (oneshot1_ == 1) {
                oneshot1_ = 0;
                event_start_hz_ = actual_hz - abs(delta_hz);
                event_max_hz_ = 0;
                event_start_time_ = actual_time;
        }

        if (actual_hz > event_max_hz_) {
                event_max_hz_ = actual_hz;
        } else if ((event_max_hz_ - actual_hz) > 0.003) {
                pos_deviation_ = 0;
                event_delta_hz_ = 0;
                event_duration_sec_ = 0;
        }
} else {
        oneshot1_ = 1;
}

if (neg_deviation_ == 1 && actual_hz < moving_avg) {
        event_delta_hz_ = event_start_hz_ - actual_hz;
        event_duration_sec_ = actual_time - event_start_time_;
}

if (pos_deviation_ == 1 && actual_hz > moving_avg) {
        event_delta_hz_ = actual_hz - event_start_hz_;
        event_duration_sec_ = actual_time - event_start_time_;
}


if (neg_response_timer_ == 1 && neg_response_sec_ <= 360) {
        neg_event_detected_ = 1;
        neg_response_sec_ = actual_time - neg_response_start_time_;
} else if (neg_deviation_ == 1 && event_duration_sec_ >= 1
        && (event_delta_hz_/event_duration_sec_) >= min_slew_rate
        && event_delta_hz_ >= (min_slew_rate * 10)) {
        neg_event_detected_ = 1;
        neg_response_timer_ = 1;
        neg_response_start_time_ = actual_time;
} else {
        neg_response_timer_ = 0;
        neg_response_sec_ = 0;
        neg_event_detected_ = 0;
}

if (pos_response_timer_ == 1 && pos_response_sec_ <= 360) {
        pos_event_detected_ = 1;
        pos_response_sec_ = actual_time - pos_response_start_time_;
} else if (pos_deviation_ == 1 && event_duration_sec_ >= 1
        && (event_delta_hz_/event_duration_sec_) >= min_slew_rate
        && event_delta_hz_ >= (min_slew_rate * 10)) {
        pos_event_detected_ = 1;
        pos_response_timer_ = 1;
        pos_response_start_time_ = actual_time;
} else {
        pos_response_timer_ = 0;
        pos_response_sec_ = 0;
        pos_event_detected_ = 0;
```

```
                }
                if (neg_event_detected_ == 1) {
                        pos_response_timer_ = 0;
                        pos_response_sec_ = 0;
                        pos_event_detected_ = 0;
                }
                //Log positive and negative event detection here
                if (neg_event_detected_ == 1) {
                        std::cout << "Negative event: "
                                << "\n\t time : " << actual_time << std::endl;
                } else if (pos_event_detected_ == 1) {
                        std::cout << "Positive event: "
                                << "\n\t time : " << actual_time << std::endl;
                }

                //Positive Response Algorithm
                if (pos_event_detected_ == 1 && pos_response_start_time_ == actual_time) {
                        unsigned int total_import_energy = vpp_ptr_->GetTotalImportEnergy
                          ↪   ();
                        std::cout << "Response total import energy:\t" <<
                          ↪   total_import_energy << std::endl;
                        float import_request = total_import_energy*pow(t1 + t2/2,-1);
                        unsigned int max_import_power = vpp_ptr_->GetTotalImportPower ();
                        if (import_request > max_import_power) {
                                import_request = max_import_power;
                        }
                        std::cout << "Response P (float):\t" << import_request <<
                          ↪   std::endl;
                        //import_power_request_ = total_import_energy*pow(t1 + t2/2,-1);
                        import_power_request_ = import_request;
                        std::cout << "Response P:\t" << import_power_request_ <<
                          ↪   std::endl;
                }
                if (pos_event_detected_ == 1 && pos_response_sec_ < 180) {
                        vpp_ptr_->SetImportWatts (import_power_request_);
                        std::cout << "Positive event response, import:\t" <<
                          ↪   import_power_request_ << std::endl;
                } else if (pos_event_detected_ == 1 && pos_response_sec_ < 360) {
                        ramp_down_power = (1 -
                          ↪   (pos_response_sec_-180)/180.0)*import_power_request_;
                        vpp_ptr_->SetImportWatts (ramp_down_power);
                        std::cout << "Event time elapsed:\t" << pos_response_sec_ <<
                          ↪   std::endl;
                        std::cout << "Positive event response, ramp down import:\t" <<
                          ↪   ramp_down_power << std::endl;
                } else if (pos_event_detected_ == 1 && pos_response_sec_ == 360) {
                        vpp_ptr_->SetImportWatts (0);
                }
                //Negative Response Algorithm
                if (neg_event_detected_ == 1 && neg_response_sec_ < 360) {
                        vpp_ptr_->SetImportWatts (0);
                        std::cout << "Negative event response, import set to 0" <<
                          ↪   std::endl;
                }
                fer_index_ = i;
        }
}

};  // end Service FER
```

A.1.8 SmartGridDevice.cpp

```
#include <alljoyn/Status.h>
#include <alljoyn/BusObject.h>
#include <alljoyn/BusAttachment.h>
#include "include/Aggregator.h"
#include "include/SmartGridDevice.h"

// Constructor
// - initialize bus object interface and smart grid device properties
SmartGridDevice::SmartGridDevice (ajn::BusAttachment* bus, Aggregator* vpp,
                                 const char* name,
                                 const char* path) : ajn::BusObject(path),
                                                    bus_ptr_(bus),
                                                    vpp_ptr_(vpp),
                                                    signal_(NULL),
                                                    interface_(name),
                                                    price_(0),
                                                    time_(0),
                                                    temp_(0) {
```

```
    const ajn::InterfaceDescription* interface
            = bus_ptr_->GetInterface(interface_);
    assert(interface != NULL);
    AddInterface(*interface, ANNOUNCED);
}
// Destructor
SmartGridDevice::~SmartGridDevice () {
  // do nothing
}  // end Destructor
// Get
// - this method will be called by DCS looking to get the updated DERAS
// - properties
QStatus SmartGridDevice::Get (const char* interface,
                              const char* property,
                              ajn::MsgArg& value) {
    QStatus status;
    if (strcmp(interface, interface_)) {
        return ER_FAIL;
    }
    if (!strcmp(property,"time")) {
        status = value.Set("u", time_);
        return status;
    } else if (!strcmp(property,"price")) {
        status = value.Set("i", price_);
        return status;
    } else if (!strcmp(property,"temperature")) {
        status = value.Set("i", temp_);
        return status;
    } else {
        return ER_FAIL;
    }
} // end Get
QStatus SmartGridDevice::SendPropertiesUpdate () {
  const char* props[] = { "time",
                          "price",
                          "temperature"};
    QStatus status;
    status = EmitPropChanged (interface_, props, 2, ajn::SESSION_ID_ALL_HOSTED);
    std::cout << "[AllJoyn]: DERAS properties changed signal." << std::endl;
    return status;
}  // end Send Properties Update
// Loop
// - if DERAS price or time have changed then send a property update signal to
// - connected DCS. This doesn't need a loop to restrict its frequency as it is
// - dependent on DERAS's property changes.
void SmartGridDevice::Loop () {
    int price = vpp_ptr_->GetPrice ();
    int temp = vpp_ptr_->GetTemperature ();
    unsigned int utc = vpp_ptr_->GetTime();
    if (utc != time_ || price != price_ || temp != temp_) {
        time_ = utc;
        price_ = price;
        temp_ = temp;
            SmartGridDevice::SendPropertiesUpdate ();
    }
}
```

A.1.9 Makefile

```
# This makefile is described by the following link
# https://hiltmon.com/blog/2015/09/28/the-simple-c-plus-plus-makefile-executable-edition/
CC := g++  #main compiler
# Project Directory
SRCDIR := src
BUILDDIR := obj
TARGETDIR := bin/debug
TARGET := $(TARGETDIR)/$(SRC)
# Auto get source files
SRCEXT := cpp
SOURCES := $(shell find $(SRCDIR) -type f -name *.$(SRCEXT))
OBJECTS := $(patsubst $(SRCDIR)/%,$(BUILDDIR)/%,$(SOURCES:.$(SRCEXT)=.o))
# General Requirments
ifeq ($(CPU), x86)
        CPUFLAGS := -m32
else
        CPUFLAGS :=
endif
```

```
CFLAGS := -Wall -pipe -std=c++11 -Wno-long-long -Wno-deprecated -g -DQCC_OS_LINUX
  ↪  -DQCC_OS_GROUP_POSIX -DQCC_DBG $(CPUFLAGS)
LIB := -lstdc++ -lpthread -lrt -lm $(PILIBS)
INC := -I src/include

# AllJoyn Requirments
LIB += -L$(AJ_LIB) -lalljoyn -lajrouter
INC += -I$(AJ_INC)

# BOOST Requirments
INC += -I$(BOOST_INC)

$(TARGET) : $(OBJECTS)
	@mkdir -p $(TARGETDIR)
	@echo "\n\tLinking $(TARGET)\n"; $(CC) $^ -o $(TARGET) $(LIB)

$(BUILDDIR)/%.o: $(SRCDIR)/%.$(SRCEXT)
	@mkdir -p $(BUILDDIR)
	@echo "\n\tCompiling $<...\n"; $(CC) $(CFLAGS) $(INC) -c -o $@ $<

clean:
	@echo "\n\tCleaning $(TARGET)\n"; $(RM) -r $(BUILDDIR) $(TARGET)

.PHONY: clean
```

A.2 DCS

A.2.1 main.cpp

```
/******************************************************************************
 *    Copyright (c) Open Connectivity Foundation (OCF), AllJoyn Open Source
 *    Project (AJOSP) Contributors and others.
 *
 *    SPDX-License-Identifier: Apache-2.0
 *
 *    All rights reserved. This program and the accompanying materials are
 *    made available under the terms of the Apache License, Version 2.0
 *    which accompanies this distribution, and is available at
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 *    Copyright (c) Open Connectivity Foundation and Contributors to AllSeen
 *    Alliance. All rights reserved.
 *
 *    Copyright (c) V2 Systems, LLC.  All rights reserved.
 *
 *    Permission to use, copy, modify, and/or distribute this software for
 *    any purpose with or without fee is hereby granted, provided that the
 *    above copyright notice and this permission notice appear in all
 *    copies.
 *
 *    THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
 *    WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED
 *    WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE
 *    AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL
 *    DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
 *    PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
 *    TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
 *    PERFORMANCE OF THIS SOFTWARE.
******************************************************************************/

// INCLUDES
#include <iostream>
#include <thread>
#include <chrono>
#include <string>
#include <vector>
#include <map>
#include "include/DistributedEnergyResource.h"
#include "include/WaterHeaterEmulator.h"
#include "include/CommandLineInterface.h"
#include "include/Operator.h"
#include "include/SmartGridDevice.h"
#include "include/ServerListener.h"
#include "include/tsu.h"
#include "include/aj_utility.h"

// NAMESPACES
using namespace std;
using namespace ajn;

// GLOBALS
bool done = false;              // signal program to stop
extern bool scheduled;  // toggle operator using program args/CLI

// Program Help
// - command line interface arguments during run, [] items have default values
static void ProgramHelp (const string& name) {
    cout << "\n[Usage] > " << name << " -c <file path> [-o <y/n>] -h\n"
            "\t[] means it has a default value\n"
            "\t -h \t help\n"
            "\t -c \t configuration filename"
        "\t -i \t ID number"
                "\t -o \t enable operator"  << endl;
}  // end Program Help

// Argument Parser
// - method to parse program initial parameters
static std::map <std::string, std::string> ArgumentParser (int argc,



    string name = argv[0];
    scheduled = false;

    // parse tokens
    map <string, string> parameters;
    parameters["ID"] = 1;
    string token, argument;

    for (int i = 1; i < argc; i = i+2){
        token = argv[i];

        // check to see if the is an argument for the program control token
        if (argc <= i+1) {
                cout << "[ERROR] : Invalid program argument: " << token << endl;
            ProgramHelp(name);
            exit(EXIT_FAILURE);
        } else {
                argument = argv[i+1];
        }


        if ((token == "-h")) {
            ProgramHelp(name);
            exit(EXIT_FAILURE);
        } else if ((token == "-c")) {
            parameters["config"] = argument;
        } else if ((token == "-o")) {
            if ((argument == "y")) {
                    scheduled = true;
            } else if ((argument == "n")) {
                scheduled = false;
            }
        } else if (token == "-i") {
            parameters["ID"] = argument;
        } else {
            cout << "[ERROR] : Invalid program argument: " << token << endl;
            ProgramHelp(name);
            exit(EXIT_FAILURE);
        }
    }
    return parameters;
}  // end Argument Parser

// THREADS
// -------

// Resource Loop
// - this loop runs the resource control loop at the desired frequency
// - it subtracks processing time of the Loop () function to make the frequency
// - more consistant
void ResourceLoop (unsigned int sleep, DistributedEnergyResource* der_ptr) {
    unsigned int time_remaining, time_past;
    unsigned int time_wait = sleep;
    auto time_start = chrono::high_resolution_clock::now();
    auto time_end = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> time_elapsed;

    while (!done) {
        time_start = chrono::high_resolution_clock::now();
            // time since last control call;
            time_elapsed = time_start - time_end;
            time_past = time_elapsed.count();
            der_ptr->Loop(time_past);
        time_end = chrono::high_resolution_clock::now();
        time_elapsed = time_end - time_start;

        // determine sleep duration after deducting process time
        if (time_wait - time_elapsed.count() > 0) {
            time_remaining = time_wait - time_elapsed.count();
            this_thread::sleep_for (chrono::milliseconds (time_remaining));
        }
    }
}  // end Resource Loop

// Operator Loop
// - this loop runs the resource control loop at the desired frequency
// - it subtracks processing time of the Loop () function to make the frequency
// - more consistant
void OperatorLoop (unsigned int sleep, Operator* oper_ptr) {
    unsigned int time_remaining;
    unsigned int time_wait = sleep;
    auto time_start = chrono::high_resolution_clock::now();
    auto time_end = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> time_elapsed;

    while (!done && scheduled) {
        time_start = chrono::high_resolution_clock::now();
        oper_ptr->Loop();
        time_end = chrono::high_resolution_clock::now();
        time_elapsed = time_end - time_start;

        // determine sleep duration after deducting process time
        if (time_wait - time_elapsed.count() > 0) {
            time_remaining = time_wait - time_elapsed.count();
            this_thread::sleep_for (chrono::milliseconds (time_remaining));
        }
    }
}  // end Resource Loop

// Smart Grid Device Loop
// - this loop runs the resource control loop at the desired frequency
// - it subtracks processing time of the Loop () function to make the frequency
// - more consistant
void SmartGridDeviceLoop (unsigned int sleep, SmartGridDevice* sgd_ptr) {
    unsigned int time_remaining;
    unsigned int time_wait = sleep;
    auto time_start = chrono::high_resolution_clock::now();
    auto time_end = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> time_elapsed;

    while (!done) {
        time_start = chrono::high_resolution_clock::now();
        sgd_ptr->Loop();
        time_end = chrono::high_resolution_clock::now();
        time_elapsed = time_end - time_start;

        // determine sleep duration after deducting process time
        if (time_wait - time_elapsed.count() > 0) {
            time_remaining = time_wait - time_elapsed.count();
            this_thread::sleep_for (chrono::milliseconds (time_remaining));
        }
    }
}  // end Resource Loop

// Main
// ----
int main (int argc, char** argv) {
    cout
        << "\n***********************************"
        << "\n*** Distributed Control System ***"
        << "\n***********************************\n";

    cout << "Initialization...\n";
    // if the config file is not passed to the program then exit
   if (strcmp(argv[1], "-c") != 0) {
        string name = argv[0];
        ProgramHelp(name);
        return EXIT_FAILURE;
    }
    map <string, string> arguments = ArgumentParser(argc, argv);

    // read config file for program configurations and object attributes
    tsu::config_map configs = tsu::MapConfigFile (arguments["config"]);

    cout << "\tCreating Distributed Energy Resource\n";
    // ~ reference DistributedEnergyResource and BatteryEnergyStorageSystem
    WaterHeaterEmulator* der_ptr
        = new WaterHeaterEmulator(configs, stoul(arguments["ID"]));

    cout << "\tCreating Operator\n";
    // ~ reference Operator.h
    Operator* oper_ptr = new Operator(configs["Operator"]["schedule"], der_ptr);

    cout << "\tCreating Command Line Interface\n";
    // ~ reference CommandLineInterface.h
    CommandLineInterface CLI(der_ptr);

    cout << "\tCreating AllJoyn Message Bus\n";
    try {
            cout << "\t\tInitializing AllJoyn...\n";
        // Must be called before any AllJoyn functionality
        AllJoynInit();
    } catch (exception &e) {
        cout << "[ERROR]: " << e.what() << endl;
        return EXIT_FAILURE;
    }

    #ifdef ROUTER
        try {
                cout << "\t\tInitializing AllJoyn Router...\n";
            // Must be called before any AllJoyn routing functionality
            AllJoynRouterInit();
        } catch (exception &e) {
            cout << "[ERROR]: " << e.what() << endl;
            return EXIT_FAILURE;
        }
    #endif // ROUTER

    cout << "\tCreating AllJoyn Bus Attachment\n";
```

```
// BusAttachment is the top-level object responsible for connecting to and
// optionally managing a message bus.
// ~ AllJoyn Docs
string app = configs["AllJoyn"]["app"];
bool allow_remote = true;
BusAttachment* bus_ptr = new BusAttachment(app.c_str(), allow_remote);

cout << "\tCreating AllJoyn About Data\n";
// The AboutObj class is responsible for transmitting information about the
// interfaces that are available for other applications to use.
// ~ AllJoyn Docs
AboutData about_data("en");
AboutObj* about_ptr = new AboutObj(*bus_ptr);

cout << "\tCreating AllJoyn Session Port\n";
// inform users of session related events
// ~ AllJoyn Docs
aj_utility::SessionPortListener SPL;
SessionPort port = stoul(configs["AllJoyn"]["port"]);

cout << "\tSetting up AllJoyn Bus Attachment...\n";
// ~ reference aj_utility.cpp
QStatus status = aj_utility::SetupBusAttachment (configs,
                                                 port,
                                                 SPL,
                                                 bus_ptr,
                                                 &about_data);

cout << "\tCreating AllJoyn Observer\n";
// takes care of discovery, session management and ProxyBusObject creation
// for bus objects.
// ~ AllJoyn Docs
const char* server_name = configs["AllJoyn"]["server_interface"].c_str();
Observer *obs_ptr = new Observer(*bus_ptr, &server_name, 1);

cout << "\tCreating AllJoyn Server Listener\n";
// ~ reference ServerListener.cpp
ServerListener *listner_ptr = new ServerListener(bus_ptr,
                                                 obs_ptr,
                                                 der_ptr,
                                                 server_name);
obs_ptr->RegisterListener(*listner_ptr);

cout << "\tCreating AllJoyn Smart Grid Device\n";
// ~ reference SmartGridDevice.cpp
const char* device_name = configs["AllJoyn"]["device_interface"].c_str();
string path = configs["AllJoyn"]["path"];
string region = "region" + to_string(rand() % 100) + "/";
string substation = "substation" + to_string(rand() % 100) + "/";
string feeder = "feeder" + to_string(rand() % 100) + "/";
path = path + region + substation + feeder + app + arguments["ID"];
SmartGridDevice *sgd_ptr = new SmartGridDevice(der_ptr,
                                               bus_ptr,
                                               device_name,
                                               path.c_str());

cout << "\t\tRegistering AllJoyn Smart Grid Device\n";
if (ER_OK != bus_ptr->RegisterBusObject(*sgd_ptr)){
    cout << "\t\t[ERROR]: Failed Registration!\n";
    return EXIT_FAILURE;
}
about_ptr->Announce(port, about_data);

// most objects will have a dedicated thread, but not all
cout << "\tSpawning threads...\n";
thread DER (ResourceLoop, stoul(configs["Threads"]["sleep"]), der_ptr);
thread OPER (
        OperatorLoop, stoul(configs["Threads"]["sleep"]), oper_ptr
);
thread SGD (
        SmartGridDeviceLoop, stoul(configs["Threads"]["sleep"]), sgd_ptr
);

// the CLI will control the program and can signal the program to stop
    cout << "Initialization complete...\n";
CLI.Help ();
string input;
while (!done) {
    getline(cin, input);
    done = CLI.Control (input);
}

// when done = true, the program begins the shutdown process
// TODO (TS): AllJoyn still leaves alot of errors when closing. The docs
// - dont really explain the shutdown procedure for lots of alljoyn objects
    cout << "Closing program...\n";

    // First join all active threads to main thread
    cout << "\tJoining threads\n";
    DER.join ();
    OPER.join ();
    SGD.join ();

cout << "\tUnregistering AllJoyn objects\n";
obs_ptr->UnregisterListener (*listner_ptr);
bus_ptr->UnregisterBusObject(*sgd_ptr);
status = bus_ptr->Stop ();
```

```
    status = bus_ptr->Join ();

    // Then delete all pointers that were created using "new" since they do not
    // automaticall deconstruct at the end of the program.
    cout << "\nDeleting pointers...\n";
    delete sgd_ptr;
    delete listner_ptr;
    delete obs_ptr;
    delete about_ptr;
    delete bus_ptr;
    delete oper_ptr;
    delete der_ptr;

    #ifdef ROUTER
        cout << "\tShutting down AllJoyn Router\n";
        status = AllJoynRouterShutdown ();
    #endif // ROUTER

    cout << "\tShutting down AllJoyn\n";
    status = AllJoynShutdown ();

    // return exit status
    return 0;
} // end main
```

A.2.2 ElectricWaterHeater.cpp

```
#include <iostream>
#include <sstream>
#include <chrono>
#include "include/ElectricWaterHeater.h"
#include "include/logger.h"
#include "include/tsu.h"
#include <time.h>

// MACROS
#define DEBUG(x) std::cout << x << std::endl

ElectricWaterHeater::ElectricWaterHeater (
        std::map <std::string, std::string> configs) :
                DistributedEnergyResource (),
                sp_(configs["serial_port"]),
                  current_transducer_1_(stoul(configs["mcp_channel"])),
                  heartbeat_(stoul(configs["ucm_heartbeat"])) {

        // verivy serial port is available and connected to UCM
        if (!sp_.open ()) {
                Logger("ERROR", GetLogPath ())
                        << "failed to open serial port: " << strerror(errno);
                exit (1);
        } else {
                device_ptr_ = cea2045::DeviceFactory::createUCM(&sp_, &ucm_);
                device_ptr_->start ();
                device_ptr_->basicOutsideCommConnectionStatus(
                        cea2045::OutsideCommuncatonStatusCode::Found
                );
        }
        response_codes_ = device_ptr_->querySuportDataLinkMessages().get();
        response_codes_ = device_ptr_->queryMaxPayload().get();
        response_codes_ = device_ptr_->querySuportIntermediateMessages().get();
        response_codes_ = device_ptr_->intermediateGetDeviceInformation().get();

        SetLogPath (configs["log_path"]);
        std::cout << "Log Path: " << GetLogPath () << std::endl;
        //last_utc_ = 0;
        SetLogIncrement (stoul(configs["log_inc"]));
        Logger("INFO", GetLogPath ()) << "startup complete";
        SetRatedImportPower (4500);
        SetExportEnergy (0);
        SetImportRamp (stoul(configs["rated_import_ramp"]));
        SetIdleLosses (100); //Based off of observed ambient losses

        ElectricWaterHeater::QueryProperties ();

}  // end Constructor

ElectricWaterHeater::~ElectricWaterHeater () {
        delete device_ptr_;
}  // end Destructor

// Begin critical peak event status
void ElectricWaterHeater::SetCriticalPeak () {
        device_ptr_->basicCriticalPeakEvent (0);
        opstate_ = 4;
        Logger("INFO", GetLogPath ()) << "Critical peak event command received";
}  /// end critical peak event status change

// Begin load up status
void ElectricWaterHeater::SetLoadUp () {
        opstate_ = 3;
        device_ptr_->basicLoadUp (0);
        Logger("INFO", GetLogPath ()) << "Load up command received";
```

```
}  /// end load up command

// Begin Grid Emergency status
void ElectricWaterHeater::SetGridEmergency () {
        opstate_ = 5;
        device_ptr_->basicGridEmergency (0);
        Logger("INFO", GetLogPath ()) << "Grid Emergency command received";
}  /// end grid emergency command

// Get Real Import Power
// - use current transducer and approximated voltage to get real power draw from
// - water heater
unsigned int ElectricWaterHeater::GetRealImportPower () {
        return current_transducer_1_.GetCurrent() * 240; // assuming Vrms = 240
}  // end Get Real Import Power

// Query Properties
// - update basic DER properties using EWH methods
void ElectricWaterHeater::QueryProperties () {
        device_ptr_->intermediateGetCommodity ();
        std::vector <CommodityData> commodities = ucm_.GetCommodityData ();
        for (const auto &commodity : commodities) {
                if (commodity.code == 0) {
                        SetImportPower (commodity.rate);
                } else if (commodity.code == 6) {
                        SetRatedImportEnergy (commodity.cumulative);
                } else if (commodity.code == 7) {
                        SetImportEnergy (commodity.cumulative);
                }
        }
        //Query operational state
        device_ptr_->basicQueryOperationalState();
        //unsigned int state = ucm_.GetOpState ();
        //std::cout << "Debugging: Opstate = " << state << std::endl;
}  // end Query Properties


// End Curtailment events
void ElectricWaterHeater::EndCurtailment () {
        device_ptr_->basicEndShed (0);
        std::cout << "Debugging: Ending previous curtailment for new command." <<
          ↪   std::endl;
} // end End Curtailment events

/////////////////
// DER Overwrites
/////////////////

// Import Power
// - send a EndShed command through the UCM with a zero duration for undefined
// - amount of time
void ElectricWaterHeater::ImportPower () {
        device_ptr_->basicLoadUp (0);  // zero duration means as along as possible
        //device_ptr_->basicEndShed (0);
}  // end Import Power

// Export Power
// - DO NOTHING since water heaters cannot export power
void ElectricWaterHeater::ExportPower () {
        // just overwrite so DER Loop () doesn't do anything to properties
        SetExportEnergy (0);
}  // end Export Power

// Idle Loss
// - probably not the best name for this method, but when idle we want the
// - water heater to no be on so we send a Shed () command through the UCM with
// - a duration of 0 for undefined amount of time
void ElectricWaterHeater::IdleLoss () {
        device_ptr_->basicShed (0);
}

// Loop
void ElectricWaterHeater::Loop (float time_past) {
        time_t now = time(0);
        struct tm time = *localtime(&now);
        unsigned int sec = time.tm_sec;
        unsigned int min = time.tm_min;

        if (sec % 2 == 0){
                ElectricWaterHeater::QueryProperties ();
        }
        if (min % heartbeat_ == 0 && sec < 1){
                device_ptr_->basicOutsideCommConnectionStatus(
                        cea2045::OutsideCommuncatonStatusCode::Found
                );
        }

        // log every minuteish, there will be some double logs here and there
        if (sec == 0 && log_minute_ != min){
                ElectricWaterHeater::Log ();
                log_minute_ = min;
        }

        if (GetImportWatts () > 0 && GetImportPower () == 0) {
                if (ucm_.GetOpState () != HEIGHTENED) {
                        ElectricWaterHeater::ImportPower ();
                }
        } else if (GetImportPower () > 0 && GetImportWatts () == 0 ) {
                if (ucm_.GetOpState () != GRID || ucm_.GetOpState () != CURTAILED) {
                        //ElectricWaterHeater::IdleLoss ();
                        device_ptr_->basicCriticalPeakEvent (0);
                }
        }

}  // end Loop

// Log
// - log important physical attributes of DER on a frequency set by config file
void ElectricWaterHeater::Log () {
    unsigned int utc = time (0);
        Logger ("DER_Data", GetLogPath ())
            << GetExportWatts () << "\t"
            << GetExportPower () << "\t"
            << GetExportEnergy () << "\t"
            << GetImportWatts () << "\t"
            << GetImportPower () << "\t"
            << GetImportEnergy () << "\t"
            << GetRatedImportEnergy () << "\t"
            << ElectricWaterHeater::GetRealImportPower () << "\t"
            << ucm_.GetOpState ();
        SetLastUTC(utc);
}  // end Log

// Display
// - print device properties to terminal
void ElectricWaterHeater::Display () {
    std::cout << "Rated Import Energy:\t" << GetRatedImportEnergy () << "\twatt-hours\n";
    std::cout << "Operational State:\t" << ucm_.GetOpState () << "\n";
    std::cout << "Import Control:\t\t" << GetImportWatts () << "\twatts\n";
    std::cout << "Import Power:\t\t" << GetImportPower () << "\twatts\n";
    std::cout << "Real Import Power:\t" << ElectricWaterHeater::GetRealImportPower () <<
      ↪   "\twatts\n";
    std::cout << "Import Energy:\t\t" << GetImportEnergy () << "\twatt-hours\n";
}  // end Display
```

A.2.3 HeatPumpWaterHeater.cpp

```
#include <iostream>
#include <sstream>
#include <chrono>
#include "include/ElectricWaterHeater.h"
#include "include/logger.h"
#include "include/tsu.h"
#include <time.h>

// MACROS
#define DEBUG(x) std::cout << x << std::endl

ElectricWaterHeater::ElectricWaterHeater (
        std::map <std::string, std::string> configs) :
                DistributedEnergyResource (),
                sp_(configs["serial_port"]),
                  current_transducer_1_(stoul(configs["mcp_channel"])),
                  heartbeat_(stoul(configs["ucm_heartbeat"])) {

        // verivy serial port is available and connected to UCM
        if (!sp_.open ()) {
                Logger("ERROR", GetLogPath ())
                        << "failed to open serial port: " << strerror(errno);
                exit (1);
        } else {
                device_ptr_ = cea2045::DeviceFactory::createUCM(&sp_, &ucm_);
                device_ptr_->start ();
                device_ptr_->basicOutsideCommConnectionStatus(
                        cea2045::OutsideCommuncatonStatusCode::Found
                );
        }
        response_codes_ = device_ptr_->querySuportDataLinkMessages().get();
        response_codes_ = device_ptr_->queryMaxPayload().get();
        response_codes_ = device_ptr_->querySuportIntermediateMessages().get();
        response_codes_ = device_ptr_->intermediateGetDeviceInformation().get();

        SetLogPath (configs["log_path"]);
        std::cout << "Log Path: " << GetLogPath () << std::endl;
        //last_utc_ = 0;
        SetLogIncrement (stoul(configs["log_inc"]));
        Logger("INFO", GetLogPath ()) << "startup complete";
        Logger("DER_Data", GetLogPath ()) << "New EWH Initialized";
        SetRatedImportPower (4500);
        SetExportEnergy (0);
}  // end Constructor
```

```
ElectricWaterHeater::~ElectricWaterHeater () {
        delete device_ptr_;
}  // end Destructor

// ***Unintentionally doubled later on
// Begin critical peak event status
void ElectricWaterHeater::SetCriticalPeak () {
        device_ptr_->basicCriticalPeakEvent (0);
        opstate_ = 4;
        Logger("INFO", GetLogPath ()) << "Critical peak event command received";
}  /// end critical peak event status change

// Begin load up status
void ElectricWaterHeater::SetLoadUp () {
        opstate_ = 3;
        device_ptr_->basicLoadUp (0);
        Logger("INFO", GetLogPath ()) << "Load up command received";
}  /// end load up command

// Begin Grid Emergency status
void ElectricWaterHeater::SetGridEmergency () {
        opstate_ = 5;
        device_ptr_->basicGridEmergency (0);
        Logger("INFO", GetLogPath ()) << "Grid Emergency command received";
}  /// end grid emergency command

// Get Real Import Power
// - use current transducer and approximated voltage to get real power draw from
// - water heater
unsigned int ElectricWaterHeater::GetRealImportPower () {
        return current_transducer_1_.GetCurrent() * 240; // assuming Vrms = 240
}  // end Get Real Import Power

// Query Properties
// - update basic DER properties using EWH methods
void ElectricWaterHeater::QueryProperties () {
        device_ptr_->intermediateGetCommodity ();
        std::vector <CommodityData> commodities = ucm_.GetCommodityData ();
        for (const auto &commodity : commodities) {
                if (commodity.code == 0) {
                        SetImportPower (commodity.rate);
                } else if (commodity.code == 6) {
                        SetRatedImportEnergy (commodity.cumulative);
                } else if (commodity.code == 7) {
                        SetImportEnergy (commodity.cumulative);
                }
        }
        //Query operational state
        device_ptr_->basicQueryOperationalState();
        //unsigned int state = ucm_.GetOpState ();
        //std::cout << "Debugging: Opstate = " << state << std::endl;
}  // end Query Properties


// End Curtailment events
void ElectricWaterHeater::EndCurtailment () {
        device_ptr_->basicEndShed (0);
        std::cout << "Debugging: Ending previous curtailment for new command." <<
          ↪  std::endl;
} // end End Curtailment events

// Critical Peak event
void ElectricWaterHeater::CriticalPeakEvent () {
        device_ptr_->basicCriticalPeakEvent (0);
        std::cout << "Debugging: Sending Critical Peak Event command." << std::endl;
} // end Critical Peak event

/////////////////
// DER Overwrites
/////////////////

// Import Power
// - send a EndShed command through the UCM with a zero duration for undefined
// - amount of time
void ElectricWaterHeater::ImportPower () {
        device_ptr_->basicLoadUp (0);  // zero duration means as along as possible
        std::cout << "Debugging: Sending load up command." << std::endl;
        //device_ptr_->basicEndShed (0);
        //std::cout << "Debugging: Sending end shed command." << std::endl;
}  // end Import Power

// Export Power
// - DO NOTHING since water heaters cannot export power
void ElectricWaterHeater::ExportPower () {
        // just overwrite so DER Loop () doesn't do anything to properties
        SetExportEnergy (0);
}  // end Export Power

// Idle Loss
// - probably not the best name for this method, but when idle we want the
// - water heater to no be on so we send a Shed () command through the UCM with
// - a duration of 0 for undefined amount of time
void ElectricWaterHeater::IdleLoss () {
        device_ptr_->basicShed (0);
        std::cout << "Debugging: Sending shed command." << std::endl;
}
// Loop
void ElectricWaterHeater::Loop (float time_past) {
        time_t now = time(0);
        struct tm time = *localtime(&now);
        unsigned int sec = time.tm_sec;
        unsigned int min = time.tm_min;

        if (sec % 2 == 0){
                ElectricWaterHeater::QueryProperties ();
        }
        if (min % heartbeat_ == 0 && sec < 1){
                device_ptr_->basicOutsideCommConnectionStatus(
                        cea2045::OutsideCommuncatonStatusCode::Found
                );
                std::cout << "Debugging: Sending heartbeat." << std::endl;
        }

        // log every minuteish, there will be some double logs here and there
        if (sec == 0 && log_minute_ != min){
                ElectricWaterHeater::Log ();
                log_minute_ = min;
        }

        if (GetImportWatts () > 0 && GetImportPower () < 4500) {
                if (ucm_.GetOpState () != HEIGHTENED) {
                        ElectricWaterHeater::ImportPower ();
                }

        } else if (GetImportPower () > 0 && GetImportWatts () == 0 ) {
                if (ucm_.GetOpState () != GRID || ucm_.GetOpState () != CURTAILED) {
                        //ElectricWaterHeater::IdleLoss ();
                        ElectricWaterHeater::CriticalPeakEvent ();
                }

        }
}  // end Loop

// Log
// - log important physical attributes of DER on a frequency set by config file
void ElectricWaterHeater::Log () {
    unsigned int utc = time (0);
        Logger ("DER_Data", GetLogPath ())
            << GetRatedImportEnergy () << "\t"
            << ucm_.GetOpState () << "\t"
            << GetImportWatts () << "\t"
            << GetImportPower () << "\t"
            << ElectricWaterHeater::GetRealImportPower () << "\t"
            << GetImportEnergy () << "\t"
            << GetExportWatts () << "\t"
            << GetExportPower () << "\t"
            << GetExportEnergy () << "\t";
        last_utc_ = utc;
}  // end Log

// Display
// - print device properties to terminal
void ElectricWaterHeater::Display () {
    std::cout << "Rated Import Energy:\t" << GetRatedImportEnergy () << "\twatt-hours\n";
    std::cout << "Operational State:\t" << ucm_.GetOpState () << "\n";
    std::cout << "Import Control:\t\t" << GetImportWatts () << "\twatts\n";
    std::cout << "Import Power:\t\t" << GetImportPower () << "\twatts\n";
    std::cout << "Real Import Power:\t" << ElectricWaterHeater::GetRealImportPower () <<
      ↪   "\twatts\n";
    std::cout << "Import Energy:\t\t" << GetImportEnergy () << "\twatt-hours\n";
}  // end Display
```

A.2.4 WaterHeaterEmulator.cpp

```
#include "include/WaterHeaterEmulator.h"
#include <string>
#include <random>
#include <algorithm>
#include <vector>
#include <time.h>
#include <map>
#include "include/DistributedEnergyResource.h"
#include "include/tsu.h"
#include <iostream>
#include "include/logger.h"

WaterHeaterEmulator::WaterHeaterEmulator (tsu::config_map &config, unsigned int ID) :
        mains_temp_(stoul(config["EWH"]["mains_temp"])),
        temp_setpoint_(stoul(config["EWH"]["temp_setpoint"])),
        thermal_ramp_(stoul(config["EWH"]["thermal_ramp"])),
        ID_(ID) {

        //Determine household size
```

```
std::srand(time(NULL));
std::random_device rd;
std::mt19937 gen(rd());

float rn = float(rand())/RAND_MAX;
int size;
if (rn <= 0.133) {
        size = 1;
} else if (rn <= 0.396) {
        size = 2;
} else if (rn <= 0.792) {
        size = 3;
} else if (rn <= 0.957) {
        size = 4;
} else {
        size = 5;
}

//Read schedule and shuffle volumes column
if (size == 1) {
        schedule_ = tsu::FileToMatrix("../data/1bed.csv", ',' ,2);
} else if (size == 2) {
        schedule_ = tsu::FileToMatrix("../data/2bed.csv", ',' ,2);
} else if (size == 3) {
        schedule_ = tsu::FileToMatrix("../data/3bed.csv", ',' ,2);
} else if (size == 4) {
        schedule_ = tsu::FileToMatrix("../data/4bed.csv", ',' ,2);
} else if (size == 5) {
        schedule_ = tsu::FileToMatrix("../data/5bed.csv", ',' ,2);
}

//Normal distribution of event times and volumes
time_t utc = time(0);
tm now = *localtime(&utc);
std::string schedule_formatted_time;
time_t event_utc;
tm event_time;
char event_formatted_time [80];
float mean_time;
float time_std_dev = 60*30; //half hour standard deviation for times
float mean_event_vol;
float event_vol_std_dev;
float event_vol;

for (unsigned int i=0; i<schedule_.size(); i++) {
        schedule_formatted_time = schedule_[i][0];
        strptime(schedule_formatted_time.c_str(), "%T", &now);
        mean_time = mktime(&now);
        std::normal_distribution<double>
          ↪  time_distribution(mean_time,time_std_dev);
        event_utc = int(time_distribution(gen));
        event_time = *localtime(&event_utc);
        strftime(event_formatted_time,80,"%T",&event_time);
        schedule_[i].at(0) = event_formatted_time;

        mean_event_vol = stof(schedule_[i][1]);
        event_vol_std_dev = mean_event_vol * 0.3;
        std::normal_distribution<double>
          ↪  event_vol_distribution(mean_event_vol,event_vol_std_dev);
        event_vol = -1;
        while (event_vol < 0) {
                event_vol = event_vol_distribution(gen);
        }
        schedule_[i].at(1) = std::to_string(event_vol);
}
//Setting member properties for WHs
SetRatedImportPower(stoul(config["EWH"]["rated_import_power"]));
SetRatedImportEnergy(3630);
SetImportRamp(4500);
SetExportRamp(0);
last_utc_ = 0;
log_inc_ = stoul(config["DER"]["log_inc"]);
log_path_ = config["DER"]["log_path"];
SetIdleLosses (stoul(config["EWH"]["idle_losses"]));
WaterHeaterEmulator::SetBypassImportWatts (0);
WaterHeaterEmulator::SetBypassImportPower (0);
WaterHeaterEmulator::SetNormalImportPower (0);
SetImportPower (0);
SetImportWatts (0);
SetDeltaEnergy(0);

//Randomize starting energy state, with middle value as mean
float mean = 0.5;
float std_dev = 0.3;
float ratio;
std::normal_distribution<double> distribution(mean,std_dev);
ratio = 2;
while (ratio > 1 || ratio < 0) {
        ratio = distribution(gen);
}
SetImportEnergy(ratio*GetRatedImportEnergy ());
WaterHeaterEmulator::SetImportEnergyFloat (GetImportEnergy ());
```

```
        std::cout
                << "Emulator Initialized:\n"
                << "Household size:\t" << size << " bedrooms\n"
                << "Import Energy:\t" << GetImportEnergy () << " watt-hours\n"
                << "Usage Schedule:\n";
        for (unsigned int i=0; i<schedule_.size(); i++) {
                std::cout
                        << schedule_[i][0] << '\t'
                        << schedule_[i][1] << '\n';
        }

};
WaterHeaterEmulator::~WaterHeaterEmulator () {};

void WaterHeaterEmulator::Loop (float delta_time) {
        float import_energy = WaterHeaterEmulator::GetImportEnergyFloat ();
        if (import_energy > 2025) {
                WaterHeaterEmulator::SetBypassImportWatts (GetRatedImportPower ());
        } else if (import_energy < 1725 && GetBypassImportPower () != 0){
                WaterHeaterEmulator::SetBypassImportWatts (0);
                WaterHeaterEmulator::SetBypassImportPower (0);
        }

        if ((GetImportWatts () > 0 || WaterHeaterEmulator::GetBypassImportWatts () > 0) &&
          ↪  import_energy > 300) {
                WaterHeaterEmulator::ImportPower (delta_time);
        } else if (GetImportWatts () > 0 && GetImportPower () > 0 && import_energy > 0) {
                WaterHeaterEmulator::ImportPower (delta_time);
        } else {
                WaterHeaterEmulator::IdleLoss (delta_time);
        }
        WaterHeaterEmulator::Usage ();
        WaterHeaterEmulator::UsageLoss (delta_time);
        WaterHeaterEmulator::Log ();
};

void WaterHeaterEmulator::ImportPower (float delta_time) {
        float seconds = delta_time / 1000;
        float hours = seconds / (60*60);
        float import_energy = WaterHeaterEmulator::GetImportEnergyFloat ();
        float import_power = GetImportPower ();
        float import_watts = GetImportWatts ();
        float delta_energy;

        if (WaterHeaterEmulator::GetBypassImportWatts () > 0) {
                if (WaterHeaterEmulator::GetBypassImportPower () <
                  ↪  WaterHeaterEmulator::GetBypassImportWatts ()) {
                        WaterHeaterEmulator::SetBypassImportPower
                          ↪  (WaterHeaterEmulator::GetBypassImportWatts ());
                }
                delta_energy = WaterHeaterEmulator::GetBypassImportPower () * hours;
                WaterHeaterEmulator::SetImportEnergyFloat (import_energy - delta_energy);
                SetImportEnergy (import_energy - delta_energy);
        } else {
                if (import_power < import_watts && import_energy > 300) {
                        SetImportPower (GetRatedImportPower ());
                }
                delta_energy = GetImportPower () * hours;
                WaterHeaterEmulator::SetImportEnergyFloat (import_energy - delta_energy);
                SetImportEnergy (import_energy - delta_energy);
        }
        if (WaterHeaterEmulator::GetBypassImportPower () > 0) {
                SetImportPower (GetRatedImportPower ());
        }
};

void WaterHeaterEmulator::IdleLoss (float delta_time) {
        float seconds = delta_time / 1000;
        float hours = seconds / (60*60);
        float import_energy = WaterHeaterEmulator::GetImportEnergyFloat ();
        float energy_loss = 22000/(import_energy + 215) * hours;
        SetImportPower (0);
        WaterHeaterEmulator::SetImportEnergyFloat (import_energy + energy_loss);
        SetImportEnergy(import_energy + energy_loss);
};

void WaterHeaterEmulator::Usage () {
        time_t now = time(0);
        char time_formatted[100];
        tm now_local = *localtime(&now);
        strftime(time_formatted, sizeof(time_formatted), "%T", &now_local);
        float tank_temp;

        float import_energy;
        float delta_energy;
        float old_delta_energy = WaterHeaterEmulator::GetDeltaEnergy ();
        for (unsigned int i=0; i<schedule_.size(); i++) {
                if (time_formatted == schedule_[i][0]) {
                        import_energy = WaterHeaterEmulator::GetImportEnergyFloat ();
                        tank_temp = temp_setpoint_ - import_energy/100;
                        delta_energy = stof(schedule_[i][1])*(tank_temp -
                          ↪  mains_temp_)*2.44;
                        //SetImportEnergy(import_energy + delta_energy);
                        WaterHeaterEmulator::SetDeltaEnergy (old_delta_energy +
                         ↪   delta_energy);
                }
        }
};

void WaterHeaterEmulator::UsageLoss (float delta_time) {
        float seconds = delta_time / 1000;
        float hours = seconds / (60*60);
        float import_energy = WaterHeaterEmulator::GetImportEnergyFloat ();
        float delta_energy = WaterHeaterEmulator::GetDeltaEnergy ();

        if (delta_energy > thermal_ramp_ * hours) {
                WaterHeaterEmulator::SetImportEnergyFloat(import_energy + thermal_ramp_ *
                 ↪   hours);
                SetImportEnergy (import_energy + thermal_ramp_ * hours);
                WaterHeaterEmulator::SetDeltaEnergy (delta_energy - thermal_ramp_ *
                 ↪   hours);
        } else {
                WaterHeaterEmulator::SetImportEnergyFloat (import_energy + delta_energy);
                SetImportEnergy (import_energy + delta_energy);
                WaterHeaterEmulator::SetDeltaEnergy (0);
        }
};

void WaterHeaterEmulator::Log () {
        unsigned int utc = time (0);
        if ((utc % log_inc_ == 0) && (last_utc_ != utc)) {
                Logger ("Emulator_Data", log_path_)
                        << "Emulator ID#: " << ID_ << "\t"
                        << GetExportWatts () << "\t"
                        << GetExportPower () << "\t"
                        << GetExportEnergy () << "\t"
                        << GetImportWatts () << "\t"
                        << GetImportPower () << "\t"
                        << GetImportEnergy () << "\t"
                        << GetBypassImportPower () << "\t"
                        << WaterHeaterEmulator::GetImportEnergyFloat () << "\t";
                last_utc_ = utc;
        }
};

void WaterHeaterEmulator::Display () {
        std::cout
                << "Import Power:\t" << GetImportPower () << "\twatts\n"
                << "Bypass Power:\t" << WaterHeaterEmulator::GetBypassImportPower () <<
                 ↪   "\twatts\n"
                << "Import Control:\t" << GetImportWatts () << "\twatts\n"
                << "Import Energy:\t" << WaterHeaterEmulator::GetImportEnergyFloat () <<
                 ↪   "\twatt-hours\n"
                << std::endl;
};

float WaterHeaterEmulator::GetBypassImportPower () {
        return bypass_import_power_;
};

unsigned int WaterHeaterEmulator::GetBypassImportWatts () {
        return bypass_import_watts_;
};

void WaterHeaterEmulator::SetBypassImportPower (float power) {
        if (power > WaterHeaterEmulator::GetBypassImportWatts ()) {
                bypass_import_power_ = GetBypassImportWatts ();
        } else if (power <= 0) {
                bypass_import_power_ = 0;
        } else {
                bypass_import_power_ = power;
        }
};

void WaterHeaterEmulator::SetBypassImportWatts (float watts) {
        if (watts > GetRatedImportPower ()) {
                bypass_import_watts_ = GetRatedImportPower ();
        } else {
                bypass_import_watts_ = watts;
        }
};

void WaterHeaterEmulator::SetDeltaEnergy (float energy) {
        delta_energy_ = energy;
};

float WaterHeaterEmulator::GetDeltaEnergy () {
        return delta_energy_;
};

float WaterHeaterEmulator::GetNormalImportPower () {
        return normal_import_power_;
};

void WaterHeaterEmulator::SetNormalImportPower (float power) {
        normal_import_power_ = power;
};

float WaterHeaterEmulator::GetImportEnergyFloat () {
        return import_energy_float_;
};

void WaterHeaterEmulator::SetImportEnergyFloat (float energy) {
        if (energy > GetRatedImportEnergy ()) {
                import_energy_float_ = GetRatedImportEnergy ();
        } else if (energy <= 0) {
                import_energy_float_ = 0;
        } else {
                import_energy_float_ = energy;
        }
};
```

A.2.5 HeatPumpEmulator.cpp

```
#include "include/HeatPumpEmulator.h"
#include "include/WaterHeaterEmulator.h"
#include <string>
#include <random>
#include <algorithm>
#include <vector>
#include <time.h>
#include <map>
#include "include/DistributedEnergyResource.h"
#include "include/tsu.h"
#include <iostream>
#include "include/logger.h"

HeatPumpEmulator::HeatPumpEmulator (tsu::config_map &config, unsigned int ID) :
        WaterHeaterEmulator (config, ID) {
        SetBypassImportWatts (0);
        heat_pump_delay_timer_ = 0;
};

HeatPumpEmulator::~HeatPumpEmulator () {};

float HeatPumpEmulator::HeatPumpPower (float import_energy) {
        float power;
        power = 454 - import_energy*0.0414;
        return power;
}

void HeatPumpEmulator::Loop (float delta_time) {
        float import_energy = GetImportEnergyFloat ();
        unsigned int import_watts = GetImportWatts ();
        unsigned int bypass_watts = GetBypassImportWatts ();
        unsigned int element_power = 4500;

        if (import_energy > 1875) {
                SetBypassImportWatts (element_power + HeatPumpPower(import_energy));
        } else if (import_energy > 1800 && bypass_watts > element_power) {
                SetBypassImportWatts (element_power + HeatPumpPower(import_energy));
        } else if (import_energy > 1575) {
                SetBypassImportWatts (HeatPumpPower(import_energy));
        } else if (import_energy > 900 && bypass_watts > 0) {
                SetBypassImportWatts (HeatPumpPower(import_energy));
        } else {
                SetBypassImportWatts (0);
                SetBypassImportPower (0);
        }

        if ((import_watts > 0 || GetBypassImportWatts () > 0) && import_energy > 50) {
                HeatPumpEmulator::ImportPower (delta_time);
        }else if ((import_watts > 0 && GetImportPower () > 0) && import_energy > 0) {
                HeatPumpEmulator::ImportPower (delta_time);
        } else {
                IdleLoss (delta_time);
                heat_pump_delay_timer_ = 0;
        }
        Usage ();
        UsageLoss (delta_time);
        Log ();
};

void HeatPumpEmulator::ImportPower (float delta_time) {
        float seconds = delta_time / 1000;
        float hours = seconds / (60*60);
        float import_energy = GetImportEnergyFloat ();
        unsigned int import_watts = GetImportWatts();
        float delta_energy;
        unsigned int bypass_watts = GetBypassImportWatts ();
        unsigned int element_power = 4500;
        float heat_pump_output = 1083;
        heat_pump_delay_timer_ = heat_pump_delay_timer_ + seconds;

        if (bypass_watts > 0 && import_watts == 0) {
                if (bypass_watts < element_power) {
                        if (heat_pump_delay_timer_ > 60) {
                                SetBypassImportPower (bypass_watts);
                                delta_energy = heat_pump_output * hours;
                        } else {
                                SetBypassImportPower (0);
                                delta_energy = 0;
                        }
                } else {
                        if (heat_pump_delay_timer_ > 60) {
                                SetBypassImportPower (bypass_watts);
                                delta_energy = (element_power + heat_pump_output) * hours;
                        } else {
                                SetBypassImportPower (element_power);
                                delta_energy = element_power * hours;
                        }
                }
                SetImportPower(GetBypassImportPower ());
        } else if (import_watts > 0) {
                if (import_energy > 1575) {
                        if (heat_pump_delay_timer_ > 60) {
                                SetImportPower (element_power
                                        + HeatPumpEmulator::HeatPumpPower(import_energy));
                                delta_energy = (element_power + heat_pump_output)*hours;
                        } else {
                                SetImportPower (element_power);
                                delta_energy = element_power * hours;
                        }
                } else if (import_energy < 1575 && import_energy > 50) {
                        if (heat_pump_delay_timer_ > 60) {
                                SetImportPower (HeatPumpEmulator::HeatPumpPower
                                  ↪   (import_energy));
                                delta_energy = heat_pump_output*hours;
                        } else {
                        }
                } else if (import_energy < 50 && GetImportPower () > 0) {
                        SetImportPower (HeatPumpEmulator::HeatPumpPower (import_energy));
                        delta_energy = heat_pump_output*hours;
                }
        }
        SetImportEnergy (import_energy - delta_energy);
        SetImportEnergyFloat (import_energy - delta_energy);
};
```

A.2.6 Makefile

```
# This makefile is described by the following link
# https://hiltmon.com/blog/2015/09/28/the-simple-c-plus-plus-makefile-executable-edition/
CC := g++  #main compiler

# Project Directory
SRCDIR := src
BUILDDIR := obj
TARGETDIR := bin/debug
TARGET := $(TARGETDIR)/$(SRC)

# Auto get source files
SRCEXT := cpp
SOURCES := $(shell find $(SRCDIR) -type f -name *.$(SRCEXT))
OBJECTS := $(patsubst $(SRCDIR)/%,$(BUILDDIR)/%,$(SOURCES:.$(SRCEXT)=.o))

# General Requirments
ifeq ($(CPU), x86)
        CPUFLAGS := -m32
elseif ($(CPU), arm)
        CPUFLAGS :=
else
        CPUFLAGS :=
endif

CFLAGS := -Wall -pipe -std=c++11 -Wno-long-long -Wno-deprecated -g -DQCC_OS_LINUX
  ↪  -DQCC_OS_GROUP_POSIX -DQCC_DBG $(CPUFLAGS)
LIB := -lstdc++ -lpthread -lrt -lm
INC := -I src/include

# AllJoyn Requirments
CFLAGS += -DROUTER
LIB += -L$(AJ_LIB) -lalljoyn -lajrouter
INC += -I$(AJ_INC)

$(TARGET) : $(OBJECTS)
        @mkdir -p $(TARGETDIR)
        @echo "\n\tLinking $(TARGET)\n"; $(CC) $^ -o $(TARGET) $(LIB)

$(BUILDDIR)/%.o: $(SRCDIR)/%.$(SRCEXT)
        @mkdir -p $(BUILDDIR)
        @echo "\n\tCompiling $<...\n"; $(CC) $(CFLAGS) $(INC) -c -o $@ $<

clean:
        @echo "\n\tCleaning $(TARGET)\n"; $(RM) -r $(BUILDDIR) $(TARGET)

.PHONY: clean
```

A.2.7 test_drawcontroller.py

```python
#Flow meter constant is set for resistive WH!

from datetime import datetime
from numpy.random import normal
from numpy import zeros, savetxt, loadtxt
import random
import RPi.GPIO as GPIO
from time import time, sleep
from threading import Thread
import os
import csv

#Initialize GPIO
GPIO.setwarnings(False)
GPIO.setmode(GPIO.BCM)
FMPIN = 6     #flow meter GPIO pin
VPIN = 17     #valve GPIO pin
GPIO.setup(FMPIN, GPIO.IN, GPIO.PUD_UP) #setup flow meter pin as input
GPIO.setup(VPIN, GPIO.OUT, initial=GPIO.LOW)     #setup valve pin as output
GPIO.add_event_detect(FMPIN, GPIO.RISING)    #add rising edge detection

#Define function to draw water
def draw_water(targetVol):
    if targetVol <= 0:
        return()
    print ('Drawing %.2f gallon(s).' % targetVol)
    volume = 0
    numPulses = 0
    start_time = time()
    GPIO.output(VPIN, GPIO.HIGH)     #open valve
    while volume < targetVol:  #keep valve open until desired volume has passed
        if GPIO.event_detected(FMPIN):
            numPulses += 1     #Count pulses from flow meter
            volume = float(numPulses) / 476     #Calculate volume
        run_time = time()
        elapsed_time = run_time - start_time
        if elapsed_time > 180:
            print('Timeout Error.')
            break
    GPIO.output(VPIN, GPIO.LOW) #close valve
    print ('Volume drawn: %.2f gallon(s).' % volume)

thread_draw = Thread(target = draw_water, args = [0])

times = []
volumes = []
file = open('3bed.csv')
read = csv.reader(file)
for row in read:
    times.append(row[0])
    volumes.append(row[1])
file.close()

#Enter main program loop
while True:
    now = datetime.now()     #Update date/time
    filename = 'WH_Data_' + str(now.month) + '-' + str(now.day) + '-' + str(now.year) +
     ↪   '.csv'
    if not os.path.isfile(filename): #Check if a new day has begun
        data = open(filename, 'w')
        data.write('Time,Draw Amount\n')
        data.close

    #Draw water if there is an event at this minute
    timestr = datetime.strftime(now, "%H:%M:%S")
    drawVolume = 0
    for i in range(0,len(times)):
        if times[i] == timestr:
            drawVolume = volumes[i]

    if drawVolume != 0:
        if thread_draw.is_alive() == True:
            print('Debugging: Previous draw is still running. Waiting for draw to
             ↪   finish.\n')
            thread_draw.join()
        thread_draw = Thread(target = draw_water, args = [float(drawVolume)])
        thread_draw.start()
        data = open(filename, 'a')
        data.write(timestr + ',' + str(drawVolume)+'\n')
        data.close
    sleep(1)
```