SYSTEM ARCHITECTURE FOR DISTRIBUTED CONTROL SYSTEMS AND ELECTRICITY MARKET INFRASTRUCTURES

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAI'I AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

MECHANICAL ENGINEERING

AUGUST 2018

By

Holm Smidt

Thesis Committee:

Reza Ghorbani, Chairperson
Peter Berkelman
Lee Altenberg

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Reza Ghorbani, for his continued support over the past years and for working together on exciting and fun projects with the Renewable Energy Design Laboratory (REDLab) at Mānoa. It has been a very enjoyable and rewarding experience.
I would also like to thank my thesis committee, Drs. Peter Berkelman and Lee Altenberg, for being such a great committee and providing valuable critiques to my research.

The following individuals have mentored and inspired me over the past years and provided me with the means to develop into the multidisciplinary researcher that I am today. I am sincerely grateful for your support: Drs. Hervè Collin and Aaron Hanai from the Kapi'iolani CC STEM program, Dr. Reza Ghorbani, Matsu Thornton, and Dr. Volker Schwarzer from the Renewable Energy Design Laboratory (REDLab) at Mānoa, Dr. Kaveh Abhari from the STEMD^2 Research & Development Group and Center on Disability Studies, and Drs. Tung Bui and Thayanan Phuaphanthong from the Hawai'i International Conference on System Sciences and Shidler College of Business.

I feel privileged and humbled by the overall support of my parents, family, friends, peers, colleagues, and mentors that have gone out of their way in providing me with the resources and guidance that have allowed me to pursue my education and that have ultimately made this work possible.

ABSTRACT

Societal interests and environmental considerations continue to fuel the evolution of the modern energy grid. As we move towards a system with increased and decentralized integration of renewable energies, the dynamic and volatile nature of these renewables need to be considered. This calls for a paradigm shift not only in the planning and operation of our energy grid, but also in the way energy is consumed, marketed, and distributed. That is, mechanisms are needed to control grid demand when needed to ensure the important balance of demand and supply on the grid. Much theoretical work exists to address these challenges, including new control strategies focused on optimization of networked resources, strategies that focus on optimizing behaviors, and different game-theoretic market infrastructures that economically incentivize the use of novel demand response strategies. An agent-based testbed system has thus been architected to allow rapid development of smart agents that implement these control strategies and market infrastructures to test their interactions when deployed in a modern information and communication technology system. The behavior of various roles (e.g. system operator, demand response aggregators, and residential homes) can be programmed in the form of Python applications that communicate over the MQTT protocol. The use of a graph databases for cyber-physical energy system modeling is demonstrated as a configuration and management tool for running distributed co-simulations. A web application for monitoring and control for each role is backed by relational and graph databases. All system components can be adapted for different purposes and then deployed using Docker containers. Two use case scenarios were implemented and demonstrated the information system's ability to simulate dynamic pricing DR programs and emergency DR programs utilizing the multi-agent system. Testing distributed agents in the presented virtual Smart Grid testbed is shown to help develop these smart agents and validate their feasibility and efficacy at scale without having to physically implement the supporting sensor and control infrastructures; thus, the testbed system can bridge the implementation gap between theoretical models and actual systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

LIST OF ABBREVIATIONS

AI Articficial Intelligence

API Application Programming Interface

BESS Battery Energy Storage System

CPES Cyber-Physical Energy System

DB Database

DER Distributed Energy Resources

DLC Direct Load Control

DR Demand Response

DRA Demand Response Aggregators

DSM Demand Side Management

ETP Equivalent Thermal Parameter

GDB Graph Database

HEMS Home Energy Management System

HVAC Heating, Ventilation, and Air Conditioning

ICT Information and Communication Technologies

IEC International Electrotechnical Commission

IoT Internet-of-Things

ISO Independent System Operator

JSON JavaScript Object Notation

Pub/Sub Publish/Subscribe

PV Photovoltaic

RAMI4.0 Reference Architectural Model Industrie4.0

RES Renewable Energy Resources

RTP Real-Time Pricing

SG Smart Grid

TOU Time-of-Use

VPS Virtual Private Server

CHAPTER 1
INTRODUCTION

1.1 Problem Description and Scope of the Research

"Data is the new oil"

The modern electrical grid is evolving and although more commonly cited in the context of today's data economy, Clive Humby's statement that "data is the new oil"[1] applies well in the grid modernization context as centralized fossil fuel generation is making way for increased integration of renewable and distributed energy resources. However, the value of these distributed and volatile energy sources only lies in their networked coordination and control.

Although fossil fuels, and especially oil, have provided much of the economic growth and wealth in the last century, their finite availability and concerning environmental implications have led to the call for a paradigm shift in the way we generate, distribute, and consume energy. Local, national and global programs, such as the Paris Agreement, evidence these developments. But as we begin to take action to integrate greater proportions of renewable energy resources (RES) into the electrical grid and reduce our fossil fuel dependence, we begin to see several problems arise. Many grids are still built with the century-old architecture in mind. When we consider the total demand in terms of power and energy supplied by the modern electrical grid, we are faced with the fact that the amount of RES currently integrated into the grid are far below the capacity to deliver the amount of power and energy required by today's standards. We add the additional complication that RES are of course restricted to the whims of nature in general and will be able to produce energy relative to the current environmental conditions. This is in stark contrast to the conventional generator which is able to be adjusted at the turn of a steam governor or other mechanism.

Simultaneously, digitalization is transforming processes in every aspect of our lives, both at the personal and professional level. Marketing terms such as the Internet-of-Things (IoT) and Big Data have emerged and artificial intelligence (AI) is said to be the new electricity[2] that will transform industries in a way that electricity did about a century ago. The state of affairs of what is technologically and economically feasible at large scale has thus changed

[1] https://tinyurl.com/hs-ms-data-oil

[2] https://tinyurl.com/hs-ms-ai-electricity

for the better. In the past, real-time access to sensors and controls for equipment has been prohibitively expensive. Historical device data from aggregates of devices were generally not available and optimization was often on a one to one basis. With the advent of affordable network connected devices and controllers, i.e. IoT networks, and large capacity for data storage and processing, i.e. Big Data and AI, we are now able to implement more powerful control strategies for coordinated demand side management (DSM) in the modern electrical grid.

Hence, data are indeed replacing fossil fuels as the most important resource in the modern energy grid. But like oil, data in its raw format alone will not modernize the grid. It takes the proper information system to turn data into information, and information into actionable insights for operation and control. As Humby states in the latter part of his statement: "It's valuable, but if unrefined it cannot really be used. It has to be changed into gas, plastic, chemicals, etc. to create a valuable entity that drives profitable activity; so data must be broken down, analyzed for it to have value"[3].

Smart Grid

The term Smart Grid (SG) refers to the concept of modernizing the electrical grid to achieve more flexible and "smarter" use of available resources, especially RES, and considers all components of the electrical grid between any point of generation and any point of consumption [5, 6, 7]. Benefits of the SG include incentives for energy efficiency and reduced carbon dioxide (and other harmful emissions), integration of renewable energies, and lower costs for both utilities and consumers [8].

While the SG touches nearly every person—end users (consumers), electric-service retailers, distribution-service providers, wholesale market operators, balancing authorities, products and services suppliers, and local, state, and federal energy policymakers to name some [6]—the consumer plays an integral role in the operation of the SG system as they can adjust their purchasing pattern and behavior (i.e. demand) based on available information and incentives [5, 9]. To allow for improved DSM, SG technologies, especially advanced information and communication technology (ICT) systems are needed to provide near real-time feedback to operationalize better decision making, and to inform and engage consumers [8, 9].

[3] `https://tinyurl.com/hs-ms-data-oil`

Demand Response

Allowing consumers to participate in the optimization and operation of the energy grid through the provision of intelligent monitoring, control, communication, and self-healing technologies is considered integral by the International Electrotechnical Commission (IEC) [5]. The term demand response (DR) describes the types and levels of consumer participation in the grid. Chiu et al. [1] further define DR as "a dynamic change in electricity usage coordinated with power system or market conditions. The response or change in usage is facilitated through DR programs designed to coordinate electricity use with electric system needs. ...DR is achieved through application of a variety of DR resource types, which include distributed generation, dispatchable load, storage, and other resources capable of supporting a net change in grid-supplied power."

As detailed in [1, 9], DR programs can be classified by the type of interaction, type of incentive, customer classes, and objectives. Considering DR program objectives, we can categorize DR programs as follows:

a) Price Response (e.g. Dynamic Pricing):
   Variable price structures incentivize altered electricity consumption during periods of extreme market prices. Prices may be at pre-set times (e.g. time-of-use (TOU) pricing) or dynamically during the day (e.g. real-time pricing (RTP)). Higher prices typically characterize peak times and low prices off-peak times. Variable price structures may also lead to negative rates to encourage energy consumption when needed.
b) Reliability Response (e.g. Emergency DR Programs):
   Shedding loads upon request, rather than starting a generator, can be a viable mean to prevent blackouts, but requires direct load control (DLC) over appliances and equipment by the administrator. Event-based DR programs are hence only used when needed, such as the sudden loss of generation for example. Consumers would typically enroll in this type of program to receive compensation for the service they provide.
c) Both Price and Reliability Response (e.g. Ancillary Services):
   Ancillary services are reserves than can be procured through bids in the wholesale electricity market. Consumers would place demand reduction bids (consisting of capacity and price) to the utility or aggregator. The type of ancillary service (e.g. operating reserve) would further determine the specifics of the time, bidding, and aggregation constraints for this type of response.

Table 1.1: Domain of the domains of SG to support DR business models [1].

Domain Name	Domain Description
Customers	Any entity that takes gas and/or electric service for its own consumption. The consumers of electric power. Customers include small to large size C&I customers and residential customers.
Markets	Power market is a system for effecting the purchase and sale of electricity, using supply and demand to set the price.
Service Providers	An entity that provides electric services to a retail or end-use customer.
Operations	The management of generation, market, transmission, distribution and usage of the electric power.
Generation	The production of bulk electric power for industrial, residential, and rural use. It also includes power storage and DER.
Transmission	Electric power transmission is the bulk transfer of electrical energy, a process in the delivery of electricity to consumers.
Distribution	Electricity distribution is the final stage in the delivery of electricity to end users. A distribution system's network carries electricity from the transmission system and delivers it to consumers.
Micro-grid	The local grid for distributed energy resources management and delivery.

Electricity Markets

While wholesale electricity market prices change periodically—with frequencies depending on the type of market (e.g. 10-minute intervals)—, invariant electricity retail rates remain today's status quo [10]. Factors for the tremendous changes in the marginal cost of electricity are "(a) the demand for electricity varies considerably; (b) it is uneconomical to store electricity in most applications; and (c) the optimal mix of generating capacity to balance supply and demand at all hours given (a) and (b) includes a combination of base load capacity with high construction costs and low marginal operating costs, intermediate capacity with lower construction costs but higher marginal operating costs, and peaking capacity with the lowest construction costs and the highest marginal costs" [11]. Consequently, consumers will use too much when prices are higher than retail rates and too little when marginal prices are lower than retail rates. Time-invariant pricing thus not only contradicts basic economic principles of demand and supply, but distorted consumption can ultimately lead to distorted investment that resists rather than promotes the modernization of grid infrastructures. The

idea of time-marginal costs and varying prices is not novel and was already applied by public utilities for wholesale markets by 1970; Kahn (1970) however critiqued that "unfortunately, the principle has usually been badly applied, in several important ways. First, if the demand charge were correctly to reflect peak responsibility, it would impose on each customer a share of capacity costs equivalent to his share of total purchases at the time coinciding with the system's peak... Instead, the typical two-part tariff bases that rate on each customer's own peak consumption over some measured time period" [12]. Already identified as a problem then, the lack and cost of advanced metering infrastructures and inconvenient complexity for utility and consumers were considered as some of the main factors that have historically hindered the adoption of Kahn's work on time-varying retail rates [11]. Both SG technologies, especially advanced metering infrastructures, and theory have since emerged to operationalize time-varying rate structures [10, 11].

With the existence of DR markets for ancillary services where a pool of consumers or DR aggregators places bids for providing services, bidders need to choose "good" bidding strategies to make profits and market makers need to aggregate and select bids to optimally provide ancillary services to the grid. Game-theoretic market models can be applied to study how overall efficacy is related to market and competition models, how aggregators should choose their strategies, and how market makers should operate their markets (see [13] and references therein).

Simulation Tools

Industry, research, and education related to electric power grid systems heavily rely on simulations to test theoretical models and control schemes. Simulation results are integral in providing insights on the efficacy of tested models and a better understanding of the overall system behavior. The cyber-physical nature of the evolving grid with its high degree of networked and distributed generation, decentralized control, and decentralized agents in the grid and on markets, presents unprecedented challenges to simulating energy systems [**?**]. Palensky et al. view the following four categories as integral when considering the energy system of the future: a) physical world: continuous models for generation, distribution, consumption and infrastructure, b) information technology: discrete models for controllers and communication infrastructure, c) roles and individual behavior: game theory models for agents acting on behalf of a customer, and d) aggregated and stochastic elements: statistical models for environmental influences such as weather [**?**]. A breadth of simulation tools and platforms for simulating aspects of these four categories is presented by [**?**]. Some tools (e.g.

OpenDSS [?] and MATPOWER [?]) are designed for modeling physical energy systems and power distribution, whereas other tools are more focused on modeling communications (e.g. OPNET Modeler [?]), or building energy consumption (e.g. EnergyPlus [?]). GridLAB-D [?] presents an example of a comprehensive hybrid simulation framework for future energy systems.

Commissioned by the US Department of Energy's Office of Electricity and developed by the Pacific Northwest National Laboratory, GridLAB-D was designed as an agent-based simulation framework that can address a wide range of SG problems [14]. The agent-based simulation approach allows GridLAB-D to remain modular and to let modelers determine which agent-based characteristics to implement in any given module. Standard modules include the power flow module, generator modules, building modules using equivalent thermal parameter (ETP) methods, and a retail market. The simulation framework's efficacy has been shown in a number of use cases (see [14] and references therein). [15], for example, analyzes DR programs at the distribution level by simulating DLC and RTP using active heating, ventilation, and air-conditioning (HVAC) controllers that respond to end-user setpoints and a price signal. As an open-source platform, GridLAB-D can also be integrated as part of other simulation frameworks, as demonstrated in the GridSpice framework.

GridSpice is an open-source simulation framework for the modeling of electric power grid networks that include aspects of generation, transmission, distribution, and markets [16]. The simulation framework integrates with the existing GridLAB-D and MATPOWER tools for grid modeling, analysis, and power flow optimization, and uses simulation clusters with supervisor and worker nodes that execute the simulation tasks on scalable cloud computing platforms. GridSpice's utility was demonstrated using simulations for integration and placement of PV, volt/var control and demand response to name a few. GridSpice's architectural design aims at addressing other platform's limitations in the co-simulation of transmission and distribution systems, especially with regards to scale and modeling capabilities [16].

Simulation environments bridging multiple domains are typically characterized by a layered architecture where existing domain simulators are parallelized and managed by a co-simulation interface, as shown in [16, ?, ?, ?]. The reader may refer to [?] for an additional overview of integrated power/network simulators. The Virtual Grid Integration Laboratory co-simulation platform additionally considers the integration of real hardware. A hardware-in-the-loop co-simulation included the control of the ventilation system of a 12-story dormitory in Denmark equipped with necessary sensors and controls. [17] further describes a simulation testbed for IoT hardware-in-the-loop that leverages PSIM, an existing power

modeling software [**?**], by connecting existing tools to network-connected devices in order to test advanced optimization algorithms against simulated grid events with both real and simulated device nodes.

The breadth of established simulation platforms in the energy system, communication, and building modeling domains—and multi-domain combinations thereof—show the importance and value of these platforms to education, research, grid operators and policy makers. Once simulated however, proposed DR programs for residential customers remain challenging to implement due to temporal and monetary constraints in the roll-out and administration of distributed sensors and control systems, even in proposed hardware-in-the-loop simulations, such as [17].

Scope of the Research and Contributions

The consumer role is fundamental to the SG and is considered to have the potential to overcome inherent challenges associated with managing high-variability distributed energy resources (DERs). DR programs are designed to provide a framework that grid operators and policy makers can use to evaluate the utility of consumer participation in an electrical grid. More advanced multi-domain and agent-based simulation platforms allow researchers to test proposed DR programs and control mechanisms; yet, the actual implementation remains a challenge. Testbed systems are needed to bridge pure simulations and full-scale distributed deployments on real hardware systems.

This work provides an agent-based testbed system designed to assist in the development of smart agents for residential DSM and market participation. The presented multi-agent system contributes to existing work by providing a platform for users to implement algorithms that have already been proven to be effective in published and peer-reviewed simulations. Instead of having time and monetary resources to go through the initial development and deployment of distributed systems for real hardware, virtual smart agents can be containerized, deployed, and tested more quickly in a distributed information system under consideration of communication protocols and control strategies that are implementable on devices with limited computing resources (IoT devices). The proposed use of a graph-theoretic modeling approach for cyber-physical energy systems (CPES) provides an easy-to-use configuration and asset management tool for distributed agents so that the implemented agent-based model applications can remain versatile and reusable for various use case applications.

With a strong focus on the residential demand side, the user needs to define a virtual SG model, define individual agent types, their behavior, and the information flow between

them, and then run a real-time co-simulation where individual actions are communicated, logged, and visualized in a web application. Utilization of this simulation tool can contribute to the understanding of the benefits and drawbacks, scalability, and security concerns of DR programs and markets when implemented on IoT systems and thus provide researchers, grid operators, and policy makers with actionable insights to adapt tested programs and markets.

1.2 Thesis Outline

This thesis presents the architectural design of the proposed testbed system as well as its implementation on two use case applications for system evaluation purposes: time-varying retail rates and DLC in emergency DR programs. This introduction and remaining chapters, especially Ch. 4, incorporate materials from papers [18, 19] by the author, coauthored by Matsu Thornton and Reza Ghorbani.

Ch. 2 provides the general scheme of the testbed platform and outlines its major components and their respective responsibilities. A process model is provided to frame the use of the testbed system. Process specifics are then explored in Ch. 3 using the two use case applications. Ch. 4 discusses the design and implementation of the multi-layered system. Components of each layer are broken down by their functionalities and implementation with respect to the tested use case applications. CPES models and use case simulation data are analyzed and critically evaluated in Ch. 5 and 6 respectively, followed by a summary and outlook for future work to conclude this thesis.

CHAPTER 2
SYSTEM ARCHITECTURE AND PROCESS MODEL

2.1 Architectural Overview

The objective is to provide a testbed that helps the implementation and testing of control mechanisms and market policies for the SG under consideration of communication protocols and data management for distributed resource constrained IoT devices. The proposed architecture describes the technical framework of an information system capable of

a) modeling modern electrical grid domains (agents) as CPES;
b) communicating information between distributed agents;
c) simulation level data storage and management;
d) co-simulating distributed agents participating in a virtual grid.

The present discussion is limited to modeling residential customer participation and demand management such that only the first four domains (customer, markets, service provider, and operations) in Table 1.1 are considered. In reference to NIST's Smart Grid Conceptual Model (Fig. 2.1), one notices that this limitation simplifies the model as the electrical interface, and thus physical limitations and constraints, between the bottom four domains is not part of the simulation. Aforementioned simulation tools are far superior in this regard and should therefore be integrated into this platform in the future if the consideration of these domains is desired.

To co-simulate the four domains, a multi-agent system is designed where each domain has its own set of behaviors and policies. The complexity of each agent model (domain) depends on the use case of interest; in some cases, domains may also be represented passively through other agents, or excluded altogether. Fig. 2.2 summarizes the modeling of components of the multi-agent system; that is, virtual agents are "placed" in a virtual grid and behave based on provided principles. Placing the virtual agents in the same virtual grid environment allows the user to study agglomerate effects of being in the same environment. Consider the following scenario for illustration.

Time-varying electricity rate structures are implemented with fixed TOU rates for three intervals throughout the day. A residential home is equipped with a home energy management system (HEMS) to monitor the home's energy usage, PV generation, and battery

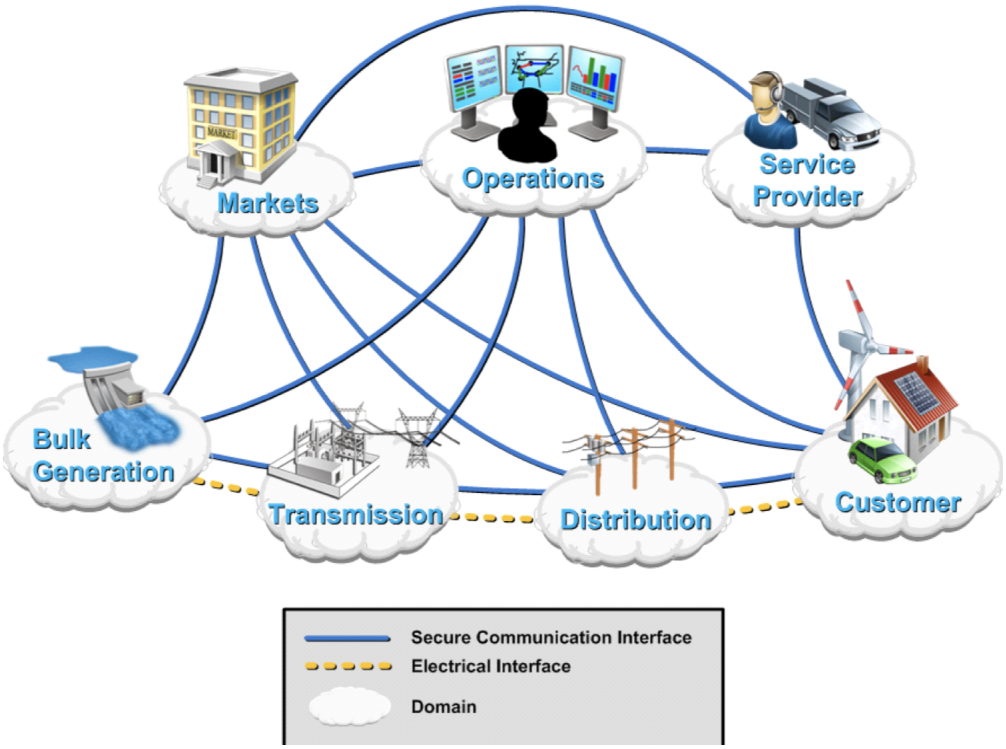


Figure 2.1: Smart Grid Conceptual Model [2]

energy storage system (BESS), and to control smart appliances and the BESS to optimize the home's overall consumption with respect to purchase, self-supply and export of energy to the aggregator. An optimization algorithm is then designed and employed on the HEMS to take historical patterns, weather predictions, and price predictions to reduce overall costs. In a single-node simulation with prescribed inputs, the simulation may show great efficacy for optimally low energy costs for the house. Running the same simulation in a multi-agent system however, one can implement more sophisticated behaviors for the service provider (aggregator) that puts purchase and export bids of each house into perspective of every other house and thus restricts a house in its optimization possibilities.

The key aspect of this illustration is that every agent in a multi-agent system affects the overall state of the system and that localized optimization schemes need to respond to externalities. As stated in [12], "everyone's economic activities indirectly affect the welfare of others—effects that do not enter into his own decisions." To achieve the co-simulation of agents and information sharing between agents whenever appropriate—i.e. one would share one's energy usage with the utility or service providers but not one's neighbors—, a layered architectural approach is taken, as shown in Fig. 2.3. Depicted on the left,

Figure 2.2: Overview of the virtual elements and their governing principles in the simulation.

a communication layer sits on top of the agent layer to facilitate communication among agents. Above the communication layer sits the administration and data layer overseeing the simulation and providing web-based administration tools such as configuration, monitoring and control. The multi-layered approach with agents in the lower layer was modeled after reference architectures for the SG, such as [20, 21].

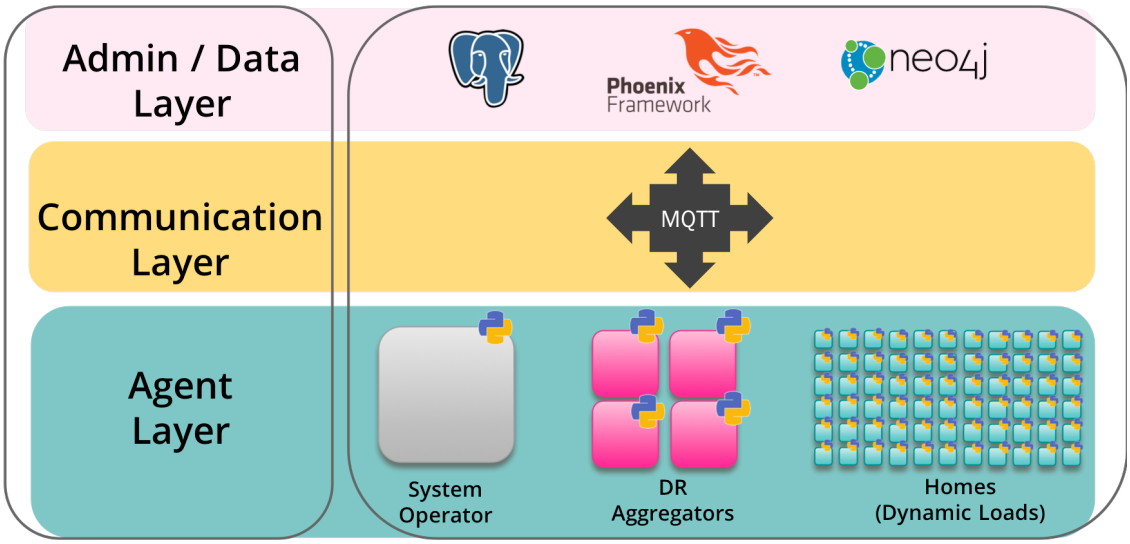


Figure 2.3: Overview of the three-layered system architecture with each layers' respective components. Icons illustrate the type of language or software used in each respective layer.

Each component is packaged and deployed as a Docker container. Docker containers are lightweight abstractions at the application layer that isolate processes from the host system [3]. Like virtual machines, Docker containers are represented as binary artifacts that can be run on any host with the Docker host environment installed. Unlike virtual machines however, Docker containers only come with minimal resources and do not entail a full operating systems. Docker containers are merely services that help make up applications

that can easily be shared and run on different systems and thus do not fall under the virtualization technology category. This differentiates the two technologies and shows that one is not a substitute of the other; they are separate concepts for separate purposes (see Fig. 2.4). In fact, the two technologies are often used in combination (e.g. when the Docker host environment is installed on a cloud VPS). Further, since Docker containers can be shared and scaled across platforms (e.g. server instances), single micro-service applications often comprise of a larger number of small Docker containers (services) that when deployed together, comprise a single application. That is, the system conceptualized in Fig. 2.3 can be considered as a single micro-service application.

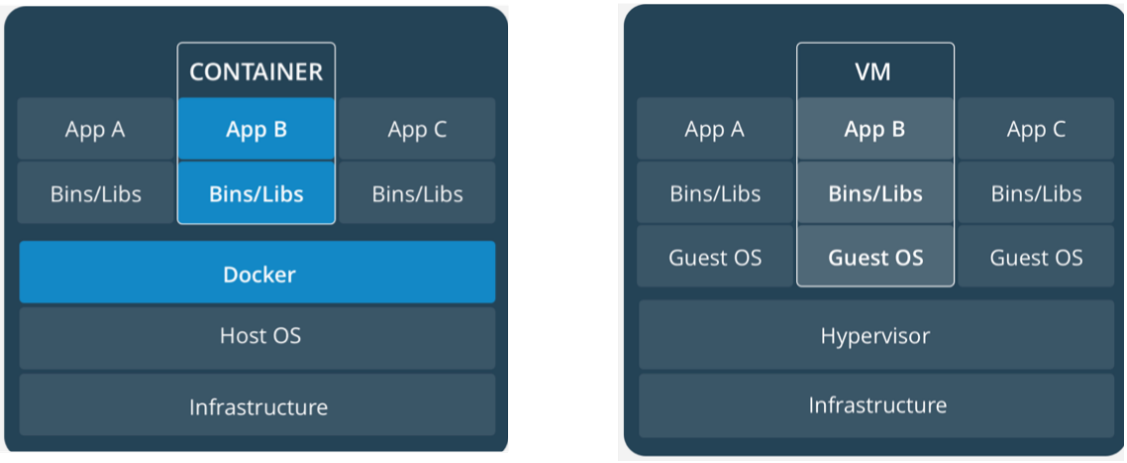


Figure 2.4: Conceptual comparison of Docker containers and virtual machines showing that containers are isolated processes that do not contain an operating system. [3]

2.2 Simulation Process Model

A standardized process model provides a streamlined workflow for designing, developing, and deploying different use case simulations. Each simulation breaks into its pre-, peri-, and post-simulation stages, each of which is modeled by a set of processes that the user has to walk-through. The pre-simulation processes are the most generalizable and are shown in Fig. 2.5. The pre-simulation stage consists of a design, setup, deployment, and start process, each consisting in turn of multiple subprocesses. The use case scenarios in Ch. 3 demonstrate the utilization of this process modeling scheme in a practical context.

The simulation design (Step 1) becomes the foundation of the entire simulation and is thus of foremost importance. This pre-simulation process involves the clear definition of the

simulation objectives as well as the resulting agent models. At the end of this step, the user should be able to design graphics analogous to those in Fig. 2.1 and Fig. 2.2 detailing the information flow between domains as well as the grid model, agent model, and their governing principles respectively. In terms of the three-layered software architecture, this corresponds to the agent layer design.

Pre-simulation process 2 entails the majority of the remaining workload before starting the application in process 4. We begin by generating configuration files which capture all of the important parameters and which can later be used as the seed for modeling the CPES. Next, each agent that follows a different behavior needs to be implemented as a containerized application with appropriate application program interfaces (APIs). Once an application has been developed in this step, it can be reused and modified at a later time in a similar use case. Consider the practical example of a residential home agent in a system with time-varying electricity rates. Assuming that an agent model from a prior simulation exists that entails the reporting of energy given a predefined load profile (see 3.1), a second simulation is to be implemented with a bottom-up simulation approach for appliance usage (see 3.2). Consequently, the user only needs to modify the existing model with the added functionality of modeling individual appliances in a household.

Besides the agent models, some modifications to the database (DB) design, web applications, and communication layer may be necessary to address all aspects of the system design. Creating Docker images is then the last subprocess before transitioning into the deployment stage, which simply requires the user to deploy containers on server equipment with the Docker host system installed. The web interface provides the functionalities needed for pre-simulation process 4, which entails the upload of DB seed files and start of the simulation.

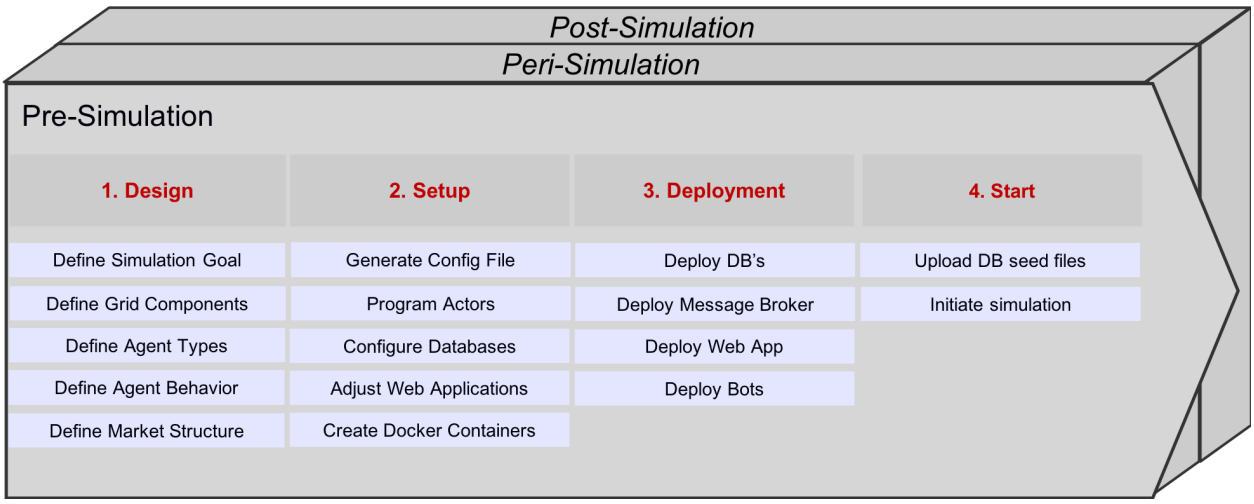


Figure 2.5: Pre-simulation process model showing the steps from the ideation to start of the simulation.

CHAPTER 3
USE CASE APPLICATIONS

3.1 Use Case A: Smart Metering with Time-varying Pricing

Dynamic pricing DR programs require the use of advanced metering infrastructures (smart meters) to correctly charge customers for their purchases coinciding with the system's electricity rate at the time of purchase. The cost associated with such infrastructure has historically prohibited the implementation of these strategies [11] but has since been overcome. By the end of 2016, 47% of 150 million electricity customers in the U.S. had smart meters installed [22]. Given the time and resources needed to deal with the complexity of manually analyzing day ahead price predictions and plan one's house's energy usage for the next day to save costs under time-varying prices, it seems uneconomic for the average residential consumer to do so. HEMS should rather be used to schedule appliances in a way that automatically adjusts usage based on time-varying prices. To show the system's ability to co-simulate such optimizations, the simplified case of smart metering and time-varying prices is considered in this use case scenario. In doing so, this use case further allows the testing of the overall three-layered architecture approach as well as the CPES modeling approach. Applying the design process (see Fig. 2.5), the following concept emerges.

P1.1 Simulation Goals: The objective is to test the testbed's ability to support dynamic pricing DR programs and test layers and components of its three-layered system architecture.

P1.2 Grid Components: The virtual grid is comprised of residential customers located across multiple zones (geographic regions) in a distribution grid.

P1.3 Agent Types: The role of the system operator, service provider, and residential home is considered, where only the operator and the homes play an active role; the service provider is only present for modeling purposes.

P1.4 Agent Behaviors: The system operator sets the market price for residential rates in 1-hour intervals using a predefined rate schedule. HEMSs, representing residential homes,

publish their predefined energy usage at 5-minute intervals.

P1.5 Market Structure. Time-varying prices are published by the system operator and the residential customer is billed based on rates coinciding with the time of energy usage.

Fig. 3.1 summarizes the subprocesses of the simulation design. The predefined rate schedule for the system operator is based on historical 1-hour energy prices from the PJM energy market between May 1 and May 14, 2018. The residential load data were taken from a residential house on Oahu, Hawaii, that was monitored by the REDLab Manoa for 14 days in April 2018 using a Fluke 1735 power logger.

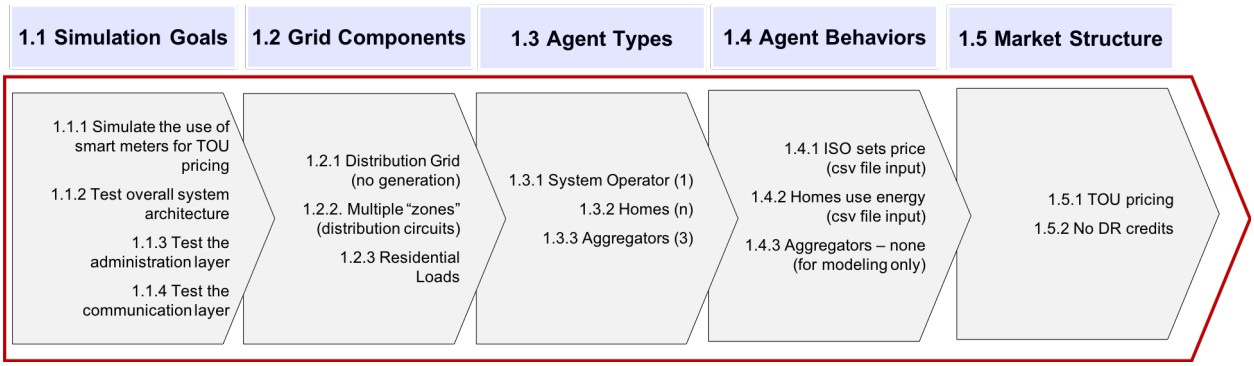


Figure 3.1: Overview of the subprocesses of the design process for the pre-simulation stage of the first test case scenario.

3.2 Use Case B: Providing Emergency DR Services

Emergency DR programs promote the use of DR strategies under special conditions, such as the sudden loss of generation. In such events, the ISO may fall back to the available immediate DR resources, its regulating reserves, by directly requesting an immediate load reduction on the grid (or in a certain zone), which is the focus of this simulation example. The simulation has a similar design to that of Use Case A but with different behaviors and market structure, as shown in the design below.

P1.1 Simulation Goals: The objective is to test the testbed's ability to support emergency DR programs that use DLC and to test the administrator's interaction with the virtual grid through the web interface.

P1.2 Grid Components: The virtual grid is comprised of residential customers located across multiple zones (geographic regions) in a distribution grid.

P1.3 Agent Types: The role of the system operator, service provider, and residential home is considered, where only the operator and the homes play an active role; the service provider is only present for modeling purposes.

P1.4 Agent Behaviors: The system operator sets the market price for residential rates in 1-hour intervals using a predefined rate schedule (same as in Sect. 3.1) and outputs direct control signals to available DR assets in the case of an immediate need for load curtailment. HEMSs, representing residential homes, model the use of appliances (controllable and non-controllable), report energy usage and DR availability, and adjust their usage when direct control events are received.

P1.5 Market Structure. Time-varying prices are published by the system operator and the residential customer is billed based on rates coinciding with the time of use. Residential customers participating in DR events receive a credit of 5x the load shed evaluated at the coinciding price.

Reiterating the implications of the defined simulation goal in terms of information system capabilities, the simulation tests:

a) a granular, bottom-up simulation approach of residential homes;

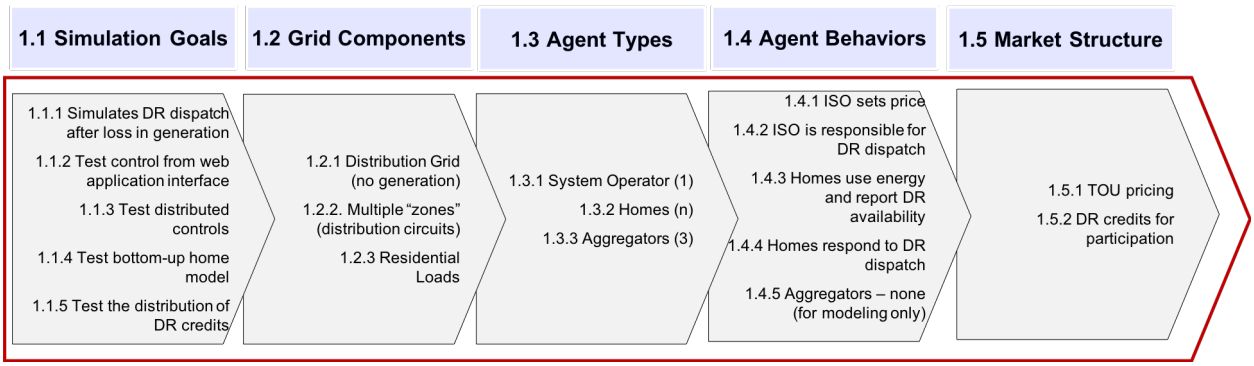


Figure 3.2: Overview of the subprocesses of the design process applied to the second use case scenario.

b) the direct interaction with the virtual environment from the control interface in the form of direct control mechanism;
c) tracking the distribution of DR credits for residential homes.

CHAPTER 4
SYSTEM DESIGN AND IMPLEMENTATION

4.1 Implementation Synopsis

Administration Layer. The administration layer provides the necessary functionalities to manage simulations from start to end. This entails the use of a graph-theoretic modeling approach to model the virtual electric grid and its connected components with their functionalities, a centralized data store to capture all events during the simulation, and a web application for administration, monitoring, and control. The driving design principle for the administration layer was to design an information system that could be used to manage actual distributed sensor and control systems.

Communication Layer. The communication layer enables the communication among agents themselves as well as the agents and the administration layer. The communication layer employs the Publish/Subscribe (Pub/Sub) communication scheme; the MQTT protocol is therefore used by the virtual resource constrained agents and components of the administration layer.

Agent Layer The agent layer describes the nodes of the multi-agent system. Each agent type implements a behavior based on the simulation design, which then translates into a set of functionalities and rationales that can be implemented as dockerized Python applications. Each node container is thus a micro-service in the simulation environment with an API for MQTT communications.

4.2 Administration Layer Implementation

The administration layer's implementation is based on the following system requirements:

a) CPES modeling for asset management in distributed systems;
b) centralized data storage for event logging and web applications;
c) administration, monitoring and controls through web applications.

4.2.1 Cyber-Physical Energy System Modeling

Graph databases (GDBs) are grounded in graph theory, a proven tool for modeling complex, highly interconnected systems, e.g. computer systems, biological systems, and social network systems, that uses graph structures such as nodes, edges, and labels. Graph-theoretic approaches to system modeling allow emphasis on component interactions and interconnections rather than device level logic. One particularly interesting and fitting application for this approach is in electrical grid modeling, where physical power lines are viewed as the connections (edges) between grid components (nodes). These applications range from pure topological modeling (see [23, 24, 25]) to extended topological methods that integrate power flow considerations to conventional network science modeling techniques (e.g. [26]) for grid robustness analysis (e.g. [27, 28, 29]) and system design (e.g. [30, 31, 32]).
In this work, a graph-theoretic approach is taken to model SG domains with their assets and intra- and inter-domain relationships in a distribution grid to provide an administration layer that allows the simulation of multi-agent systems. That is, a CPES model is created using a graph database that captures administrative information of each component.

Graph modeling with neo4j

The CPES model describes assets (agents or their representative systems) using neo4j, a graph database that implements the property graph model [**?**]. As such, the graph consists of nodes and relationships. Nodes are basic entities that can exist in and of themselves. Relationships connect exactly two nodes, the source node and the target node. Tokens are nonempty strings of Unicode characters; nodes can have sets of labels (one or more tokens) and relationships have exactly one relationship type (exactly one token). Both, nodes and relationships can have properties, which are key-value pairs (one or more tokens). Graph traversal describes how the graph database is being traveled, or in other words the navigation through a graph to find paths. Fig. 4.1 depicts an illustration of three nodes connected by

three relationships. Each node has a different label (i.e. DRA, HEMS, Zone) and different sets of properties. Unlike relational databases, entities of the same type (nodes with the

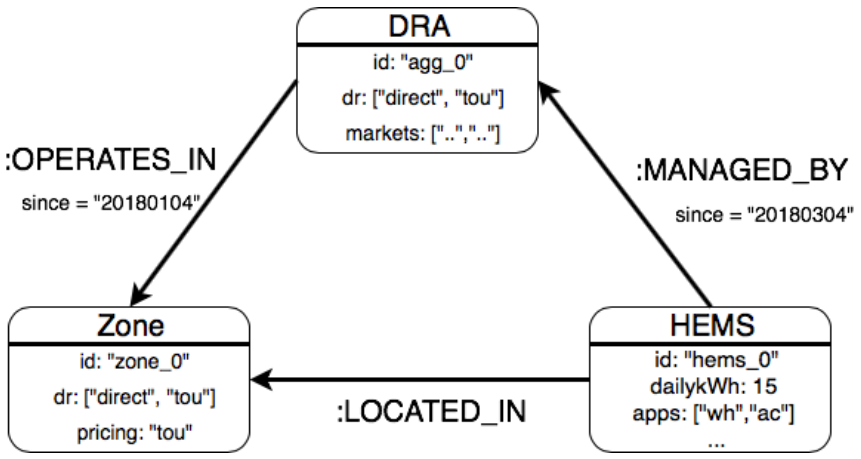


Figure 4.1: Sample graph illustrating concepts of property graph models in neo4j.

same label) do not need to have the same set of properties; that is, one HEMS may have information on the average daily energy usage whereas another HEMS does not. Properties can be defined as strings, lists, or numeric data types as indicated by the properties of the HEMS node. Graph traversal can be illustrated in Fig. 4.1. If one wants to know all the HEMS that are `:LOCATED_IN` the zone with id `zone_0` and `:MANAGED_BY` the DRA with id `agg_0`, then one can first find the DRA operating in `zone_0`, and then the HEMSs that are managed by the previously identified DRAs.

Cypher, the query language used in neo4j provides declarative ways of querying the database using ascii-art syntax. The above described traversal, for example, could be implemented in Cypher as

```
MATCH (Zone {id: "zone_0"}) <- [:OPERATES_IN] - (d:DRA {id: "agg_0"}
MATCH (h:HEMS) - [:MANAGED_BY] -> (d)
RETURN h.id, h.dailykWh
```

Alternatively, if one wanted to know the total average energy consumption in homes managed by `agg_0` in `zone_0`, one could modify the query above and utilize the `sum()` aggregation function.

```
MATCH (Zone {id: "zone_0"}) <- [:OPERATES_IN] - (d:DRA {id: "agg_0"}
MATCH (h:HEMS) - [:MANAGED_BY] -> (d)
```

```
RETURN sum(h.dailykWh)
```

The GDB provides great utility for reflecting the state of the system with its components and their relationships, properties, and functionalities. The GDB is not used however to log historical device-level data; more traditional databases are used for that purpose (see Sect. 4.2.2). The graph model hence provides a snapshot of the system at any given time but does not provide the functionality of observing system or device level states at historical points in time.

Simulation configuration using administration shells

Administration shells, introduced in the RAMI4.0 reference architecture [21], are virtual representations of objects that describe their technical functionalities needed for integrating, managing, and operating the object. In this simulation, each agent's administration shell comprises of the agent's functionalities and characteristics, and is captured in the form of node labels, relationships, and relationship properties in the CPES model. This then allows the GDB to serve as a configuration reference for the simulation. On startup, each agent queries the GDB and retrieves configuration parameters pertaining to the agent's behavior. Using the Bolt protocol, a connection oriented network protocol over TCP connection integrated in neo4j, the GDB microservice is made available other services.

Consider the following example. Each HEMS is tasked to simulate a house's energy using a bottom-up simulation approach based on the type and number of appliances in the home. Rather than hardcoding each house configuration as part of the Python application script, the configuration can be stored in the CPES model. The Python application than merely needs to query the GDB to determine the type and quantity of appliances to simulate. The Python application's complexity is thus significantly reduced as the same application can be deployed any number of times as long as the provided unique node identifier is captured in the CPES.

4.2.2 Data Store

The CPES modeling approach using a neo4j database presented one type of data store that is suitable for managing and configuring distributed assets. File storage and object-relational databases are other data stores more suitable for capturing

a) simulation inputs (e.g. appliance load profiles);
b) simulation events (e.g. energy usage or DR availability);
c) simulation controls (e.g. start/end of a simulation).

File Storage

Static data files, e.g. csv files, are a standard means for sharing data as they are easy to read, easy to write to, and easy to share over the web using a static file server. Take the example of Use Case A where the reference load profile of a residential home needs to be accessed by each of over 400 HEMS. The reference file is provided for download from a NGINX web server such that it can easily be modified without effecting any of the dockerized agent applications. Like all other microservices, the file server service is deployed as a Docker container. The in List. 4.1 shown Dockerfile can be used to generate the Docker image so that all files from the `static/` directory will be hosted on the server. Once the image is built, it can be deployed as shown in List. 4.2. A `loadprofile` node can be added to the CPES with the file URL as one of the properties, so that it becomes available to all agents in the agent layer.

```
1 FROM nginx
2 COPY static /usr/share/nginx/html
```

Listing 4.1: Dockerfile showing the simplicity of creating the static file server container.

```
1 $ Docker build -t nginx-file-server .
2 $ Docker run --it --restart unless-stopped -p 8080:80 nginx-file-server
```

Listing 4.2: Commands for Unix based system to build and then run the static file server. The working directory should contain the Dockerfile and the `static/` directory form List. 4.1.

Object-Relational Database

PostgreSQL, an open source database management system, is used as the centralized DB for storing information at the administrative level and supporting the web application. The PostgeSQL DB is used for storing simulation events and is modeled based on the application domain and system requirements. Each simulation may have different variables that need to be tracked or maybe even different entities and entity relationships; yet, the in Fig. 4.2 depicted relational schema was designed to support a variety of simulations. If needed, it can also easily be adjusted and expanded.

The relations in the top row refer to the "physical" components in the simulation that were also provided in the sample property graph model in Fig. 4.1. The `pricing`, `emeasurements`, and `billings` relations are capturing data pertaining to residential energy rates and usage. These three relations in combination with the addition of the `nodes` and `zones` tables would suffice to capture data generated by the simulation in Use Case A. As the ISO publishes electricity prices for each zone, these will be stored in the `pricing` table with the `zoneid` as a foreign key. As a HEMS publishes its energy consumption, these data are stored in the `emeasurements` table with a foreign key referencing a `node` in the `nodes` relation. For each `emeasurements` entry created, a `billings` entry is added as well as the product of consumed energy and electricity price at that time.

The remaining two tables in the middle column capture the reported DR availability by the nodes (`drnas`) as well as DR events that control the nodes (`drncs`). Adding these two relations to the relational schema described for Use Case A, one can log all information needed for Use Case B. DR credits are added to the `billings` relation as negative amounts.

The remaining tables to the left account for the addition of service providers, e.g. DRAs (`aggregators`), which aggregate, coordinate, and manage residential homes (`nodes`). These DRAs also report their DR availability as bids for DR services for each respective zone. Once the winning aggregator has been determined, the control event and contract could captured in the `dracs` and `transactions` relations respectively. This shows how the presented schema can easily be extended to account for additional requirements.

The current system design in itself is flexible enough to accommodate different behaviors for the same set of nodes. Instead of having many columns in each table for each variable (e.g. energy, power, voltage, current in the `emeasurements` table), all variables are stored as `jsonb` data types, which is PostgreSQL's decomposed binary format for JSON objects[1].

[1] `https://www.postgresql.org/docs/10/static/datatype-json.html`

The input format follows all JSON rules[2]. Using the `jsonb` data type provides a similar flexibility to that of popular document-based NoSQL databases, such as MongoDB[3].

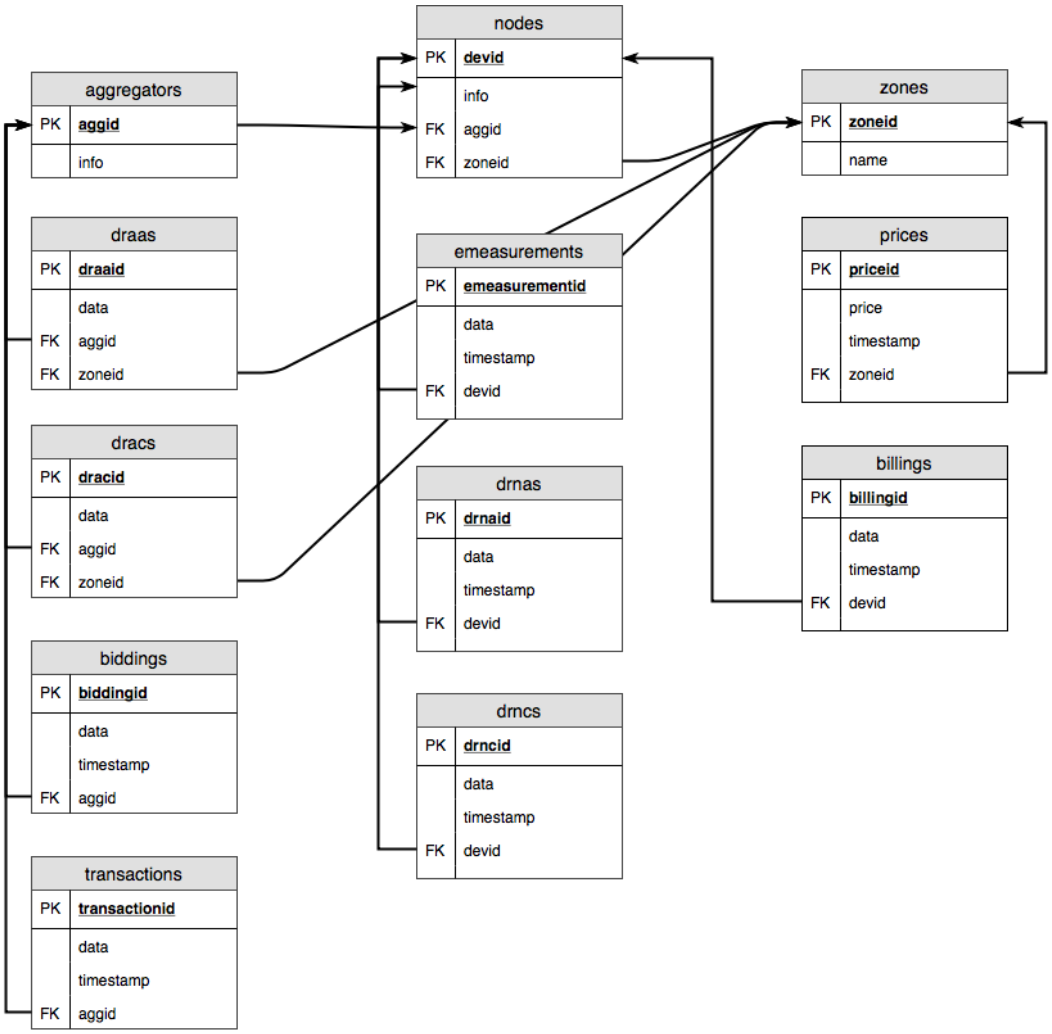


Figure 4.2: Sample relational schema for capturing data at the administrative level during the simulation.

[2] `https://tools.ietf.org/html/rfc7159`
[3] `https://www.mongodb.com`

4.2.3 Web Application

The web application, developed using the Phoenix framework, was designed to provide an interface for administrative tasks, and to capture all events in the communication layer. The administrative tasks considered are the

a) setup of the simulation by uploading DB seed files;
b) query of data from the DB;
c) monitoring of simulations;
d) coordination of manual simulation events;

Phoenix Framework and Elixir Umbrella Application

The Phoenix framework follows a server-side MVC pattern and is known for bringing concurrency and functional programming to web application development [33]. Phoenix is built using the functional Elixir programming language and thus also runs on the Erlang VM [**?**]. The functional approach to web application development and use of concurrent lightweight Elixir processes for handling real-time connections are especially well-suited for smart applications that require efficient socket connections for human-computer interactions (e.g. live updates for monitoring) as well as API-based machine-to-machine interactions (e.g. processing of IoT events). In the Phoenix framework, incoming requests are simply passed between layers and transformed at each step by groups of functions, so-called pipelines [33], making the handling of high numbers of API requests for IoT events possible. Phoenix renders web pages (templates) developed using HTML, CSS, JS, and EEx (Embedded Elixir, which allows the developer to embed Elixir code inside the HTML pages). The following code snippet for the residential home dashboard exemplifies the concept of EEx (see List. 4.3). In the request pipeline, once the request reaches the controller, the controller queries the DB and then passes the variables `currentp` and `dailyb` to the view, which in turn renders the template containing the variables in their embedded form as indicated by the `<%= %>` tags.

```
<div class="row">
  <div class="col-md-4 cards">
    <div class="card card-pricing card-raised">
      <div class="card-body">
        <h3 class="card-category">Demand</h3>
        <div class="card-icon icon-primary">
          <i class="material-icons">power</i>
        </div><br>
        <h3 class="card-title"><%= @currentp %> W </h3>
        <p class="card-description">
        Demand is the last reported value from the device.
        (Updated every 5 min.)</p>
      </div>
    </div>
  </div>
  <div class="col-md-4 cards">
    <div class="card card-pricing card-raised">
      <div class="card-body">
        <h3 class="card-category">Billing</h3>
        <div class="card-icon icon-primary">
          <i class="material-icons">monetization_on</i>
        </div><br>
        <h3 class="card-title">$ <%= @dailyb %> </h3>
        <p class="card-description">
          Energy bill for current day in USD. Cost per kWh is set by the ISO.
        </p>
      </div>
    </div>
  </div>
</div>
```

Listing 4.3: HTML code snippet exemplifying the use of EEx in Phoenix templates.

The web application is implemented as a child application in an Elixir Umbrella application [34]. Using an Elixir Umbrella application scheme allows us to separate system requirements into multiple child applications while making each child aware of its siblings. This allows us to separate the two main functionalities (administrative tasks and event capturing) into two applications: a Phoenix app that provides the web interface for administrative tasks and an Elixir app (MqttHandler) that subscribes to events from the communication layer. Both applications can access the code of one another, so that the MqttHandler app can use

the Pheonix app's API to log events to the DB, and so that the Phoenix app can publish MQTT through the MqttHandler app's API to the message broker.

Administrative Tasks

Simulation setup. *The testbed system should allow the user to upload the network configuration files before starting a new simulation.* The settings page is designed to help with setting up simulations by providing an overview tab that summarizes all agents currently stored in the DB, a data upload tab that allows the user to upload individual csv files for HEMSs or DRAs for example, and a simulation tab that allows the user to start/stop the simulation.

Data Query. *The web interface should allow the user to query data through the web interface.*
To simplify data retrieval from the DB, a web form is provided. The user thus does not have to log in to the server, log in to the database management system, and query the data of interest, instead, the web form can be used. Queries can be exported as csv files.

Simulation Monitoring. *The simulation platform should provide an interface so that the user may monitor ongoing simulations.*
Monitoring of the simulation can be performed through various dashboards that present data at various levels: the ISO level, the DRA level, and the HEMS level. Each dashboard varies in its emphasis on data pertaining to respective behaviors.

Manual Simulation Events. *The testbed system should allow the user to manually initiate events of interest, e.g. the loss of generation.*
Based on the simulation design process (Sect. 2.2), each simulation should be designed to test certain events, functionalities (e.g. algorithms), etc. When testing a distributed system's ability to respond to a sudden loss of generation for example, one may want to initiate events manually rather than having to pre-program it somewhere. That is, the web application should provide a control interface that allows the administrator to request a reduction in demand by some desired amount. Once the action has been initiated by the user through the web interface, the action is communicated to the agents in the agent layer to respond to that scenario. In Use Case B, a message is published to the `drsim/event` topic that the

ISO listens and reacts to by then in turn sending control commands to the HEMSs that can provide DR services at that time.

Event Capturing

An MqttHandler app is responsible for capturing events published by nodes in the communication layer, and thus heavily depends on the in Sect. 4.3 described communication scheme and API design. To implement the MQTT communication scheme, the application uses the tortoise package[4], a MQTT client for Elixir. List. 4.4 shows how each agent type is handled in its own module that is started and supervised by the application[5]. That is, when the application starts, the `:start_link` function of each module is called.

```
defmodule MqttBroker.Application do
  @moduledoc false

  use Application
  def start(_type, _args) do
    # List all child processes to be supervised
    children = [
      %{
        id: PriceLogger,
        start: {PriceLogger, :start_link, ["my_client_id_24"]}
      },
      %{
        id: HemsLogger,
        start: {HemsLogger, :start_link, ["my_client_id_26"]}
      },
      %{
        id: Publisher,
        start: {Publisher, :start_link, [[]]}
      }
    ]
    opts = [strategy: :one_for_one, name: MqttBroker.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

Listing 4.4: Supervisor of the Elixir app with one child for each type of agent.

[4] https://hex.pm/packages/tortoise
[5] https://hexdocs.pm/elixir/Supervisor.html

List. 4.5 shows that the `HemsLogger` module will then start a new connection to the message broker and subscribe to a list of topics. The following `:handle_message` functions are callback functions that are executed for every incoming message and use pattern matching on the message topic in their function parameters. The `Dris.Data` module stems from the Phoenix app (called `Dris`), which exemplifies how code from the sibling application is made available to the MqttHandler app in the Elixir Umbrella application.

```
defmodule HemsLogger do
   # code omitted here #

   def start_link(clientid \\ "my_client_id_2") do
     Tortoise.Supervisor.start_child(
          client_id: clientid,
          handler: {HomesTortoise, []},
          server: {Tortoise.Transport.Tcp, host: 'post.redlab-iot.net', port: 55100},
          subscriptions: [{"metering/energy/#", 0},{"drna/now/#", 0} ])
   end

  # code omitted here#

  def handle_message(["metering", "energy", agg, dev], message, state) do
    b = message |> Poison.Parser.parse!
    attrs = %{agg_id: agg, dev_id: dev, data: b["data"]}
    Dris.Data.create_emeasurement(attrs)
    {:ok, state}
  end

  def handle_message(["drna", "now", agg, dev], message, state) do
    b = message |> Poison.Parser.parse!
    attrs = %{agg_id: agg, dev_id: dev, data: %{now: b["value"]}}
    Dris.Data.create_derna(attrs)
    {:ok, state}
  end
end
```

Listing 4.5: Extract of the HemsLogger module showing how each process subscribes to the communication broker and handles incoming messages.

Deployment

As with all other system components, the web application is deployed as a Docker container. Consisting of the Phoenix application for administrative tasks and the Elixir application for event capturing, the Docker container is built using the Dockerfile shown in List. A.5. Once built, the image can be deployed and run to host the web application on any system and also across multiple systems.

4.3 Communication Layer Implementation

The second layer of the simulation platform is the communication layer with the following requirements:

a) The communication layer should utilize a lightweight Pub/Sub communication scheme designed for resource-constrained devices;
b) The communication layer should provide an easy-to-use API;
c) The communication layer should address cyber security concerns.

4.3.1 MQTT Data Protocol and Message Broker

MQTT is a lightweight Pub/Sub protocol that is widely used for resource constrained sensor devices. MQTT has excellent compatibility with various programming environments, sits on top of TCP/IP, and offers TLS support. The Pub/Sub protocol implies that a client publishes messages to a topic without knowing who is subscribed to that topic. The subscriber does not know who publishes messages in a topic and who else is subscribed to it, but will receive any message published to that topic [4]. Fig. 4.3 depicts this concept of publishers, topics, and subscribers. In addition to communication clients (publishers and subscribers), the Pub/Sub messaging scheme requires a message broker that filters messages and routes them from publishers to subscribers. The message broker is further responsible for client authentication and authorization, which means that one may configure the system to only allow registered users to participate and to publish/subscribe to certain topics.

Here, the VerneMQ MQTT Broker is used. VerneMQ is built on the Erlang/OTP (open telecom platform), a platform that has proven its concurrency model and fault tolerance in the operation of telecommunication networks. VerneMQ was specifically designed for soft real-time, distributed control and messaging applications while providing fault-isolation and

fault-tolerance [35]. A Docker image for the VerneMQ MQTT broker is provided by the creators [36].

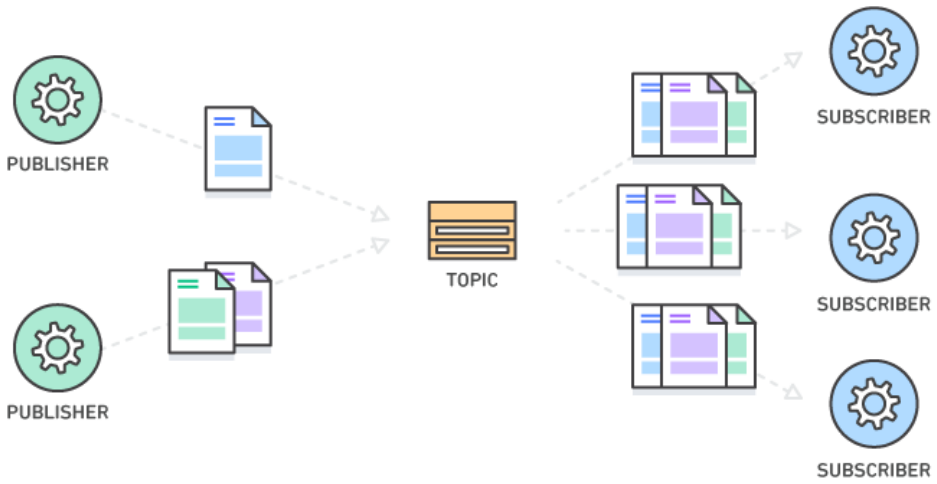


Figure 4.3: Conceptual depiction of the publish/subscribe protocol where clients can either publish or subscribe to topics [4].

4.3.2 MQTT API Design

A generalized topic scheme was established to provide a standardized API for nodes in the agent and administration layer. Table 4.1 shows a selection of topics separated by the type of client that would publish to the topic. Topics are designed to be intuitive, to allow efficient usage of wildcards—placeholders for single or multiple topic levels when subscribing to multiple topics at a time—, and to ease the authorization for topics. As shown in Table 4.1, message payloads are formatted as JSON objects. The shown keys for each topic are only select samples for the performed simulations, but they typically just depend on the simulation.

Each topic consists of multiple levels, where the first level generally indicates the type of data or the type of action that is captured in the payload. Topics starting with the term `meterings` for example refer to sensor data (measurements) that are being reported, and topics starting with `dr` relate to DR monitoring or control events. That is, the `drna` topic informs on the "demand response node availability" for different time intervals and the `drnc` topic represents a control command for nodes (e.g. a single HEMS) to reduce/increase energy usage.

To understand why efficient wildcard usage is important, consider the example of the administration layer that needs to capture all energy measurements published by HEMSs. Instead of having to subscribe to every single topic in that category, i.e. all devices, the multi-level wildcard `#` can be used. The administration layer, or more specifically the `HemsLogger` (see List. 4.5), could then just subscribe to `metering/energy/#` and receive all published energy measurements. The reverse—having a unique identifier first and the message type last— would not be possible as multi-level wildcards are only allowed at the end of a topic. The same reasoning applies to the authorization of clients that allows them to only publish and subscribe to certain topics. For the `metering` example, the HEMS would only be allowed to publish to the `metering/energy/aggid/devid` topic, where `aggid` and `devid` vary for each HEMS, whereas the DRA and ISO could be authorized to subscribe to `metering/energy/aggid/+` and `metering/energy/#` respectively. (The `+` symbol is a single-level wildcard.)

The implementation of client/topic authorization schemes generally depends on the message broker being used. In the case of VerneMQ, either file-based or database-based authorization is possible (see [37]).

Table 4.1: Overview of select MQTT topics ands their sample payloads.

Publisher	Topic	Payload Sample
HEMS	`metering/energy/aggid/devid`	`{"avgp": _, "energy": _, "vrms": _ }`
HEMS	`metering/bess/aggid/devid`	`{"soc": _, "state": _}`
HEMS	`metering/degen/aggid/devid`	`{"avgp": _, "energy": _}`
HEMS	`drna/aggid/devid`	`{"now": _, "hour": _, "day": _ }`
ISO, DRA	`drnc/aggid/devid`	`{"type": _, "amount": _, "time": _}`
DRA	`draa/zoneid/aggid`	`{"now": _, "hour": _, "day": _ }`
ISO	`drac/aggid`	`{"type": _, "amount": _, "time": _}`
ISO	`iso/rtp/zoneid`	`{"value": _ , "ts": _}`
admin	`drsim/settings`	`{"mode": "start/stop"}`
admin	`drsim/events`	`{"type": _, "amount": _}`
admin	`set/drmode/devid`	`{"value": "on/off"}`
admin	`get/info/devid`	`{"type": _ }`

4.3.3 Cyber Security

Securing devices, communication, and data are a growing concern when considering distributed systems, e.g. IoT systems, with large numbers of connected devices. As with most information systems, the security level provided by the MQTT communication protocol strongly depends on how it is being implemented and used. At the data protocol level, confidentiality can be ensured using authentication and authorization methods that restrict clients in their ability to publish and subscribe to certain topics. Doing so ensures that data are not simply publicly shared. As aforementioned, using VerneMQ, a database can be used with an entry for each client that specifies the client's username, password and topics that the user can publish and/or subscribe to. Using a scalable, reliable, and high-performance message broker such as VerneMQ, can help in ensuring availability of the MQTT messaging service. The MQTT protocol further allows the option to use SSL/TLS for added confidentiality and integrity, although this can come at a cost for resource constrained IoT devices. Ultimately, the above stated options are only optional security features, and it is left to the system designer to make use of them. For some simulation purposes, these features can be ignored, and for others they should not, depending on the goal of the simulation.

Although not in the scope of this work, cyber-security motivated simulations can be considered for this simulation tool in the future. This could entail the use of additional encryption algorithms at the agent-level, the use of a different data protocol altogether, or the implementation of emerging blockchain technologies. This simulation tool is well suited for such studies given the multi-agent nature of the tool, modular design, and containerized implementation.

4.4 Agent Layer Implementation

The agent layer implements the micro level behavior of the distributed nodes which then translates into the macro level behavior in the virtual grid. Provided the requirements below, the agent layer typically demands the most modifications out of the three layers for each simulation.

a) Implement agent behavior in the form of dockerized Python applications;
b) Implement the aforementioned communication scheme;
c) Implement the ability to query information from the administration shells in the CPES model.

4.4.1 Common Interfaces

Requirements b) and c) are shared among all agent types and refer to the implementation of MQTT and Bolt clients. As shown in List. B.1 and List. B.3 the `paho-mqtt`[6] client library and the `neo4j-driver`[7] are used respectively to achieve this. All agents are dockerized using the in List. A.7 shown Dockerfile. The required libraries and drivers are specified by means of text files that are copied into the Docker container and then parsed for installation in the build process. Using the `alpine` flavor of the base image reduces the overall size of the container. To differentiate containerized applications that share the identical code base, a unique identifier is passed as an environment variable to the container and the Python application parses the environment variable and queries the CPES model based on that identifier. List. 4.6 and List. 4.7 demonstrate this concept as well as the spawning of containers using a for-loop iteration.

```
1 for i in {1..10}
2 do
3   Docker run -it --restart unless-stopped -d --name housebot-$i -e devId=dev_$i
      ↪  housebot-v2 python app.py
4 done
```

Listing 4.6: Bash command to to start ten Docker containers, each with a different name and environment variable. The environment variable is picked up by the Python script in the container.

[6]https://pypi.org/project/paho-mqtt/#client
[7]https://neo4j.com/docs/api/python-driver/current/

```
1 import os
2 devId = os.environ["devId"]
```

Listing 4.7: Python code snippet showing how environment variables from the Docker environment are loaded.

4.4.2 Use Case A

Recall that in this example, the objective was to simulate a dynamic pricing DR program where residential consumer publish their energy usage every five minutes and the system operator publishes the electricity price every 60 minutes.

ISO

In addition to the above described common interfaces, the specifics of the behavioral strategy of the ISO are rather simple. The ISO publishes electricity prices to the `iso/rtp/zoneid` topic, where the `zoneid` indicates the regions where the price applies. Fig. 4.4 shows the flowchart for this scenario.

HEMS

The HEMS agent type follows a very similar program flow to that aforementioned, with the only difference that electricity data were sent more frequently (5-minute intervals). In addition, instead of price data, the energy data are being sent. Given the program similarities, Fig. 4.4 can be used as a reference for the program flow for the HEMS model.

4.4.3 Use Case B

Use Case B includes all of the in Use Case A described concepts, with the following additions:

a) a bottom-up simulation approach for the residential home behavior;
b) user initiated control events to shed a desired amount of load immediately;
c) DR credits provided to the user in the case of an event taking place and load being shed.

ISO

The implementation of the ISO is much the same to that conceptually depicted in Fig. 4.4, with the added functionality that the ISO can send control signals to individual HEMS agents

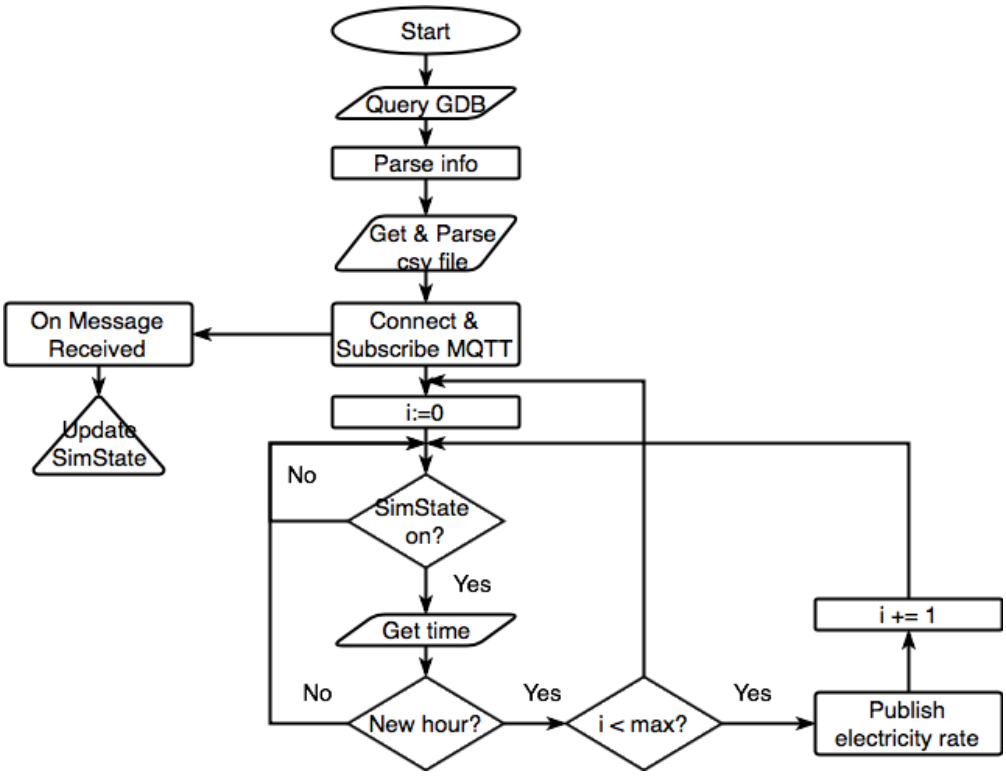


Figure 4.4: Flowchart of the ISO agent for Use Case A. The same flowchart also applies for the HEMS, only that the frequency of published data and the data itself (energy data not electricity rates) are different.

to request immediate load shedding. As users manually interact with the web application and initiate DLC events, the web application publishes a message for each interaction to the `drsim/events` topic with the type (DLC) and amount (5, 50, 100, 500, 1000 kW) specified in the payload (see Table 4.1). The ISO is subscribed to this topic and then immediately schedules each individual node to request the load reduction. The ISO retrieves the most recent time-ordered list of availabilities by node, aggregator, and zone, and then starts curtailing loads on a last-come, first-served principle until the requested amount of DR is dispatched (open-loop control). Last-come, first-served here means that the node with the most recent update on its availability—i.e. the last to publish its value—is curtailed first as the probability that the reported value is still valid is highest. Since the ISO does not keep a data store on its own to store node availabilities for the scope of this work, a JSON API request is made to the web application, which has the central data store of all published availabilities, measurements, etc.

HEMS

A bottom-up approach is used for the HEMS agent to model the individual power consumption of the house's appliances throughout the day, to publish aggregated consumption and availability data to the system, and to subscribe to control events from the system operator. [38] designed an educational MATLAB Simulink model that simulates fixed and controllable appliances, an HVAC system, an electric water heater, a PV system, a BESS and an aggregator, and allowed users to test their DR bidding strategies for residential homes. This model was adapted into the HEMS Python application after simplifying it to only consist of fixed appliances, a variably controllable HVAC system, and an added baseload. Table 4.2 summarizes the model's load types and their parameters.

The house load at a given time step t_i consists of three components: the sum of the loads due to fixed appliances that are scheduled to run at t_i, the HVAC load at t_i based on the schedule and house's thermal model, and the baseload at t_i determined by the reference loadprofile. Having already described the latter part in Fig. 4.4, let us consider the first two components further.

Starting with N appliances, we can let $\mathbf{a}$ be the list of appliances, and $\mathbf{s}$ be the list of states with s_n being the state of a_n at a given time t_i. Then the power consumption P_a due to all fixed appliances is

$$P_a = \mathbf{a} \cdot \mathbf{s} \tag{4.1}$$

Table 4.2: Overview of appliances considered in the house model.

Appliance	Controllability	Power	Scheduled Usage
Lights	fixed schedule	360 W	5-8am, 6-11pm
Refrigerator	fixed schedule	200 W	12-1am, 5-6am, 8-9am, 11-12pm, 3-4pm, 7-8pm
TV & Entert.	fixed schedule	200	5-10pm
Range & Oven	fixed schedule	1500 W	11-12pm, 6-7pm
HVAC	schedule & variable control	3500 W	12am-3am, 10-1pm, 5-11pm
Baseload	none	250 W*	N/A

* Baseload is 25 % of the reference loadprofile (Sect. 4.5 at any time in the simulation.

The Advanced Python Scheduler (APScheduler)[8] library presents a convenient tool for

[8]See `https://tinyurl.com/hs-ms-apscheduler`

scheduling the appliances based on their states; that is, using the Cron-style[9] scheduling method, we turn each device on *on every day* and *on the hour(s) of the day* provided and also turn them off *on every day* and *the hour(s) of the day* provided. Since the HVAC system is also running on a predefined schedule, it's on/off switching can be handled in much the same way as the fixed appliances. In addition to its schedule, the HVAC system's power draw P_{HVAC} depends on the house and environmental conditions. That is, control outputs are designed to keep the house at a comfortable setpoint temperature T_{SP} (e.g. $23°C$) during the HVAC system's scheduled time of operation.

In the considered thermal model based on [38], the temperature changes based on the inside temperature T_H, the ambient temperature T_a, the equivalent heat capacity of the house C_H, thermal equivalent resistance R_{TH} and the HVAC system. The first-order thermal equivalence circuit of the house is shown in Fig. 4.5 and can be described as

$$\frac{dT}{dt} = \frac{-1}{C_H}\left(P_{HVAC} + \frac{T_H - T_a}{R_{TH}}\right) \tag{4.2}$$

Where C_H is the heat capacity of the room measured in J/K and R_{TH} the thermal equivalent resistance in K/W. From Eq. 4.2, the temperature of the house T_{H_i} at time t_i given $T_{H_{i-1}}$ t_{i-1} is given by

$$T_{H_i} = T_{H_{i-1}} - \frac{1}{R_{TH}C_H}\int_{t_{i-1}}^{t_i}\left(R_{TH}P_{HVAC} + (T_{H_{i-1}} - T_a)\right)d\tau \tag{4.3}$$

During the HVAC system's operation, variable control is used during the HVAC system's "on"-time to keep the temperature at a comfortable level. A proportional-integral (PI) controller was implemented for this purpose as it is a fundamental control mechanism for feedback control systems and widely used in the industry, especially in HVAC applications [39, 40]. The gains of the controller, k_p and k_i, determine the control output given current and time-integrated errors, as described below.

$$c(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau \tag{4.4}$$

Where e is the temperature difference $T_a - T_{SP}$ (positive difference indicates cooling), and $c(t)$ is the control output at time t. Considering that the HVAC system is the only load that can be shed, the total DR availability at any time is expressed by P_{HVAC} (i.e. the product of the HVAC state and control variable).

[9]See `https://tinyurl.com/hs-ms-crontab`

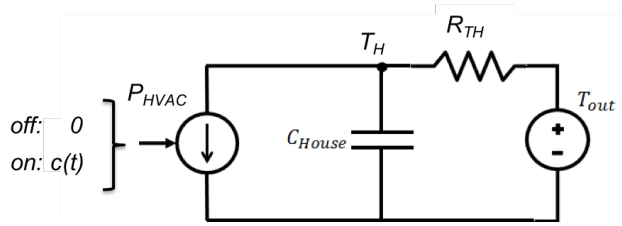


Figure 4.5: Thermal equivalent circuit of the first-order thermal system.

The Python application approximates the first-order system (Eq. 4.2) using Eq. 4.3 in its discrete form. The discrete time step should be sufficiently small based on the system's dynamics. That is, depending on the heat capacity and the thermal resistance, or in other words the system's time constant $\tau = RC$, the scheduling interval for the `updateHouseTemp()` function should be adjusted. Table 4.3 summarized the parameters used during implementation. However, since these values are stored in and queried from the CPES model, they can be updated for different simulation runs.

Based on the MQTT API design in Table 4.1, the HEMS application subscribes to the a) `drnc/aggid/devid` and b) `set/drmode/devid` topics. Should an event be published to topic a), the HEMS will shed its load by temporarily raising the HVAC's temperature such that P_{HVAC} decreases. This provides more flexibility in terms of shedding loads gradually, but for the scope of this work, raising the temperature is considered equivalent to turning it off. Events on the latter topic, topic b), control the HEMS participation in DLC events based. Should the DR mode be set to off, the HEMS reports zero availability and does not respond to messages on topic a).

Table 4.3: Setpoints, environmental conditions, and thermal properties used during implementation.

Parameter	Value
Setpoint T_{SP}	$23°C$
Initial Temp. T_0	$23°C$
Ambient Temp. T_a	CSV file*
Heat Capacity C_H	$2.25e5\ J/K$
Equiv. Resistance R_{TH}	$5.7e-3\ K/W$

* Ambient temperature is provided at a 5-min resolution.

4.5 Simulation Parameters

To deploy and run the simulation, a combination of cloud and in-house hosting services was used. The Linode[10] platform was used primarily with a total of four available Linodes and one in-house server. The following table summarizes the type of server used, the domain assigned to them, and the type of micro-service hosted on them. Except for the Nanode, which was hosted in Dallas, TX, USA, the Linodes were all hosted in Fremont, CA, USA.

Table 4.4: Overview of servers used for the simulation.

Domain	Server Type	RAM	CPU Cores	Usage
`linode1.redlab-iot.net`	Linode 4GB	4 GB	2	HEMS
`linode2.redlab-iot.net`	Nanode 1GB	1 GB	1	ISO, Web Application
`linode3.redlab-iot.net`	Linode 4GB	4 GB	2	HEMS, File Server
`linode4.redlab-iot.net`	Linode 2GB	2 GB	1	HEMS
`post.redlab-iot.net`	In-house*	64 GB	8	VerneMQ, PostgreSQL

* Dell r7805 server in the REDlab Manoa's server rack located at the UH at Mānoa

Table 4.5 and Table 4.6 show the configuration parameters for both implementation scenarios. In both scenarios, the time scale of the simulation was real-time and nodes were synchronized using UTC time since servers were split across different timezones and time commands in most programming languages default to UTC time as well. In Use Case A, 441 simulation HEMS nodes were deployed across four servers and the simulation ran continuously for about three days.

Table 4.5: Simulation configuration parameters for Use Case A.

Parameter	Value
Simulation time step	real-time
Simulation duration	3 days
Simulation time zone	UTC time
ISO nodes	1
Zones	2
Aggregator nodes*	3
HEMS nodes	441

* These nodes are only considered for the CPES model.

Use Case B on the other hand entailed only 200 simulation nodes due to a an increased

[10]https://www.linode.com

memory utilization for the HEMS node. Since manual interaction with the system was required in Use Case B, the duration of the simulation varied. Based on the time of interaction, one would observe different behaviors as different appliances are scheduled at different times in the UTC timezone.

Table 4.6: Simulation configuration parameters for Use Case B.

Parameter	Value
Simulation time step	real-time
Simulation duration	variable
Simulation time zone	UTC time
ISO nodes	1
Zones	2
Aggregator nodes*	3
HEMS nodes	440

* These nodes are only considered for the CPES model.

The reference load profile acquired by the REDLab Manoa for a residential home—used as a simulation input—is shown in Fig. 4.6 and Fig. 4.7. The 5-minute temporal resolution is maintained in Fig. 4.6, while Fig. 4.7 resampled the load data with hourly averages to overlay it with the referenced electricity rates.

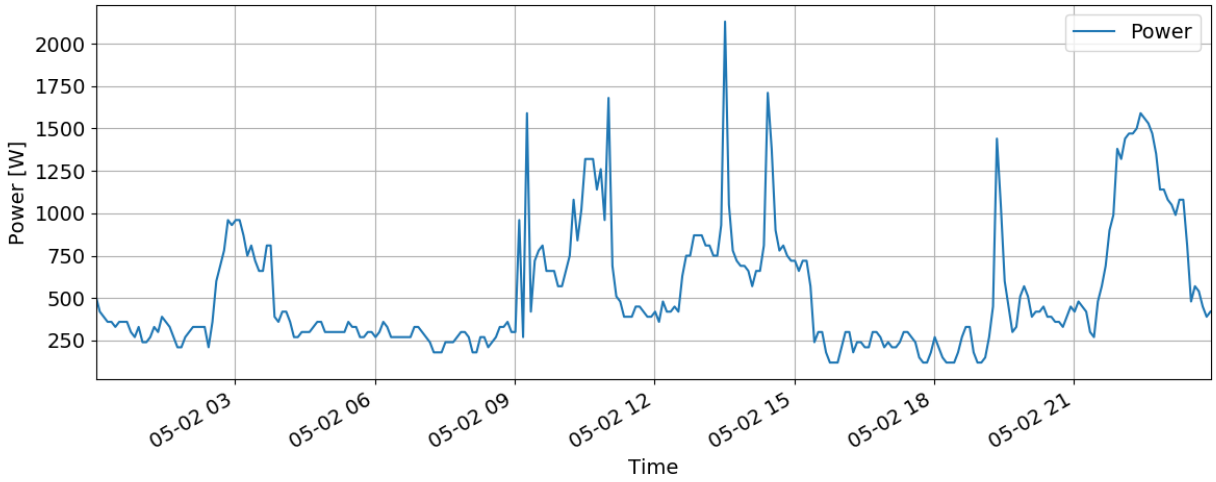


Figure 4.6: Reference load profile of a residential home on Oahu, Hawaii. The dates were shifted forward by 18 days for graphing purposes.

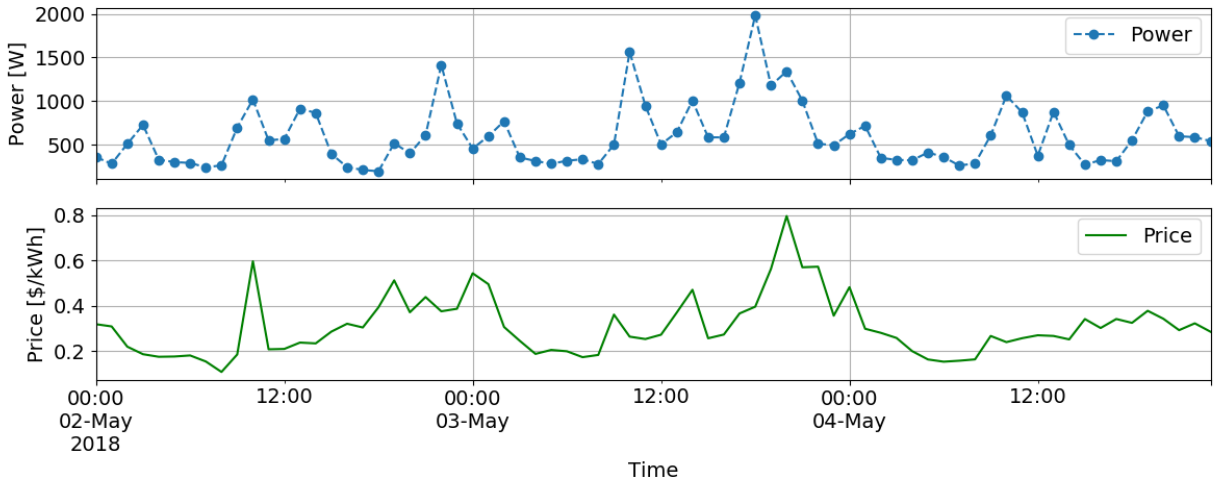


Figure 4.7: Reference load profile at a 1-hour temporal resolution (top) plotted along reference electricity rates (bottom). Dates are adjusted for graphing purposes.

CHAPTER 5
TESTBED PLATFORM EVALUATION

This chapter presents the results of the use cases with respect to the overall testbed system; that is, the testbed is evaluated based on its ability to manage multi-agent co-simulations. The smart metering application proved to be a good first use case as it required the use of the CPES modeling methodology to share information (e.g. configuration settings) across distributed agents. The web application proved effective in providing administrative and event logging functionalities as illustrated by a series of screen captures of the front-end interface. Using a simple enough, yet realistic, example, a baseline was obtained for possible resource requirements for the designed micro-service architecture. The bottom-up building model additionally showed how existing CPES models can be modified and extended to model the required system complexities for respective applications.

5.1 CPES Modeling

A graph-theoretic CPES modeling approach was used to represent agents and their administration shells through node labels, relationships, and relationship properties. Fig. 5.1 and 5.2 show subgraphs of the implemented neo4j database demonstrating how this modeling approach has been implemented at different levels of granularity. Fig. 5.1 provides a coarse-grained overview of the CPES with a 50 node subset of HEMSs. The graph visualizes how an aggregator's presence in a residential area (zone) is simply determined by the location of the houses that the aggregator is managing. The reference files for the load profile and price rates are similarly depicted. In the shown implementation, price profiles were specific to each zone, and the load profile was generic to all nodes. If one wanted to change this, one could (a) do it at the individual level where the `loadprofile` is connected to `hems` nodes directly; (b) do it at the aggregator level where the `loadprofile` is connected to `agg` nodes; (c) do it at the zone level, as it's currently being done for the `priceprofile` nodes.

In contrast, the subgraph depicted in Fig. 5.2 focused on a single HEMS, its neighbors and the load profile. This type of subgraph is queried by the HEMS application on startup; it contains all the information it needs to determine (a) which zone to subscribe to for time-varying electricity rates; (b) which aggregator it is managed by, and (c) which reference inputs to use (e.g. for load profile or weather data). The remaining two small nodes below the `hems` node are `drna` and `emeasurement` nodes, which could potentially be used to store

latest information on DR availability and power consumption. During initial testing, the MqttHandler app (see Sect. 4.2.3) was also responsible for updating these two nodes in the GDB for every incoming message, but the implemented connection protocol to the neo4j database was not sufficiently stable to support reliable updates. Further work is needed to publish real-time sensor data updates to the CPES model.

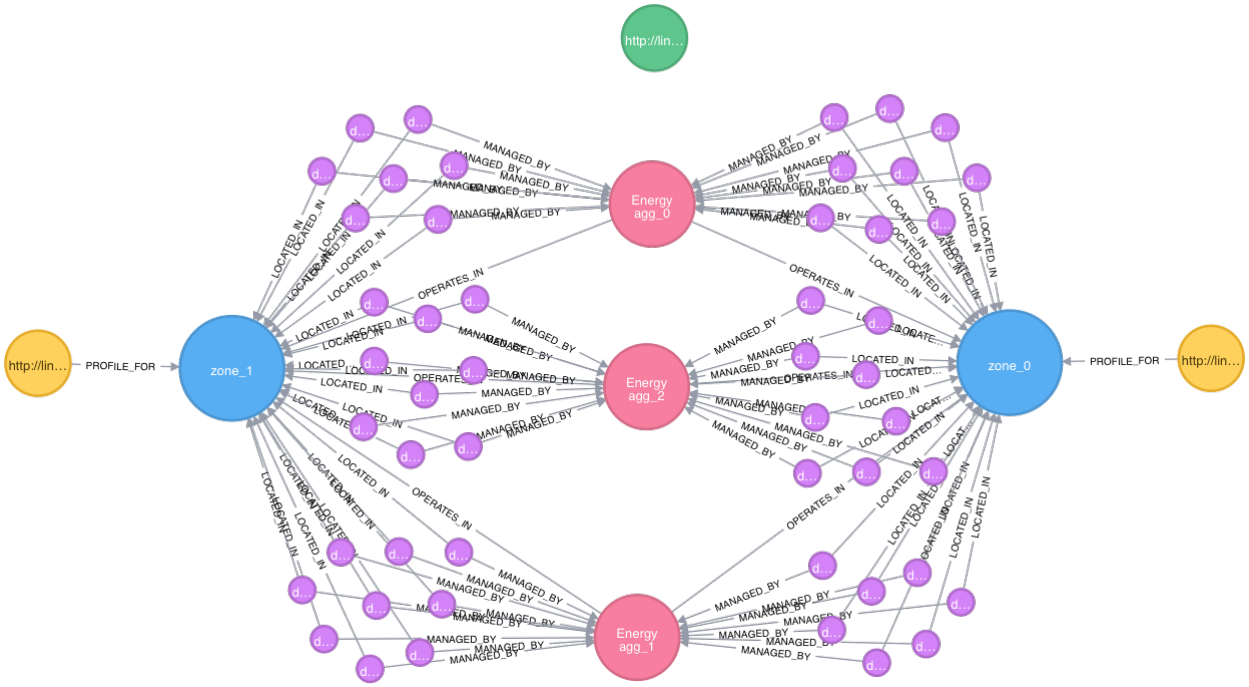


Figure 5.1: Subgraph showing the key node types and their connections. The largest nodes (blue) have the `zone` label, the centered nodes (red) have the `agg` label, the smallest nodes (violet) have the `hems` label, the outermost nodes (yellow) have the `priceprofile` label, and the isolated top-most node (green)has the `loadprofile` label. 50 `hems` nodes were queried and shown in this subgraph.

To meet the requirements for Use Case B, the in Fig. 5.1 shown CPES was extended to also include weather profiles and appliances, as shown in Fig. 5.3. Similarly to the load profile, these nodes can be left isolated to make them apply to all agents, or they can be related to zones, aggregators, or HEMS agents individually.

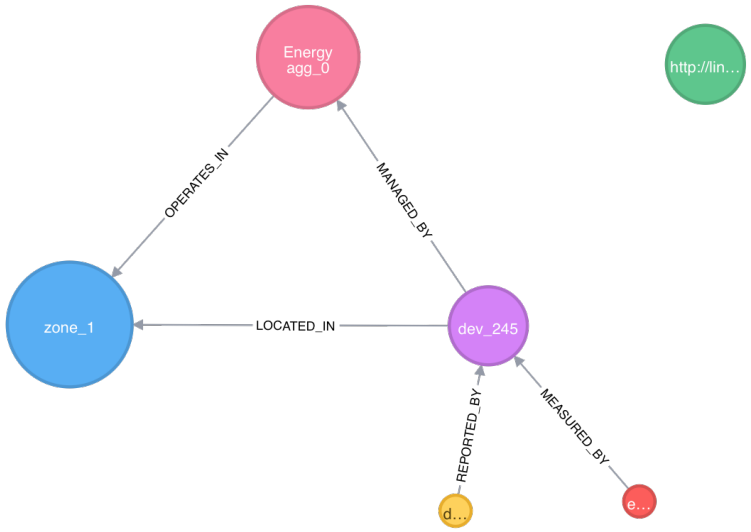


Figure 5.2: Subgraph showing the `dev_245`-node with its neighbors and the `loadprofile` node (left-most node). The two smallest nodes are `drna` and `emeasurement` nodes that could potentially be used to store descriptive data or real-time updates.

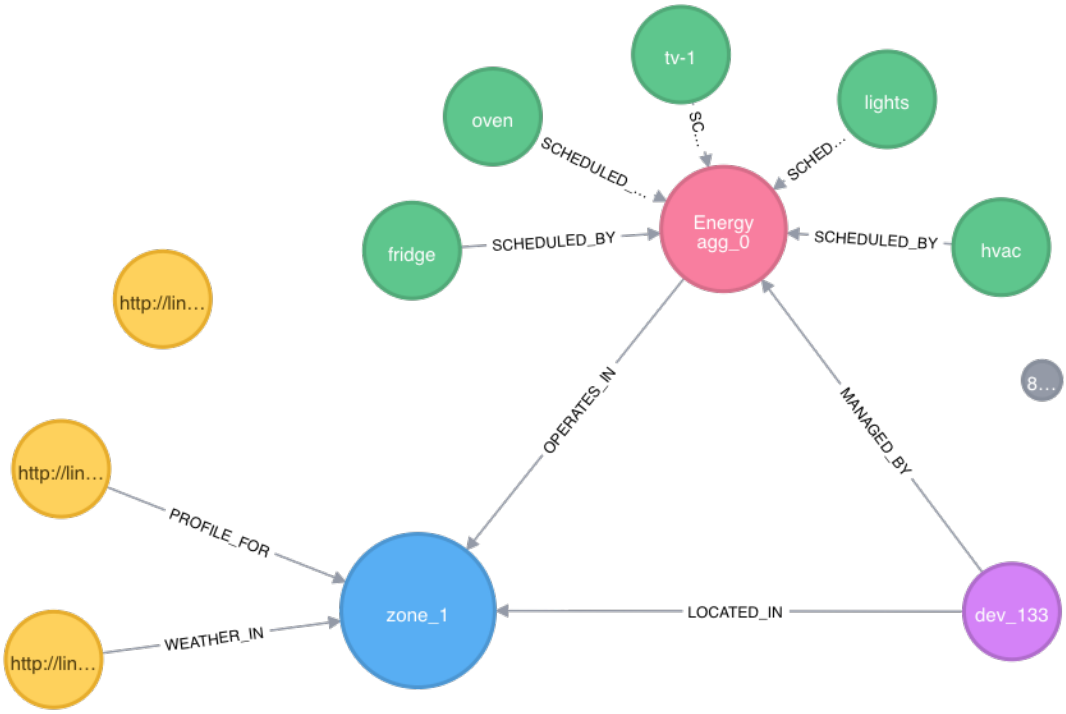


Figure 5.3: Extended CPES model to account for appliances (green nodes) and weather profiles (left and bottom-most yellow node). Appliance nodes connect to the aggregator by a `SCHEDULED_BY` relationship and contain information from Table 4.2 as node properties.

5.2 Web Application

5.2.1 Administrative Tasks

The web application proved functional to support testbed administration, simulation monitoring, data storage, and manual event control. Each of these is captured through screen captures and their descriptions in this section and Appendix D. The general structure of the interface is presented in Fig. D.1 that shows that the side menu structures the application into dashboard pages (User, Aggregator, and System Operator tabs) and administration pages (Graph-DB, Data, and Settings).

Data Upload

From the settings page, one can navigate to the data tab (Fig. 5.4a) and upload seed data as csv files using simple forms (Fig. 5.4b). The simulation can be started from the simulation tab, also shown in Fig. D.2.

Simulation Monitoring

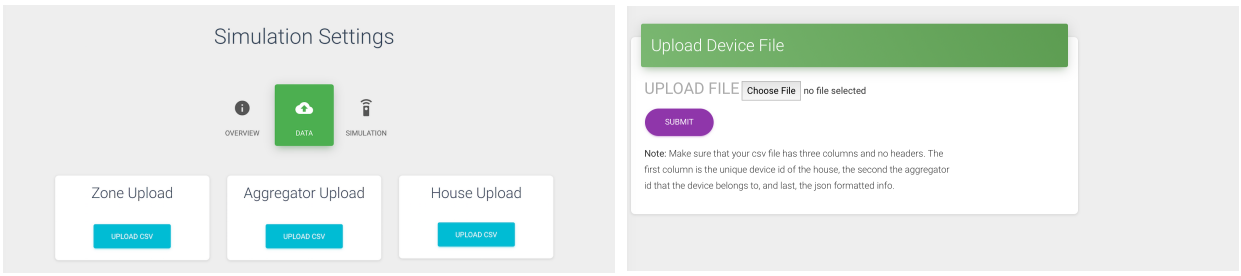
The simulation can be monitored in several ways, one being the use of dashboards. Using dashboards like those shown in Fig. D.4 and Fig. D.6 for an individual home and the system operator respectively, ensures that the system is running and logging data. They can also be used in the decision making process for manual event interactions when one needs to decide when, where, and how much load to curtail.

Data Query

The data query tool is shown in Fig. D.8. The user is provided with predefined queries to choose from. The PostgreSQL can also be accessed directly for custom queries.

Manual Interaction

In addition to the starting and ending of simulations, the web application provides options to directly play a role in the simulation. That is, the HEMS and ISO dashboards have options to curtail load immediately through the use of buttons (see Fig. D.4 and Fig. D.7).



(a) Data Upload Overview (b) Upload Form

Figure 5.4: Settings view of the interface with options for csv upload to populate the database with agents and their configuration.

5.2.2 Event Capturing

The administration layer proved capable of subscribing to all simulation events and logging them to the respective DB relations. One additional aspect of interest in distributed sensor and control systems is that of the delay. Delays can occur at the data transfer level (communication protocol) or the data processing level (application level). The delay at the data transfer level has been studied extensively for the MQTT protocol (see [41] and references therein); here, we analyze the delay at the processing level which is mainly associated with the software level implementation (see Sect. 4.2.3).

The reported and stored energy measurements from the HEMS nodes were used to quantify the overall delay between the event that data are sent and the event that data are inserted into the DB. More specifically, the HEMS is programmed to add a Unix timestamp[1] to its payload, which then gets logged with the measurement data. The web application additionally adds a timestamp upon data insert. Analyzing $N = 100000$ records from the `emeasurements` relation showed an average delay of 13 seconds with a standard deviation of 8 seconds (Table 5.1). The frequency distribution in Fig. 5.5 showed a right-skewed distribution with a median of 12 seconds.

Although the delay in the event capturing component of the system was not interfering with the systems ability to run the simulation, it suggests that there is a bottleneck in the logging component of the system. Given low server utilization, it appears that it is not a problem associated with limited computing resources and thus cannot be mitigated by deploying the system with more computing resources. More likely, the problem is rooted in the event subscription method for the MqttHandler app that cannot process sufficiently large numbers of messages simultaneously. This can be tested by implementing a VerneMQ plugin

[1]Unix timestamps are expressed in seconds since January 1, 1970 at UTC

with similar functionalities to that of the MqttHandler for the message broker itself, such as described in [18], and comparing the performance of both approaches. Further investigation is therefore needed to identify the system's bottleneck.

Table 5.1: Summary of statistics for the data processing delay.

Parameter	Value
N	100000
Mean	13
Median	12
Std. Deviation	8
Minimum	0
Maximum	62

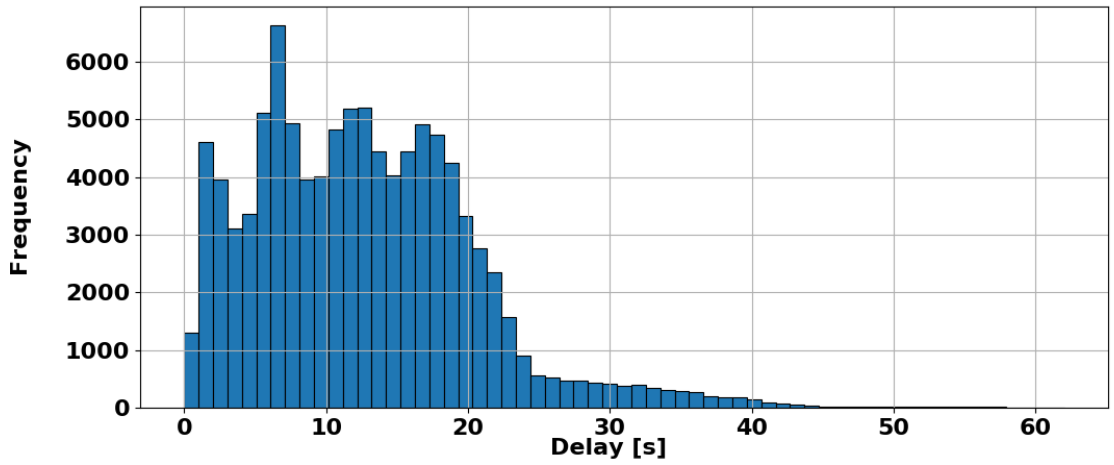


Figure 5.5: Right-Skewed distribution of delays depicted in a histogram with apparent interval limits at each second. The mean, median, and standard deviation are 12.7, 12.0 and 7.9 respectively.

5.3 Deployment

All application services are deployed as Docker containers. The pool of HEMS applications requires the most resources overall as it has the most instances (i.e. 441 containers in Use Case A). Table 5.2 describes the resource usage of each container type obtained with the `docker stats container-name` and with respect to the server that they are deployed on. The use of Python objects and dictionaries in the implementation of appliance scheduling and thermal modeling for the HEMS application resulted in additional 20 MB of memory allocation compared to the initial use case shown in Table 5.2.

Table 5.2: Overview of containers and their resource usage.

Container Type	Server	Memory Usage*	CPU%**
HEMS	`linode4`	32.32 MiB	0.01%
ISO	`linode2`	24.56 MiB	0.02%
Phoenix WebApp	`linode2`	99.97 MiB	0.62%
PostgreSQL	`post`	74.48MiB	0.18%
neo4j	`post`	887 MiB	1.26%
VerneMQ	`post`	1.516 GiB	2.33%

* Maximum value across all servers running this application.

** Percentage is with respect to the server.

CHAPTER 6
SIMULATION RESULTS

This chapter outlines the further analysis of simulation data obtained from the two use cases. The testbed was used successfully to facilitate the necessary communications and simulation administration to capture energy meter data with time-varying price rates, to configure appliances through administration shells in the CPES, and to execute reliability DR controls.

6.1 Energy Metering

Recall the in Fig. 4.7 depicted reference load and price profiles that were used as system inputs for Use Case A. With Gaussian noise added to the input, the recorded agent output closely followed the input, shown in Fig. 6.1. The HEMS agent `dev_133` and HEMS agent `dev_233` differed from each other and from the provided input by a time shift. Each agent corresponded to one docker container and as containers were started sequentially, small delays carried over throughout the simulation. Using time-based rather than interval-based events helped to prevent these shifts in Use Case B.

Time-varying prices were implemented in this example; the recorded billing instances were graphed against a hypothetical average flat rate price, shown in Fig. 6.2. The difference in the running total fluctuated depending on the amount and time of energy usage. For this agent, the energy consumption at 3am and 10pm were more cost-efficient whereas the consumption at 9am and 1pm were less cost-efficient with respect to the average daily cost. The results showed that the testbed communication and administration layers easily allow for the implementation of time-varying rate structures and energy metering. Future work could consider changes to the agent layer so that energy usage is adjusted based on forecasted and actual prices. A more sophisticated behavioral model for the HEMS agent would be needed for such purpose while all other components could remain the same.

6.2 Building Models

Use Case B implemented the in Sect. 4.4.3 described bottom-up building model with scheduled appliances and a variable HVAC controlled thermal model. The annotated graph in Fig. 6.3 for the one-day power profile showed efficacy for configuring appliance parameters

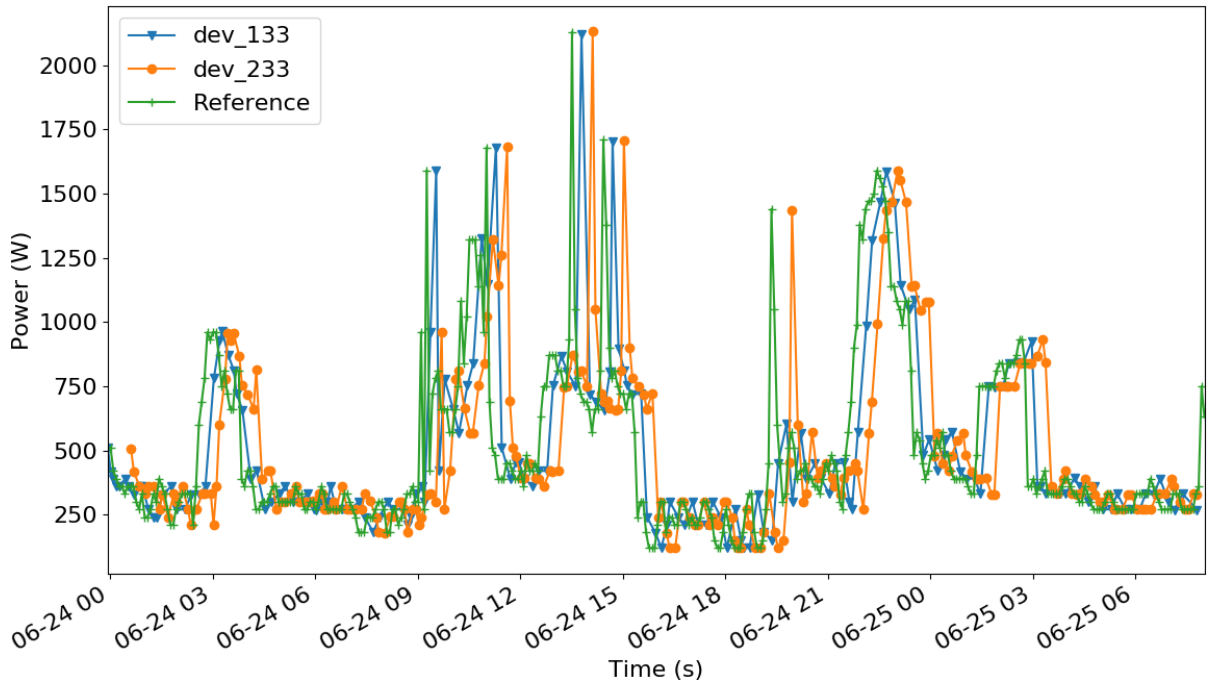


Figure 6.1: One-day power log two devices compared to the reference input at a temporal resolution of five minutes.

in the CPES model. Agents successfully queried the GDB and then scheduled appliances based on query results. The power profile reflected the power ratings of each appliance and showed the difference between fixed and variable appliances. That is, the power output of the HVAC system varied depending on the house temperature and setpoint, whereas fixed appliances consistently used what they were rated for.

The thermal response of the house is shown with respect to the ambient temperature and the overall power consumption of the house. During the off-times, the house temperature followed the ambient temperature in a first order response with a time constant of about 21 minutes. The HVAC system consumed maximum power on system startup, thus causing a rapid spike in the load curve. During steady-state operation between 11am and 1pm, the system drew $1500W$ with a steady-state error of $1°C$. At night, the house temperature dropped below the setpoint and the HVAC system did not run despite its scheduled usage. Given the presented house and control model, one could expect the load curve to follow a similar profile each day provided similar ambient temperatures and no external factors such as load shedding.

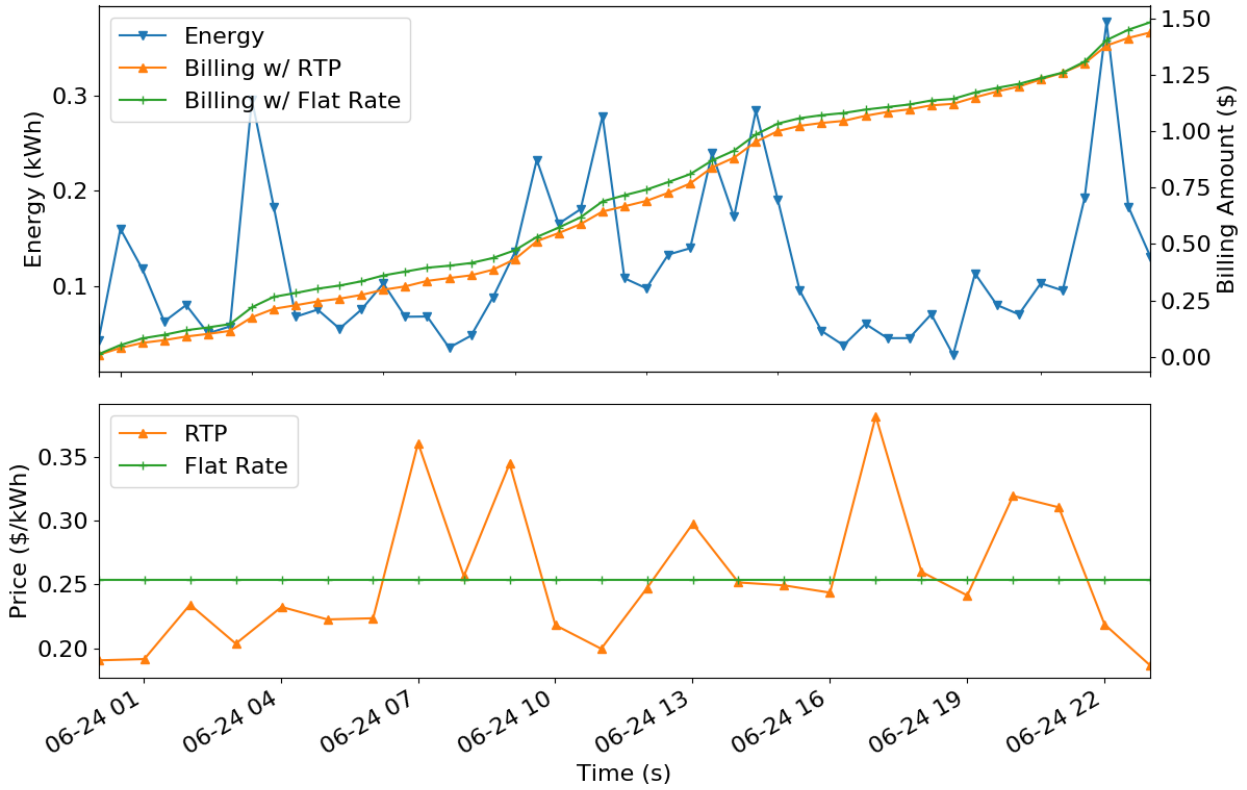


Figure 6.2: Cumulative billing for RTP and flat rate pricing are graphed with respect to half-hourly energy consumption for HEMS agent `dev_133`.

6.3 Emergency DR Program

In addition to recording power and temperatures of all agents, DLC events were also illustrated using the testbed platform. Once an event was initialized from the web interface, e.g. "shed 500kW at 10:33 am on July 1", the ISO queried all available loads at that time and sent out requests to each agent to shed its controllable load. Since in this particular event that amount requested exceeded the amount available, all available agents were requested to shed their controllable HVAC load. The macro level response of all HEMS agents is portrayed in Fig. 6.4. The response was compared to the system's macro-level behavior during the same time interval on the following day that did not have any DR events. The curve for the case with the DR event showed the power drop after the event was initiated accompanied by a temperature increase of 1.5°C during the event. After the DR event, the proportional controller regulated the temperature back down at the expense of a rise in demand. The observed spikes in demand for the HVAC system alludes to macro level behaviors that can

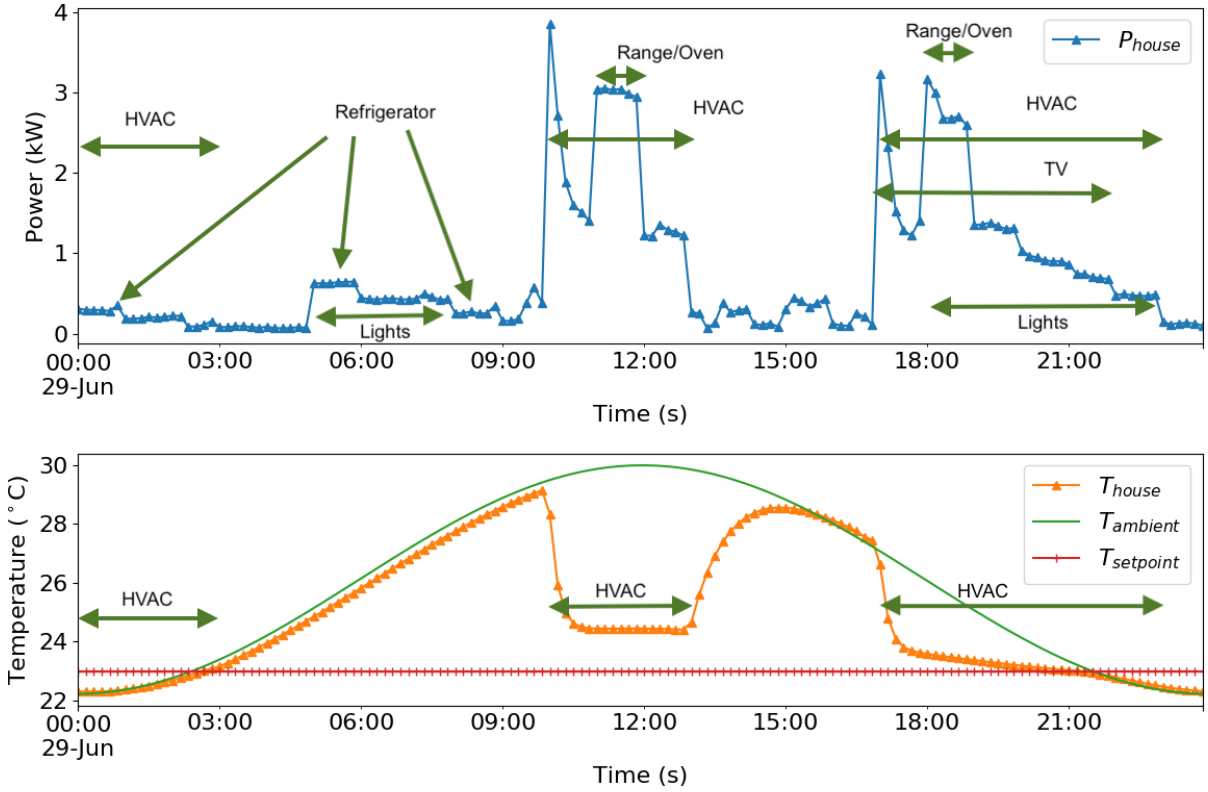


Figure 6.3: House load and temperature curves for node `dev_133`. Annotations indicate the scheduled runtime of appliances. Table 4.2 provides power ratings for each appliance. The refrigerator is only marked for the first three scheduled time slots.

be expected from autonomous agents that do not have properly implemented mechanisms to return to their default operation after a DR event. Using this testbed, smarter HEMS controllers can be quickly developed, deployed, and if proven to be effective, quickly installed on a real residential house for continued testing on this platform.

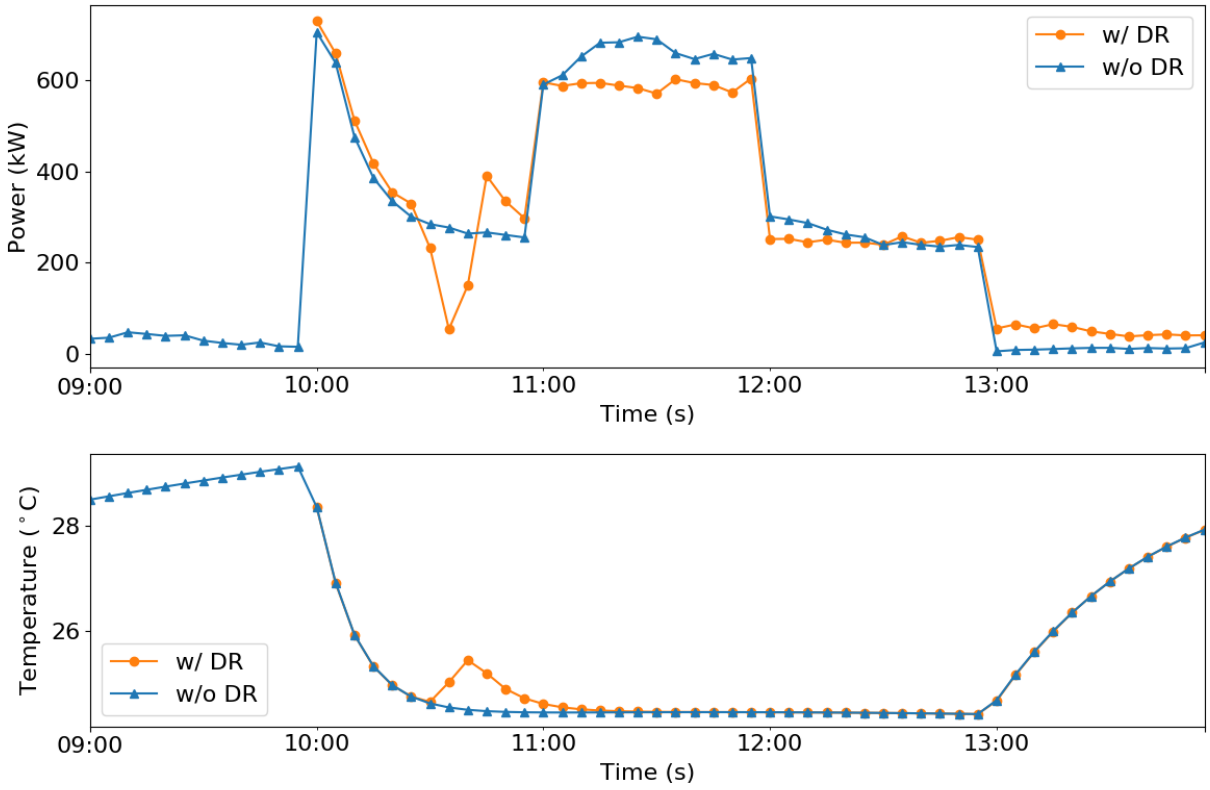


Figure 6.4: Macro level system response on a day with DR event superimposed onto a day without DR event. The ambient temperature conditions for both days were the same.

CHAPTER 7
CONCLUSION

7.1 Summary

This thesis presented the design, implementation, and application of a multi-agent testbed system. The presented system was designed to address the need for a representative platform that can help implement smart agents and to test the system level outcome of their decentralized behaviors in a Smart Grid environment under consideration of the distributed and resource-constrained nature of actual smart devices. The platform was further designed to specifically consider characteristics of residential demand and demand side management due to the ever increasing importance of consumer participation for the optimization of resource utilization in a modern energy grid with high levels of interconnected distributed renewable energy resources.

Ch. 2 presented the three-layered micro-service architecture and information flow in the distributed system. That is, a virtual electrical grid is implemented in the form of deployable agent applications that represent the different Smart Grid domains. The agent models are realizations of Smart Grid domains that trade energy in the form of metered consumption or demand response services. Functionalities of the communication and administration layers that sit on top of the agent layer enable the configuration and co-simulation of the containerized agent-based models that implement—individually or as an aggregate—antecedent research in demand side management.

The design and implementation of the architecture (Ch. 4) was presented in light of two sample use cases, that of a smart metering infrastructure and that of a DR program with direct load control in emergency situations, as described in Ch. 3. The implementation design—and thus system utility—hinges on three key aspects: a) the ability to deploy containerized applications with defined agent-level behavior on any computing platform while maintaining a standardized communication scheme, b) the ability to represent agents, their functionalities, and their relationships in a graph database, and c) the ability to capture data, visualize data, and provide interaction mechanisms for users to participate in the simulation.

Sample use case applications evidenced how each of these aspects played a role when the testbed was used in practice. In particular, the bottom-up modeling approach of residential household energy usage showed how agent-based applications can be networked, linked, and

configured using a graph database that captures each agent's parameters in virtual administrations shells using node properties (Ch. 5). This new approach to managing distributed co-simulations for complex dynamical systems enables the presented testbed to easily manage virtual (or real) agents during simulations (Ch. 6). The testbed system thus becomes a valuable tool in the rapid development of smart agents for the grid based on antecedent research and alleviates challenges in the implementation of demand side management research for the Smart Grid.

7.2 Future Work

The presented testbed platform considered building models, market structures, and agent-based strategies for consumers, system operators, and service providers. As alluded to in prior sections, a variety of DR strategies for ancillary service can currently be developed and tested through changes in the agent layer. DR strategies and simulation results for DR programs are already readily available in the vast pool of DR research; their implementation and evaluation as smart agents however remains as future work for this testbed system.

In addition to DR programs, this testbed has great potential for developing energy management agents that participate in the market. Multiple users can develop different strategies and deploy them together as agent-based applications, similar to the approach in [38] for demand side management or [42] for power generation, but with much simplified communication interface (MQTT API), deployment scheme (docker containers), and web-based administration interface. This endeavor can be further enhanced when existing power grid simulation tools, such as GridLAB-D, are tied into this platform to also consider the interaction of demand and supply at the physical transmission level.

Lastly, with much research in cyber security and distributed ledger technologies, the current agent-layer design with distributed docker applications is well-suited for implementing distributed ledgers for certified energy exchange and data management in the Smart Grid.

APPENDIX A
DOCKER RELATED CODE

A.1 Administration Layer

Graph Database

```
$ docker run --name dris-neo4j-1 --publish=7474:7474 --publish=7475:7687
 ↪  --volume=/home/holm/dris/neo4j/data:/data --volume=/home/holm/dris/neo4j/logs:/logs
 ↪  -dit --restart unless-stopped neo4j:3.3.3
```

Listing A.1: Docker command to start a new neo4j database using image version `neo4j:3.3.3`. All parameters can be adjusted.

File Server

```
FROM nginx
COPY static /usr/share/nginx/html
```

Listing A.2: Dockerfile showing the simplicity of creating the static file server container.

```
$ docker build -t nginx-file-server .
$ docker run --it --restart unless-stopped -p 8080:80 nginx-file-server
```

Listing A.3: Commands for Unix based system to build and then run the static file server. The working directory should contain the Dockerfile and the *static/* directory form List. A.2.

PostgreSQL Database

```
$ docker run --name dris-psql-1 -p 5005:5432 -e POSTGRES_USER=postgres -e
 ↪  POSTGRES_PASSWD=postgres -dit --restart unless-stopped postgres:10.2
```

Listing A.4: Docker command to start a new PostgreSQL database. Name, ports, user, and password are just examples here.

Web Application

```
FROM elixir:alpine
ARG APP_NAME=simeng_umbrella
ARG PHOENIX_SUBDIR=apps/dris
ENV MIX_ENV=prod \
    REPLACE_OS_VARS=true \
    TERM=xterm
WORKDIR /opt/app
RUN apk update \
    && apk --no-cache --update add nodejs nodejs-npm \
    && mix local.rebar --force \
    && mix local.hex --force
COPY . .
RUN mix do deps.get, deps.compile, compile
RUN cd ${PHOENIX_SUBDIR}/assets \
    && apk --no-cache add --virtual native-deps \
      g++ gcc libgcc libstdc++ linux-headers make python \
    && npm install --quiet \
    && ./node_modules/brunch/bin/brunch build -p \
    && cd .. \
    && MIX_ENV=prod mix phx.digest
RUN MIX_ENV=prod mix release --env=prod --verbose \
    && mv _build/prod/rel/${APP_NAME} /opt/release \
    && mv /opt/release/bin/${APP_NAME} /opt/release/bin/start_server
FROM alpine:latest
RUN apk update \
    && apk --no-cache --update add bash openssl-dev
ENV PORT=5005 \
    MIX_ENV=prod \
    REPLACE_OS_VARS=true
WORKDIR /opt/app
COPY --from=0 /opt/release .
EXPOSE ${PORT}
CMD ["/opt/app/bin/start_server", "foreground"]
```

Listing A.5: Dockerfile used to create the web application container that is deployable on any system.

A.2 Communication Layer

```
$ docker run --name dris-vernemq-1 -p 5883:1883 -dit --restart unless-stopped
↪  erlio/docker-vernemq
```

Listing A.6: Docker command to run a container with the `docker-vernemq` image. Administration and adjustments to the configurations can be done from within the container.

A.3 Agent Layer

```
FROM python:2-alpine
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
COPY requirements.txt /usr/src/app/
RUN apk --update add --no-cache g++
RUN pip install --no-cache-dir -r requirements.txt
ADD crontab.txt /crontab.txt
COPY entry.sh /entry.sh
RUN chmod 755 /entry.sh
RUN /usr/bin/crontab /crontab.txt
COPY ./src/app.py /usr/src/app/app.py
CMD ["/entry.sh"]
```

Listing A.7: Dockerfile used to create the Python application container. Python packages/drivers are specified in the requirements.txt file in the working directory of the Dockerfile.

APPENDIX B
PYTHON RELATED CODE

MQTT

```
#! /usr/bin/env python2

import paho.mqtt.client as mqtt
import json
from time import time, sleep

def on_connect(client, userdata, flags, rc):
    """ Callback function when client connects """
    client.subscribe("drsim/settings")
    client.subscribe("iso/rtp/+")

def on_message(client, userdata, msg):
    """ Callback function when message received """
    global simon, price, drsignal
    data = json.loads(msg.payload)
    print "Received: ", data, "   Topic: ", msg.topic

client = mqtt.Client("test-123")
client.on_connect = on_connect
client.on_message = on_message

try:
    client.connect("post.redlab-iot.net", 55100, 60)
    print "connection established"
except:
    print "connection failed"

client.loop_start()

while True:
    topic = "redlab/dr"
    payload = '{"hello": "world", "ts": '+ str(int(time())) + ' }'
    client.publish(topic, payload, qos=0, retain=False)
    print "published: ", payload, "   to: ", topic
    sleep(300)
```

```
36
37 client.loop_stop()
```

Listing B.1: Python code demonstrating the use of the `paho-mqtt` library.

```
$ python mqtt-client-demo-2.py
connection established
published:  {"hello": "world", "ts": 1529792804 }    to:  redlab/dr
Received:  {u'mode': u'start'}    Topic:  drsim/settings
Received:  {u'ts': 1529791260, u'value': 0.3944}    Topic:  iso/rtp/zone_1
Received:  {u'ts': 1529791260, u'value': 0.3976}    Topic:  iso/rtp/zone_0
published:  {"hello": "world", "ts": 1529793104 }    to:  redlab/dr
published:  {"hello": "world", "ts": 1529793404 }    to:  redlab/dr
published:  {"hello": "world", "ts": 1529793704 }    to:  redlab/dr
```

Listing B.2: Sample console output when running the demo script in List. B.1

Neo4j-Driver

```
1  import os
2  from neo4j.v1 import GraphDatabase
3
4  # devId should be set as an environment variables
5  devId = os.environ["devId"]
6
7  # connection information (replace ***** with actual user/pass)
8  uri = "bolt://post.redlab-iot.net:55097"
9  driver = GraphDatabase.driver(uri, auth=("*****", "*****"))
10
11 # function to get info based on devId
12 def get_config_from_neo4j(devId):
13     with driver.session() as session:
14         with session.begin_transaction() as tx:
15             for record in tx.run("MATCH (a:Agg) <- [:MANAGED_BY] - (h:Hems) "
16                                  "MATCH (h:Hems) - [:LOCATED_IN] -> (z:Zone) "
17                                  "MATCH (l:Loadprofile)"
18                                  "WHERE h.hemsid = {devid}"
19                                  "RETURN [a.aggid, z.name, h.appliances, l.url]",
                                     ↪  devid=devId):
20                 return record["[a.aggid, z.name, h.appliances, l.url]"]
21
22 # get the configuration info from the GDB
23 [aggId, zoneId, appliances, loadprofile]  = get_config_from_neo4j(devId)
```

Listing B.3: Python code snippet demonstrating the use of the neo4j-driver.

APPENDIX C
NEO4J RELATED CODE

```
CREATE INDEX ON :Sysop()
CREATE INDEX ON :Zone(name)
CREATE INDEX ON :Agg(aggid)
CREATE INDEX ON :Hems(hemsid)

LOAD CSV WITH HEADERS FROM "file:///neoZones_1.csv" AS csvLine FIELDTERMINATOR ','
create (z:Zone {name: csvLine.zoneName})
create (z) <- [:RTP_IN] - (:RTPrice {timestamp: timestamp(), price: 0.28})

Match (z:Zone)
create (z) <- [:PROFILE_FOR] - (:Priceprofile {url:
 ↪  "http://linode3.redlab-iot.net:8080/pjm_rt_hrl_lmps.csv"})

LOAD CSV WITH HEADERS FROM "file:///neoAggs_1.csv" AS csvLine FIELDTERMINATOR ','
create (:Agg {aggid: csvLine.aggid, aggname: csvLine.aggname, services:
 ↪  csvLine.services, zones: csvLine.regions})

MATCH (a:Agg)
UNWIND split(a.zones, ',') AS zone
MATCH (z:Zone {name: zone})
CREATE (a) - [:OPERATES_IN] -> (z)

LOAD CSV WITH HEADERS FROM "file:///neoDevs_1.csv" AS csvLine FIELDTERMINATOR ','
match (z:Zone {name: csvLine.zone})
match (a:Agg {aggid: csvLine.aggid})
create (h:Hems {hemsid: csvLine.devid, appliances: csvLine.appliances})
create (h) - [:LOCATED_IN] -> (z)
create (h) - [:MANAGED_BY] -> (a)
create (h) <- [:MEASURED_BY] - (:Emeasurement {name: "emeas_"+csvLine.devid, avgp:
 ↪  null, vrms: null, pf: null, ts: timestamp()})
create (h) <- [:REPORTED_BY] - (:Derna {name: "derna_"+csvLine.devid})

create (:Loadprofile {url:
 ↪  "http://linode3.redlab-iot.net:8080/20180503_recorded_data.csv"})
```

Listing C.1: Collection of Cypher commands used to build and populate the initial database from provided csv files. The shown commands need to be executed individually.

APPENDIX D
INTERFACE

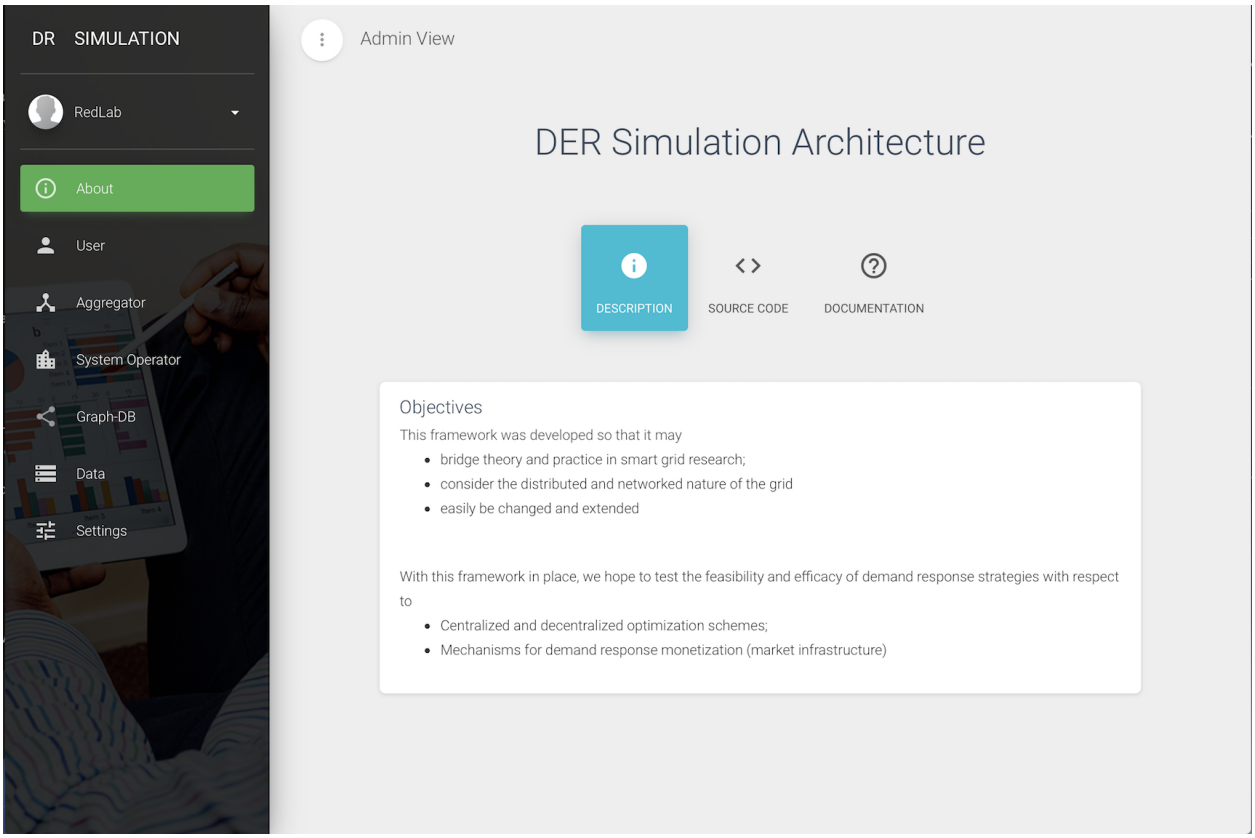


Figure D.1: Homepage of the interface with introductory text. The application contains three dashboards for the user (home), aggregator, and system operator. The data tab allows the query and download of data and the settings tab provides controls to configure and run the simulation.

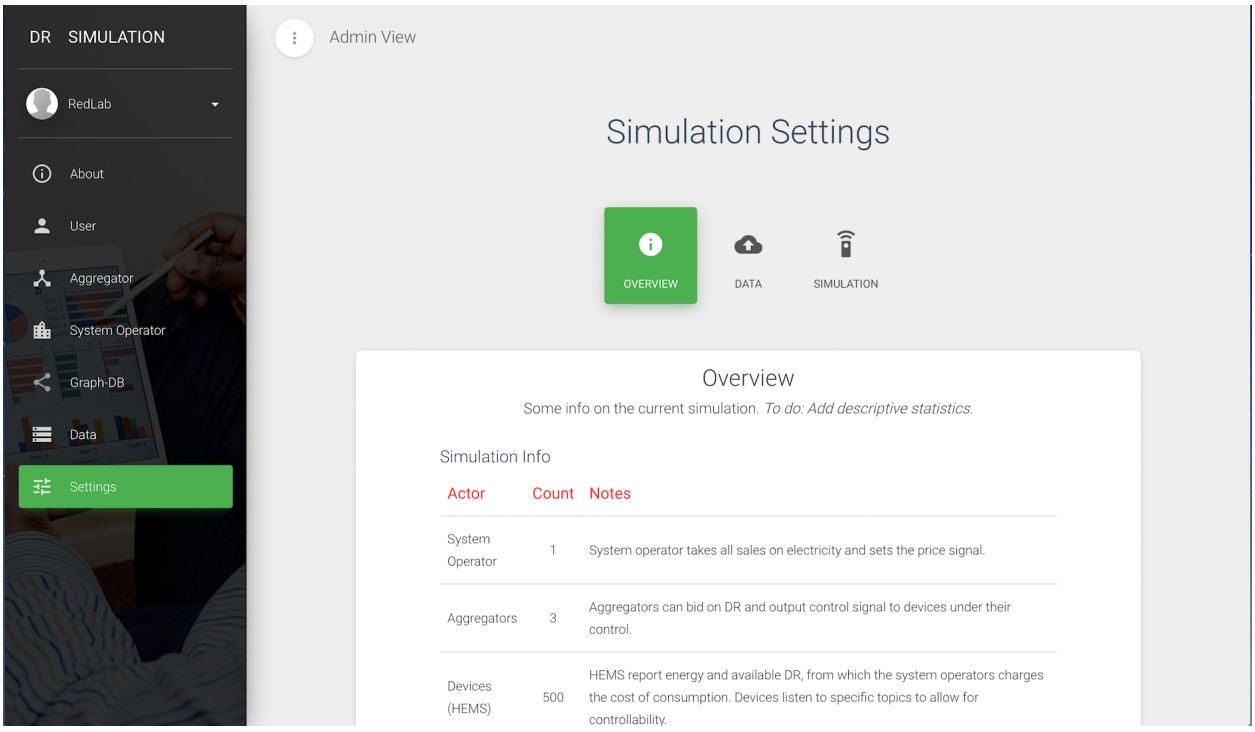


Figure D.2: The settings page provides some general information and then has tabs for data upload and simulation controls (start/stop/pause).

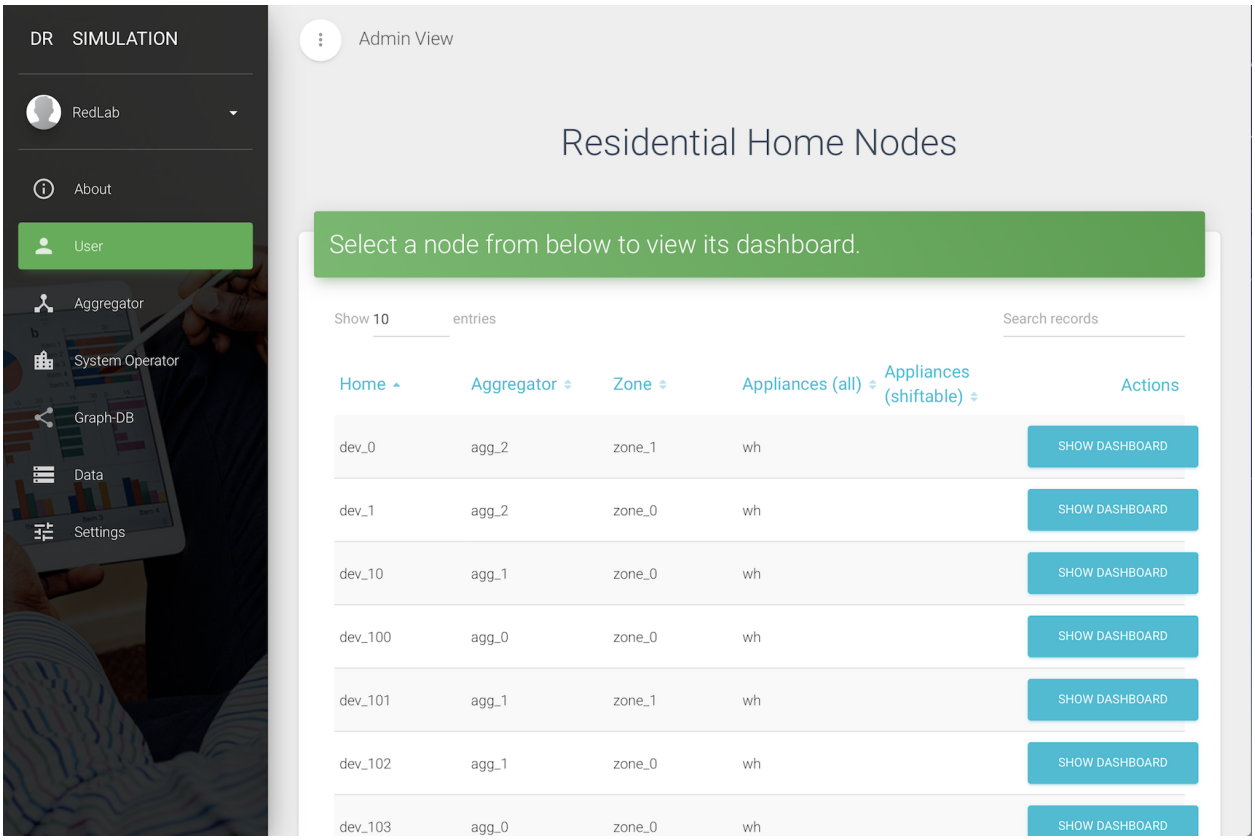


Figure D.3: Data tables give the user the option to search for nodes and then look at their respective interface.

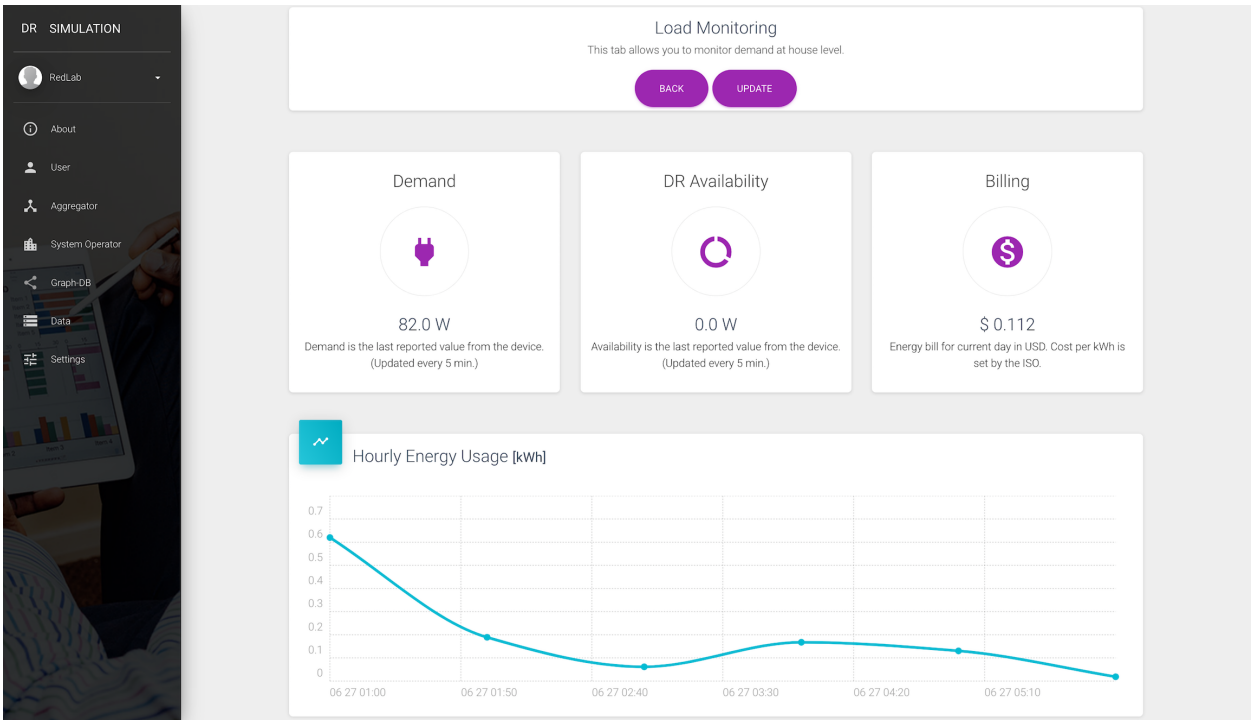


Figure D.4: Monitoring dashboard for home nodes. The demand value is the latest power measurement submitted and the billing value the cumulative charge for a given day (calculated based on time-varying prices and the UTC timezone). Hourly energy usage is reported for the past 24 hours in a line chart. At the time of this capture, the HVAC system was not scheduled to run.

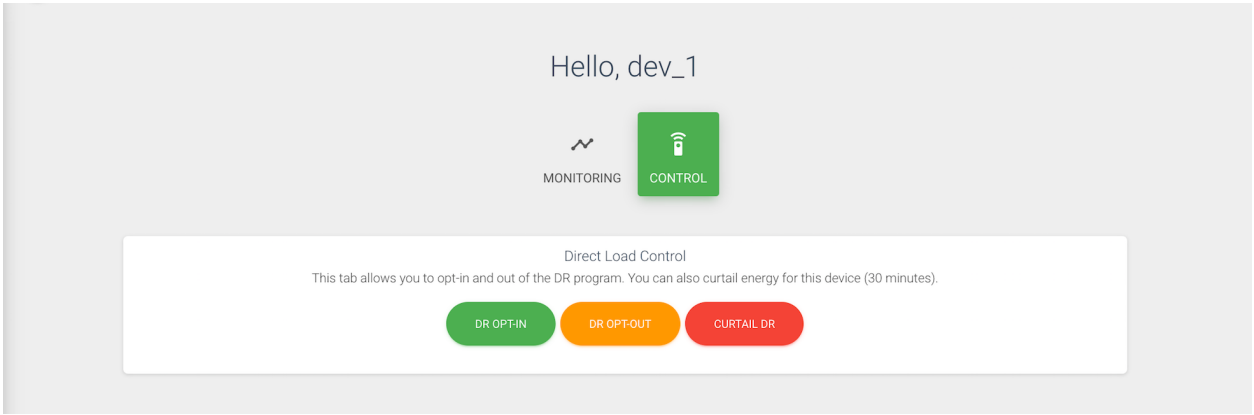


Figure D.5: Control interface for home nodes. The user can choose to opt-out of the DR program or to also manually curtail energy usage.

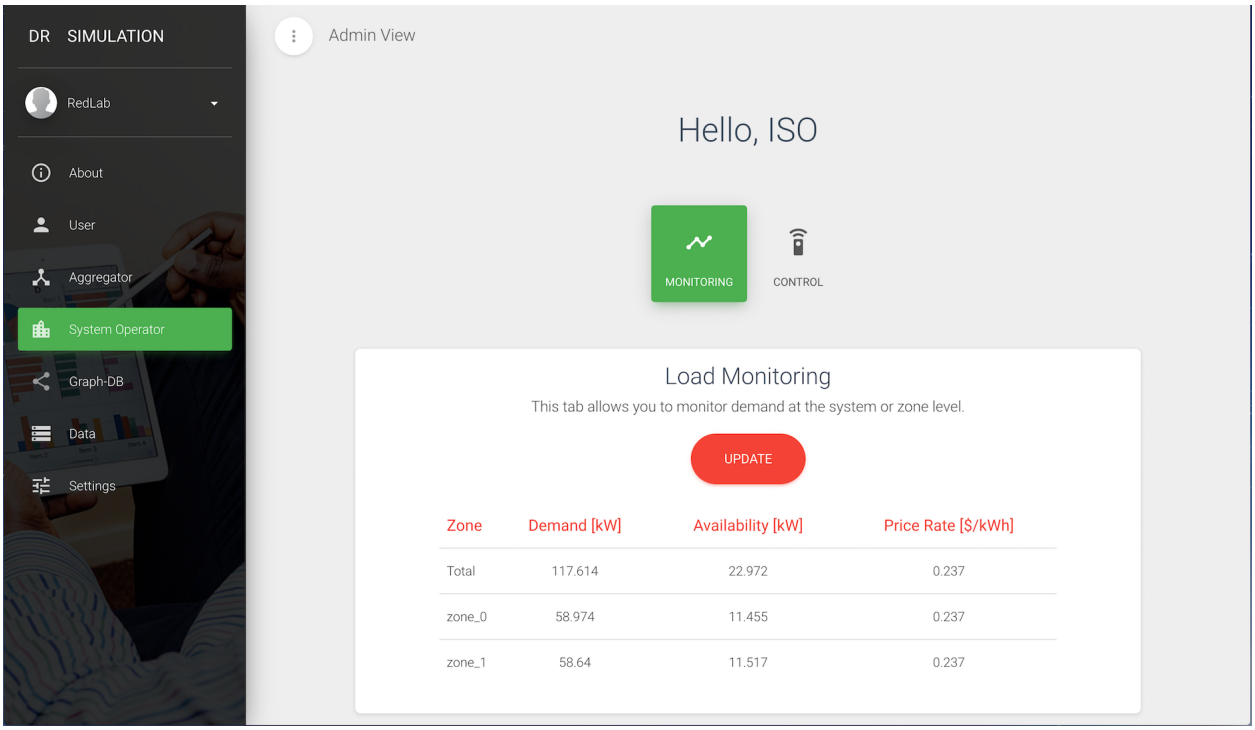


Figure D.6: Interface for the system operator to monitor total and by zone aggregated load, availability, and price data.

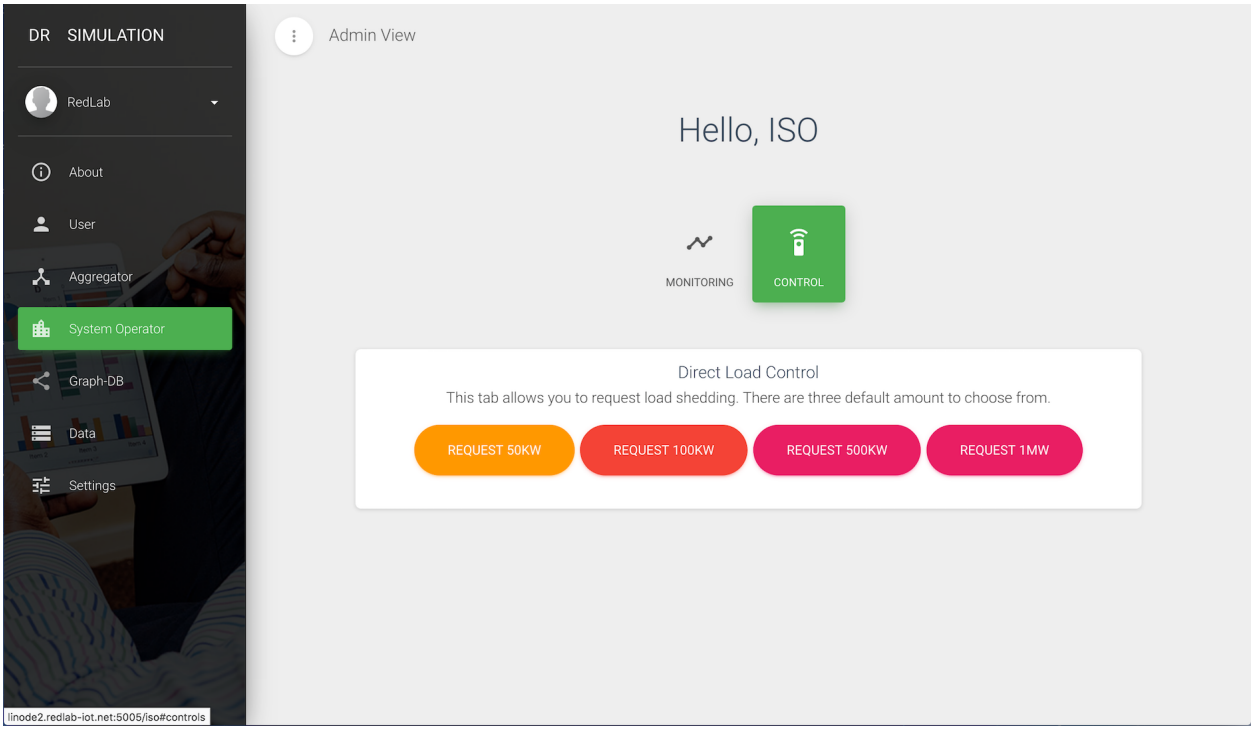


Figure D.7: Interface for the system operator with options to curtail 50, 100, 500, or 1000 kW. Once the button is clicked, an MQTT message is published to the `drsim/events` topic, which the ISO can subscribe to.

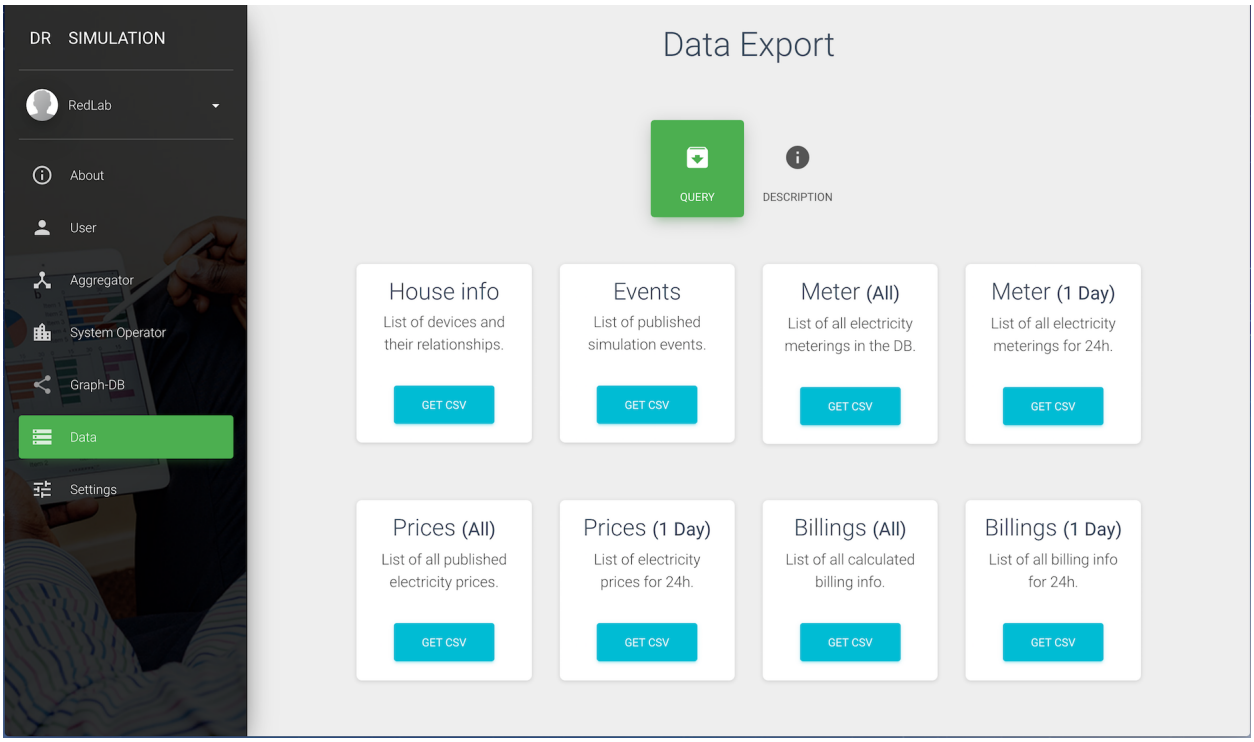


Figure D.8: Interface with predefined options for querying data from the database as csv files.

BIBLIOGRAPHY

[1] A. Chiu, A. Ipakchi, A. Chuang, *et al.*, "Framework for integrated demand response (DR) and distributed energy resources (DER) models," 2009.

[2] Electric Power Research Institute, "Report to NIST on the smart grid interoperability standards roadmap," tech. rep., 2009.

[3] Docker Inc., "What is a Container." `https://www.docker.com/what-container`, accessed 2018-11-05.

[4] AWS, "Pub/Sub Messaging," 2018. `https://aws.amazon.com/pub-sub-messaging`, accessed 2018-06-15.

[5] International Electrotechnical Commission, "What is a Smart Grid?." `http://www.iec.ch/smartgrid/background/explained.htm`, accessed 2018-06-04.

[6] E. M. Lightner and S. E. Widergren, "An orderly transition to a transformed electricity system," *IEEE Transactions on Smart Grid*, vol. 1, no. 1, pp. 3–10, 2010.

[7] U.S. Department of Energy, "The Smart Grid: An introduction," tech. rep., 2008.

[8] S. Succar and R. Cavanagh, "The promise of the Smart Grid: Goals, policies, and measurement must support sustainability benefits," 2012.

[9] P. Siano, "Demand response and smart grids - A survey," *Renewable and Sustainable Energy Reviews*, vol. 30, pp. 461–478, 2014.

[10] S. Borenstein, "Time-varying retail electricity prices: Theory and practice," in *Electricity deregulation: Choices and challenges* (Griffin J. M. and S. Puller, eds.), University of Chicago Press, 2009.

[11] P. L. Joskow and C. D. Wolfram, "Dynamic pricing of electricity," *American Economic Review: Papers & Proceedings*, vol. 102, no. 3, pp. 381–385, 2012.

[12] A. E. Kahn, *The economics of regulation: principles and institutions*. New York: John Wiley & Sons, volume 1 ed., 1970.

[13] M. Motalleb, A. Eshraghi, E. Reihani, H. Sangrody, and R. Ghorbani, "A game-theoretic demand response market with networked competition model," *2017 North American Power Symposium*, no. July, 2017.

[14] D. Chassin, J. Fuller, and N. Djilali, "GridLAB-D: An agent-based simulation framework for smart grids," *Journal of Applied Mathematics*, pp. 1–12, 2014.

[15] K. P. Schneider, J. C. Fuller, and D. Chassin, "Analysis of distribution level residential demand response," *IEEE/PES Power Systems Conference and Exposition*, pp. 1–6, 2011.

[16] K. Anderson, J. Du, A. Narayan, and A. E. Gamal, "GridSpice: A distributed simulation platform for the smart grid," *2013 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*, vol. 10, no. 4, pp. 1–5, 2013.

[17] M. Thornton, H. Smidt, V. Schwarzer, M. Motalleb, and R. Ghorbani, "Internet-of-Things hardware-in-the-loop simulation testbed for demand response ancillary services," in *Materials for Energy, Efficiency and Sustainability, TechConnect Briefs 2017*, pp. 66–69, TechConnect, 2017.

[18] H. Smidt, M. Thornton, and R. Ghorbani, "Smart application development for IoT asset management using graph database modeling and high-availability web services," in *HICSS-51*, 2018.

[19] H. Smidt, M. Thornton, and R. Ghorbani, "Network coordinated evolution: modeling and control of distributed systems through on-line genetic PID-control optimization," 2018.

[20] CEN/CENELEC/ETSI Joint Working Group on Standards for Smart Grids, "CEN-CENELEC-ETSI Smart Grid Coordination Group: Smart Grid Reference Architecture," no. November, pp. 1–107, 2012.

[21] VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik, "Status Report; Reference Architecture Model Industrie 4.0 (RAMI4.0)," tech. rep., 2015.

[22] U.S. Energy Information Agency, "Nearly half of all U.S. electricity customers have smart meters," 2017. `https://www.eia.gov/todayinenergy/detail.php?id=34012`, accessed 2018-06-22.

[23] A.-L. Barabasi, *Network science*. Cambridge, United Kingdom: Cambridge University Press, 2016.

[24] M. E. Newman, *Networks: An Introduction*. Oxford University Press, 2010.

[25] D. J. Watts and S. H. Strogatz, "Collective dynamics of'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[26] G. A. Pagani and M. Aiello, "The power grid as a complex network: a survey," *Physica A: Statistical Mechanics and its Applications*, vol. 392, no. 11, pp. 2688–2700, 2013.

[27] R. Albert, I. Albert, and G. L. Nakarado, "Structural vulnerability of the North American power grid," *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, vol. 69, no. 2 2, pp. 1–4, 2004.

[28] Z. Wang, A. Scaglione, and R. J. Thomas, "Electrical centrality measures for electric power grid vulnerability analysis," *Proceedings of the IEEE Conference on Decision and Control*, pp. 5792–5797, 2010.

[29] E. Bompard, R. Napoli, and F. Xue, "Analysis of structural vulnerabilities in power transmission grids," *International Journal of Critical Infrastructure Protection*, vol. 2, no. 1-2, pp. 5–12, 2009.

[30] I. Sayeekumar, K. Ahmed, P. Karthikeyan, K. Sah, and U. Raglend, "Graph theory and its applications in power systems -A Review," pp. 154–157, 2015.

[31] J. Quirós-tortós and V. Terzija, "A Graph Theory Based New Approach for Power System Restoration," in *IEEE Grenoble Conference*, 2013.

[32] S. Dutta and T. Overbye, "A graph-theoretic approach for addressing trenching constraints in wind farm collector system design," *2013 IEEE Power and Energy Conference at Illinois, PECI 2013*, pp. 48–52, 2013.

[33] C. McCord, B. Tate, and J. Valim, *Programming Phoenix - Productive, Reliable, Fast*. Pragmatic Programmers, LLC, 2016.

[34] Plataformatec, "Umbrella projects." `https://goo.gl/SVsFTW`, accessed 2018-06-06.

[35] Octavo Labs AG, "VerneMQ - A MQTT broker that is scalable, enterprise ready, and open source." `https://vernemq.com/intro/index.html`, accessed 2018-05-05.

[36] Octavo Labs AG, "Running VerneMQ using Docker." `https://vernemq.com/docs/installation/docker.html`, accessed 2018-02-05.

[37] Octavo Labs AG, "Authentication and authorization using a database." `https://vernemq.com/docs/configuration/db-auth.html`, accessed 2018-06-14.

[38] E. Santi, A. Wunderlich, H. Abdollahi, and S. Arrua, "FEEDER power flow control experience," 2018.

[39] J. Ziegler and N. Nichols, "Optimum Settings for Automatic Controllers," *Transactions of the AS E*, vol. 64, pp. 759–768, 1942.

[40] A. Jayachitra and R. Vinodha, "Genetic algorithm based PID controller tuning approach for continuous stirred tank reactor," *Hindawi Publishing Corporation Advances in Artificial Intelligence*, vol. 2014, pp. 1–8, 2014.

[41] M. H. Amaran, N. A. M. Noh, M. S. Rohmad, and H. Hashim, "A comparison of lightweight communication protocols in robotic applications," *Procedia Computer Science*, vol. 76, no. Iris, pp. 400–405, 2015.

[42] J. Contreras, A. J. Conejo, S. Member, S. D. Torre, S. Member, and M. G. Muñoz, "Power engineering lab : Electricity market simulator," *IEEE Transactions on Power Systems*, vol. 17, no. 2, pp. 223–228, 2002.