

A Cloud-based Framework for Smart Grid Data, Communication and Co-simulation

by

Gabriel Adeyemo

Bachelor of Science, Landmark University, 2018

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s): Kenneth B. Kent, PhD, Computer Science
Examining Board: Weichang Du, PhD, Computer Science, Chair
Kalikinkar Mandal, PhD, Computer Science
Julian Meng, PhD, Electrical and Computer Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

October, 2021

© Gabriel Adeyemo, 2021

Abstract

Renewable energy has caused rapid advancements in electric power systems. The advanced grid is a smart grid with information and communication technologies and bi-directional flow of information. Data in a smart grid aligns with the characteristics of big data. Choosing the most efficient technology to manage data in the grid (real and/or simulation) is crucial to the performance of the grid. This project explores a framework that supports large scale power and network co-simulation and manages communication and data in smart grid co-simulation, real world smart grid systems and a combination of both using message-oriented middleware and cloud technologies. We designed and implemented a framework with RabbitMQ, Apache Kafka, OpenDSS, OMNeT++, Apache Spark, Docker and Kubernetes. We evaluate our implementation on accuracy, scale and usability with three applications including a demand-response application based on logistic regression. The results of our evaluation meet the goals defined for the research thesis.

Acknowledgements

I would like to thank my supervisor, Dr. Kenneth B. Kent, from the University of New Brunswick (UNB), for his patience, guidance and support, Stephen MacKay (UNB) for his feedback and continuous effort on improving my technical writing and Jonas Fernandes and other members of the Atlantic Grid project from UNB Electrical and Computer Engineering Department for their contributions to this thesis.

A special thanks to my family and friends in Canada and beyond for their endless support.

I would like to thank the Centre for Advanced Studies — Atlantic (CASA) for a resourceful and conducive research environment. In addition, I would like to thank my colleagues from CASA for their valuable help.

Lastly, I would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program.

Table of Contents

Abstract	ii
Acknowledgments	iii
Table of Contents	iv
List of Tables	viii
List of Figures	ix
Abbreviations	xi
1 Introduction	1
2 Background and Related Work	4
2.1 Smart Grid	4
2.2 Smart Grid Simulations	5
2.2.1 Power Dynamic System	5
2.2.2 OpenDSS	7
2.2.3 Communication Simulation	7
2.2.4 OMNeT++	8
2.2.5 Co-simulation	9
2.3 Big Data in Smart Grid	11
2.3.1 Data Transmission	12
2.3.1.1 AMQP	12

2.3.1.2	MQTT	13
2.3.2	Data Communication Middleware	13
2.3.2.1	RabbitMQ	14
2.3.2.2	Data Format	15
2.3.2.3	Schema Evolution	16
2.3.3	Data Aggregation and Analysis	17
2.3.3.1	Apache Kafka	17
2.3.3.2	Apache Spark	18
2.3.3.3	Logistic Regression	19
2.3.4	Data Storage	20
2.3.4.1	MongoDB	21
2.3.5	Security	21
2.3.5.1	Point-to-point and End-to-end Security	22
2.3.5.2	Transport Layer Security and Secure Sockets Layer	22
2.3.5.3	Asymmetric Encryption	23
2.3.5.4	RSA Encryption	23
2.3.5.5	Key Management	24
2.4	Cloud-based Smart Grid Management	24
2.4.1	Containerization and Docker	25
2.4.2	Kubernetes	25
2.5	Related Work	26
3	Motivation	30
4	Design and Implementation	33
4.1	Co-simulation	33
4.1.1	Time Synchronization	34
4.1.2	Data Exchange Middleware	36

4.2	Data management	36
4.2.1	Data Transmission	38
4.2.2	Data Communication	39
4.2.3	Data Aggregation	40
4.2.4	Data Analytics	41
4.2.5	Data Storage	41
4.2.6	Data Security	42
4.3	Cloud-based Scaling	44
4.4	Implementation	45
4.4.1	Hardware Environment	45
4.4.2	Software Environment	46
4.4.3	Message Structure	46
4.4.4	Micro-services, Supporting Libraries and Modules	48
4.4.5	Cloud Deployment and Supporting Tools	51
5	Performance Evaluation	53
5.1	Performance Metrics	53
5.2	Experiment	54
5.2.1	Test Case IEEE 13 Node Circuit	55
5.2.2	A: Simple Voltage Monitoring	55
5.2.3	B: Wide Area Monitoring Network with Varying Generation Rates	58
5.2.3.1	Scaling	64
5.2.3.2	Message Size	64
5.2.4	C: Cloud-based Demand-Response and Predictive Control	66
5.2.4.1	Modelling the Logistic Regression Model	68
5.2.4.2	Results	69
5.3	Summary	70

6	Conclusions and Future Work	72
6.1	Summary	72
6.2	Limitations and Future Work	73
	Bibliography	84
A	Containerization Configuration and Deployment Details	85
A.1	Docker Files	85
A.2	Deployment Files	86
B	Simulation Configuration for Simple Application	92
B.1	Configuration for OmNeT++ for Simple Application	92
B.2	OpenDSS Modifications to IEEE 13-bus for Simple Application . . .	93
B.3	Configuration for OmNeT++ for Wide Area Network	93
C	Configuration for Demand-Response Application	95
C.1	Modifications to IEEE 13-bus for Demand Response Application . . .	95
C.2	Logistic Regression Dataset Generation	96
	Vita	

List of Tables

2.1	Traditional grid vs. Smart grid [24]	5
2.2	Related Work on Co-simulation	29
4.1	Middleware options and support for co-simulation framework requirements.	37
4.2	A comparison of Apache Spark and Map Reduce for big data analytics	41
4.3	CASA Kubernetes Cluster	45
4.4	Software Environment	46
5.1	A sample dataset for a minimum and maximum threshold logistic regression model	69

List of Figures

2.1	An Illustration of power system dynamic simulation [40]	6
2.2	An Illustration of communication network simulation [40]	8
2.3	High-level layered architecture of ICT services in smart grid [58].	11
3.1	Message Latency of 10,000 JSON messages using database polling and publish-subscribe middleware.	32
3.2	Write time of 10,000 JSON messages using database polling and pub- lish subscribe middleware	32
4.1	Combination of Time-stepped and Primary-secondary Synchronization	35
4.2	A simple co-simulation with OpenDSS and OMNeT++	37
4.3	A simple co-simulation with OpenDSS and OMNeT++, RabbitMQ as middleware for intra and external data exchange.	37
4.4	Kafka as a tool for aggregation	40
4.5	The role of Apache Spark for Analytics	42
4.6	A secure and scalable framework for data, communication and co- simulation	43
4.7	A kubernetes cloud-based translation of the framework	44
4.8	Micro-service Implementation of Framework for Simulation	50
4.9	Micro-service Implementation of Framework for Real Device	50
5.1	Without Co-simulation: Voltage Reading at 15 minute intervals for 24 hours	56

5.2	Without vs With Co-simulation(Sequential Scheduler): Voltage Reading at 15 minute intervals for 24 hours	57
5.3	Co-simulation with pausing and Sequential Scheduler: Voltage Reading at 15 minute intervals for 24 hours	59
5.4	Wide Area Monitoring Network [11]	60
5.5	Time required to simulate one minute of simulation time with Message-oriented and Database Middleware	62
5.6	Message Latency of message with varied messages per second	63
5.7	Average Latency of Message-oriented Middleware (MOM) with JSON and Protocol buffer with varied messages per second	63
5.8	Average real time to simulate one minute with varying number of running instances at 10 samples a second	65
5.9	Effect of number of values in a message payload on message size	67
5.10	Effect of message size on simulation time	67
5.11	A modified IEEE 13-bus feeder with generator and a monitor	69
5.12	Voltage reading at Load 611 for 300 seconds without communication co-simulation	70
5.13	Voltage at Monitor with cloud-based demand-response with analytics section of co-simulation framework	71

List of Symbols, Nomenclature or Abbreviations

SG	Smart Grid
ICT	Information and Communication Technology
gcd	Greatest Common Divisor
RSA	Rivest, Shamir, Adleman Encryption
TLS	Transport Layer Security
SSL	Secure Sockets Layer
MOM	Message Oriented Middleware
AMQP	Advanced Message Queuing Protocol
MQTT	Message Queue Telemetry Transport

Chapter 1

Introduction

Electric power generation, distribution and demand management are rapidly changing due to advances in renewable energy sources and devices. In the traditional grid, information flow is one sided, control is centralized and most connected devices and agents, especially the demand, are passive. However, in a smart grid, electricity and information flow is bi-directional, energy sources are decentralized, the demand is active, and control is distributed. The existing power networks require significant changes to accommodate these advancements. These improvements include the underlying information and communication technology (ICT) infrastructure, data communication networks of connected smart meters and sensors on low- and medium-voltage lines, devices, and substations [38].

The main component of the smart grid is the application of information and communication technology (ICT) to power grids. This highlights two things. First, smart grids are complex systems that consist of various interdependent domains. The performance of the entire grid depends on the joint performance of the connected domains and of their interactions. Therefore, smart grid system designs should be jointly verified and validated through simulations before implementation [48]. While it is impractical to simulate these tests on real systems, available simulators for a do-

main may not support the simulation of other domains. For example, power system simulators do not support modelling communication infrastructures and vice versa. Constructing a new simulator that supports the simulation of both power system dynamics and communication networks is time consuming and expensive [38]. Co-simulation is a less expensive alternative. It is a technique where different simulation environments are combined to create a collaborative system with data exchange between simulators [11]. A co-simulation framework enables simultaneous simulation of power system dynamics and communication networks (and other domains) with less time and cost. However, the main challenges of co-simulation are time synchronization of different simulators, efficient data exchange and time ratio—the simulation time relative to wall clock time.

Second, the introduction of a bi-directional flow of information and more connected data active devices imply a spate of data to be managed across the smart grid. This large amount of data conforms to the characteristics of Big Data, the 4Vs: volume, velocity, veracity, and variety [19]. Additionally, the data could provide a fifth ‘V,’ value. Smart grid data can provide valuable insights useful to both the consumers and utilities companies. However, secure management of data sources, integration, aggregation, analytics, and storage could be challenging, due to the data being dynamic and in large volume. Also, the data is in various forms and mostly requires real-time processing for reactive control and monitoring. To properly handle smart grid data, a system must be secure, scalable, highly available, and have mechanisms for fast processing, storage, and retrieval.

This research explores a cloud-centric scalable framework to support three functions: to efficiently handle data and communication in a smart grid co-simulation and in real world smart grid systems; to enable communication between simulation and real systems; and to securely manage the aggregation, analytics, and storage of big data in smart grid systems. To this end, the thesis reviews the use of different communication

technologies and techniques. Further, it provides a test case implementation of the framework and the results of an evaluation of the framework over the test case.

Chapter 2

Background and Related Work

This section provides a brief introduction to some important concepts relevant to understanding this research. Further, it also highlights some of the existing research in the areas discussed.

2.1 Smart Grid

A smart grid is an advanced electrical grid that incorporates information and communication infrastructure and technologies to accommodate renewable energy systems and improve reliability, efficiency and flexibility of electricity generation, distribution, and utilization. The European Union defines smart grid as an electricity network that can intelligently integrate the behavior and actions of all users to ensure sustainability [72]. The United States Department of Defense defines the smart grid as a power grid that uses digital technology to improve reliability, security, and efficiency of the electricity system [72]. A smart grid can also be considered as a modern electric power grid infrastructure for enhanced efficiency and reliability through automated control, high-power converters, modern communications infrastructure, sensing and metering technologies, and modern energy management techniques based on the optimization of demand, energy, and network availability [28]. The main theme across the various

Table 2.1: Traditional grid vs. Smart grid [24]

Traditional grid	Smart grid
Electromechanical	Digital
One-way Communication	Two-way Communication
Centralized generation	Distributed generation
A few sensors	Sensors throughout
Manual monitoring	Self-monitoring
Manual restoration	Self-healing
Failure and Blackouts	Adaptive and Islanding
Limited Control	Pervasive control
Few customer choices	Many customer choices

definitions of smart grid is the use of information and communication technology (ICT) to improve its operations. The other highlights of the various definitions are the improvements of the smart grid over the traditional grid, like renewable energy sources and devices. These improvements are summarized by Farhangi [24] in Table 2.1.

2.2 Smart Grid Simulations

The smart grid is complex because it comprises many connected systems. It is ideal and common to test smart grid components and systems by simulation. Common types of simulations include power flow analysis, fault study analysis, harmonic analysis and quasi-static time-series analysis. Each simulation may test different combinations of the domains of the smart grid. This section describes how two domains of the smart grid work and are simulated.

2.2.1 Power Dynamic System

Power system dynamic simulation is commonly modeled as a continuous time system simulation; simply put, it is time-driven. In a continuous time system, the system

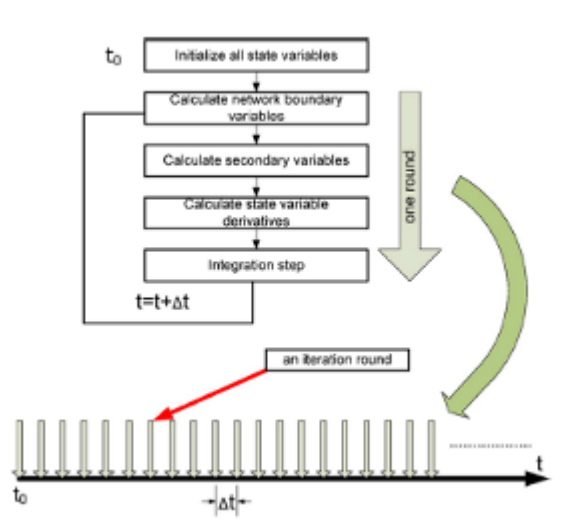


Figure 2.1: An Illustration of power system dynamic simulation [40]

state variables change in a continuous manner with respect to time. Typically, the system dynamics are a set of differential equations in which the transitions between continuous state variables are defined [40]. These differential equations can be solved analytically for simple cases, or with numerical algorithms for more general cases [38]. In numerical algorithmic simulations, the system behavior is captured using an iterative process. One loop of the system consists of the calculation of network boundary variables. The next step is the calculation of secondary variables. This loop continues until the simulation reaches a preset stop time. If the simulation loop is expanded on a time axis, a sequence of discrete iteration rounds can be found with small time intervals. Figure 2.1 shows the description of a power dynamic system. This sequence shows that a continuous time system is numerically solved in a discrete manner [40]. There are many tools for modelling and simulating the power aspects of the smart grid. Vogt et al. provide a survey of some of these tools. The most common of these tools are OpenDSS, MATLAB, PowerFactory, and GridLab-D [68].

2.2.2 OpenDSS

Open Distribution System Simulator (OpenDSS) is an open-source simulation tool for power distribution systems [38]. It is a differential equation solver that solves harmonics and dynamics based on power generation and load. This well-documented simulator allows the modeling of several load types, photovoltaic systems, on-load tap changers, energy storage systems, and other distribution network components [66], [22]. OpenDSS allows a user to define circuits, specify and execute controls on circuit elements, configure simulation parameters and visualize and/or export results to files. It supports time power flow simulation in various configurable time steps. OpenDSS provides a COM server interface for applications in other languages like C++, Python, C-Sharp, and MATLAB. This feature enables external applications to control the simulation and execute commands and other functions over elements in the circuit. The simulator supports most common power simulations such as power flow analysis, harmonic analysis and fault study [22].

2.2.3 Communication Simulation

As an essential part of a smart grid, data communication is commonly modelled and tested. Communication network simulation employs a discrete event-driven technique. Discrete event-driven simulation is suitable for systems whose state is only subject to change due to discrete events. In this simulation, the occurrence of events is stochastic relative to time [40]. An event could be the sending of a message or a packet across the network as illustrated in Figure 2.2. The communication network models are solved by computational loops, which process events in an event queue. Each event is processed by executing relevant actions, where some actions may trigger other events. Events and the event queue are managed by an event scheduler. The scheduler maintains the queue of events against a system or simulation timestamp and determines how events are processed. For example, a sequential scheduler

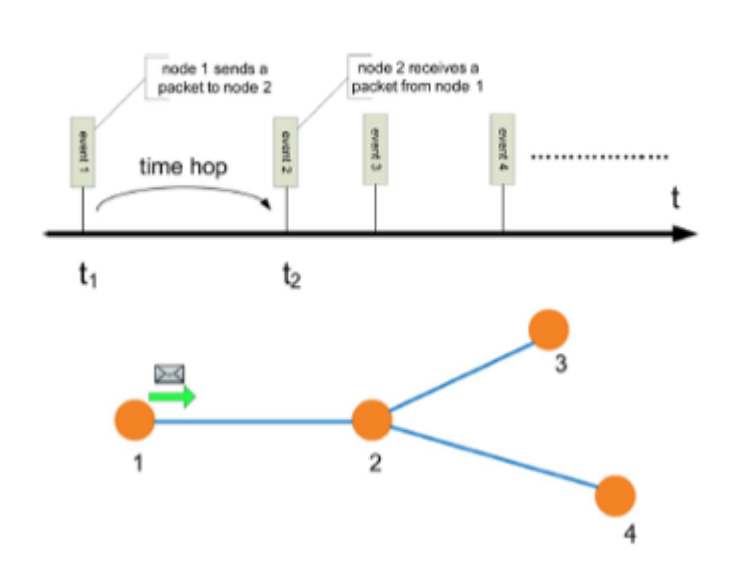


Figure 2.2: An Illustration of communication network simulation [40]

processes events in the queue one after another, while a real time scheduler synchronizes the processing of events with the wall clock time. Simulation is stopped when the event queue is empty (all scheduled events have been processed) or a certain computation time limit is reached. A simulation could also be programmed to stop when a specific event occurs. OMNeT++, NS-2, NS-3, OPNET and NetSim are commonly used to model communication networks [68].

2.2.4 OMNeT++

OMNeT++ is an open-source, modular C++ discrete-event simulation library and framework, primarily for modelling communication networks, multiprocessors and distributed systems [47, 67]. It provides support for modelling various protocols in the standard Internet and related suites. This includes support for TCP, IP, Ethernet, Wi-Fi, and several commonly used networking applications [11]. A model in OMNeT++ consists of modules. Modules communicate using messages sent over connections called channels, which can be configured with the delays and other properties of real network connections. Simple modules are reusable and can be combined

to form compound modules. The modular structure of OMNeT++ allows building supporting libraries like INET that extend the base protocols. This also allows customization of existing or building new extensions to interface with other simulation systems, hardware or software. In addition, it provides facilities for statistical analysis and a graphical user interface.

2.2.5 Co-simulation

Commercially available power system simulators lack support for modeling communication infrastructures and vice versa. Therefore, one of three approaches presented by Li et al. [38] could enable a simulation of power and communication systems. The first option is to build new software capable of modeling both domains. This option is expensive and time consuming. Another option is to extend the functionalities of individual simulators to enable simulation of the other domain. However, the resulting tools can only simulate simple scenarios and if the power system dynamics or the communication networks become too complex; they are no longer efficient [59]. The third and most viable option involves the integration of existing independent tools designed to simulate a particular domain, to form a co-simulation tool for a smart grid [38]. Power simulators are time-driven, while network simulators are event driven, the most crucial issue is a mechanism to connect and efficiently synchronize data, time, and interactions between the simulators. A successful simulator should solve the challenges of a common data model that includes components of all domains and their interconnections and choice of a good co-simulation time step [43]. W. Li et al. [38] and M. Mirz et al. [43] highlight three basic synchronization techniques:

- Time-stepped Synchronization

In this approach, each domain-specific simulator runs independently. The simulators are paused and synchronized at a specified time interval for interaction.

During simulation, all events requiring interaction, such as data request or control command execution, are stored and resolved at the next synchronization. This approach is fast; however, choosing an optimal time step is important. A large time-step accumulates many errors before the synchronization point hence the system performs poorly for time critical events, and if the steps are too small, the simulators are paused many times without need for interactions. This problem is known as the time accumulation error, and a potential solution is to dynamically adjust the time synchronization [2, 26, 15, 25].

- Master/Slave Synchronization

During co-simulation, either the power system simulator or the communication network simulator could be the Master. The simulator named Master has higher priority and coordinates co-simulation steps. The slave cannot communicate with the master unless required. This method is simpler than the other two methods. One of the challenges of the Master/Slave approach is to adequately adapt the time resolution of both simulators to avoid inaccuracies when exchanging messages [59]. One of the simplest ways, by implementation, to integrate OpenDSS and OMNeT++ is to adopt the Master/Slave approach, where OMNeT++ is the Master and OpenDSS is the Slave [66].

- Global Event Driven Synchronization

In a global event driven system, the co-simulator keeps a global list of events from a combination of power system iteration events and communication network events. These events are recorded by timestamp and executed accordingly. Power system dynamic simulation is time-driven but is solved in a discrete manner, then, each of the iteration rounds can be treated as a special discrete event [40]. Only one event runs at a time, other events are paused until its completion. This approach eliminates the time accumulation error.

The downside of this approach is a slower simulation speed as each simulator is paused for the resolution of every event [38].

Another bottleneck in a co-simulation framework is the mechanism for exchanging data. Communication overhead in a co-simulation environment can significantly increase simulation time for large network sizes [43]. An efficient co-simulation framework should implement a protocol that minimizes this overhead.

2.3 Big Data in Smart Grid

Big Data is the information asset characterized by such a high volume, velocity and variety to require specific technology and analytical methods for its transformation into value [19]. This implies that services that effectively manage data transmission, integration, aggregation, analysis, storage and security aspects of the smart grid require certain considerations; some of these aspects may overlap others. Further, the services to manage these areas of the smart grid differ based on the structure of the grid and type of information obtained from the grid. Figure 2.3 by J. Rodriguez-Molina and D. M. Kammen represents a high-level architecture of the information and communication structure in smart grid systems. Each of the services to manage data and communication is classified into one or more layers.

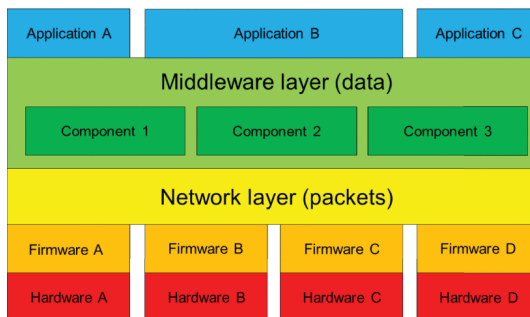


Figure 2.3: High-level layered architecture of ICT services in smart grid [58].

2.3.1 Data Transmission

Data transmission involves the mechanism and services responsible for moving data from one point to another across the grid. Many devices do not have directly compatible interfaces; network protocols allow these devices to communicate. Network protocols provide an established set of rules for devices with different interfaces or designs to communicate over the internet. Data transmission falls into the network and middleware layers of the architecture in Figure 2.3. There are many protocols used in data transfer over networks; Hypertext Transfer Protocol (HTTP) used by web browsers is the most popular. However, lighter weight protocols like AMQP and MQTT are commonly used in smart grid and constrained environments.

2.3.1.1 AMQP

Advanced Message Queuing Protocol is a programmable messaging protocol that enables conforming clients to communicate with conforming middleware brokers. The clients or applications define the routing schemes, messaging properties, queue properties and how messages are routed between applications and clients, the broker only ensures that messages are routed to the proper destinations [3]. Clients can subscribe to and/or produce messages to a channel. Messages are routed through flexible exchanges that decide in which queues to deliver a message. The protocol specifies different types of exchanges, and the type determines how the messages are routed. AMQP queues store and relay messages to clients subscribed to the queue [3]. They can be defined as persistent to allow multiple consumers to read the same message. In addition, the protocol provides an acknowledgement mechanism to ensure reliable message delivery. The broker can acknowledge messages sent from an application and a consumer can also acknowledge read messages.

2.3.1.2 MQTT

Messaging Queue Telemetry Transport (MQTT) is lightweight publish-subscribe protocol designed for power constrained and low bandwidth devices used for machine-to-machine connectivity over TCP/IP [44]. Similar to AMQP, MQTT specifies clients and brokers. Clients can publish messages to, and subscribe to an exact or multiple topics on a broker using wildcards such as # and +; brokers or servers are responsible for routing the messages. The publish-subscribe architecture makes the protocol lightweight with a small footprint on the network and hosting device [17]. MQTT ensures reliable message delivery by three Quality of Service (QoS) specifications. QoS 0, termed “at-most-once delivery,” delivers a message without an acknowledgment of receipt. QoS 1, termed “at-least-once delivery,” sends an acknowledgment to the sender for every message; unacknowledged messages are redelivered with a duplicate tag. Lastly, QoS 2, termed “exactly-once delivery,” ensures that duplicate messages are not delivered to the receiving client [17]. For security, MQTT supports TLS/SSL and authentication mechanisms. Since developed by Andy Stanford-Clark and Arlen Nipper at IBM in 1999, it has been made free and standardized with many versions and variations [44].

2.3.2 Data Communication Middleware

Data communication middleware are the services responsible for the effective exchange of data and control between devices, sections or applications in the smart grid. The performance and reliability of the grid is based on how effectively this is achieved, in a suitable format and within an accurate timeframe. According to Rodríguez et al., the middleware is classified based on services availability, message coupling, computational capabilities and middleware distribution [58]. Hence, communication middleware can be divided into Remote Procedure Call Oriented Middleware, Transaction-Oriented Middleware, Object-Oriented Middleware and Message-

Oriented Middleware [3]. Message-Oriented Middleware (MOM) is the best fit for the smart grid because data collected and exchanged are event based.

MOM provides a mechanism for sending messages across applications on a distributed system. The characteristics of MOM include support for asynchronous and synchronous communication, data format transformation, loose coupling, parallel message processing and multiple message priority levels [3]. M. Albano et al. broadly divide MOMs into three paradigms:

- Message Passing provides a direct communication between applications over a channel. The middleware eased the creation of the channel.
- Message Queuing allows data communication via First-In-First-Out queues. The middleware provides the infrastructure for creating and accessing the queues, routing messages to appropriate queues, and broadcasting messages to consumers of a queue.
- Publish Subscribe MOM is a type of Message queuing. The publishers send messages via middleware to one or many subscribers. The subscription can be topic based, content based or data oriented. The publishers and subscribers are fully decoupled. First, the publisher does not require information about the subscriber. Secondly, the publisher and subscriber are not required to be online at the same time, messages can be published in the queue and consumed at different times. Lastly, it allows asynchronous notification of subscribers by events via callbacks [3].

Message-Oriented Middleware systems offer three levels of delivery schematics, at-least-once, exactly-once and at-most-once delivery [46].

2.3.2.1 RabbitMQ

RabbitMQ is an open-source message broker with support for multiple messaging

protocols including AMQP and MQTT. It enables clustering, federation and distributed configurations for high-availability. It provides a publish and subscribe messaging mechanism. Producers write into queues through an exchange while consumers subscribe and read messages from the queues via Push or Pull methods. RabbitMQ brokers host message queues and routing exchanges. Further, it supports two delivery schematics, at-most-once and at-least-once delivery [46]. To achieve this, RabbitMQ brokers have four exchange types for routing messages from queues to subscribers. First, Fanout exchange sends messages to all queues bound to the exchange. Second, Direct exchange uses a routing key; a routing key binds one or more queues to an exchange. A message sent to a routing key is sent to all queues bound to the key. Third, Topic exchanges route messages to one or many queues based on matching between a message routing key and the pattern that was used to bind a queue to an exchange [54]. Lastly, the Header exchanges route messages based on header attributes. Messages are primarily kept in the main memory but are persisted to disk when flagged as durable.

2.3.2.2 Data Format

The middleware only creates an abstraction over communication and does not specify the message structure or content. This means parties in a data exchange must agree on a data format and each party must support serialization and deserialization over the format. The choice of data format impacts message size and the speed of serialization and deserialization. Hence, the performance of the middleware depends on data format and efficient data conversion. To meet the constrained, dynamic, high speed needs of the smart grid environment, a data format must be flexible, lightweight, cross platform and efficient. JSON and Protocol Buffers are two common data formats.

- JavaScript Object Notation (JSON) is a language independent, human read-

able and relatively lightweight data-interchange format [31]. JSON is built on two structures, a collection of key-value pairs and an ordered list of values. It is flexible and does not enforce a schema. Therefore, parties in an exchange are not required to agree on a schema before exchange, serialization and deserialization. This makes it easier to use, however, the drawback is that some clients may send unreliable or incomplete data. While JSON is native to JavaScript, it is supported by multiple languages such as Java, C++ and Python.

- Protocol Buffers are binary-based language-independent, lightweight and efficient mechanisms for serializing structured data [27]. They support different primitive data types encoded into binary. Encoding is done by a method called Varints algorithm [27]. Varints are used to serialize integers using one or more bytes. Protocol Buffers are schema based; the message structure is defined in a proto file. A message structure consists of multiple fields. A field definition is a data type, field rule and a number used to identify the field in binary format. The proto file is used to generate optimized platform-specific source code for data serialization and deserialization. The use of schema, proper encoding of specific data types and platform specific code makes Protocol buffers efficient, reliable and lightweight. Relative to JSON, Protocol buffers are faster and have smaller message sizes. Some languages supported include Python, C++, Go and Java.

2.3.2.3 Schema Evolution

The type and content of data changes over time as a system evolves. When data changes, their representation or schema in a particular data format also change. Schema evolution involves managing forward and backward compatibility of data as it changes in a system. It is essential to manage schema evolution to ensure clients can understand previous and current data. One of the ways to handle schema

evolution is versioning. Each message is tagged with a schema version used by the client to serialize and deserialize the data in the correct format. A schema registry stores, tracks and maintains different versions of data schema.

2.3.3 Data Aggregation and Analysis

To derive value from the large stream of data available in the smart grid environment, it must be efficiently and securely collected, analyzed and visualized. The application layer in Fig 2.3 includes services that interface with data and services on the middleware layer. To meet the needs of the smart grid environment the services must be fast, efficient, secure and scalable. Before Apache Spark was widely adopted, the combination of Apache Hadoop Map Reduce, Hadoop Distributed File System and YARN was de-facto for analysis on large datasets. Hadoop works by dividing the datasets into smaller sets and performs computation in parallel. However, MapReduce is suitable for batch processing but performs poorly with streaming and real-time data [45]. Added to the difficulty of programming Map Reduce, the algorithm provided by the platform is limited. Recently, Apache Spark, which supports stream processing, is commonly adopted as a replacement in big data analytics. Stream processing involves calling dependent logic on each new data instance, rather than waiting for the next batch of data and then reprocessing everything. Stream processing provides timely and more accurate results when compared to batch processing [45]. This method of processing is well suited for the smart grid environment. Other technologies like Kafka, Flume or Storm are also utilized.

2.3.3.1 Apache Kafka

Apache Kafka is an open-source distributed event streaming platform. It functions primarily as a distributed commit log, a publish-subscribe messaging system or as a distributed streaming platform for high-performance data pipelines, streaming

analytics, and high-speed data ingestion [45]. Kafka supports clustering for high-availability, fault tolerance and scalability. A unit of data in Kafka is a message. A message is constructed of a key and a value. These messages are classified into topics and partitions. A topic can have multiple partitions replicated across different Kafka nodes in a cluster. Kafka messages, topics and partitions are managed by a broker. One or more Kafka brokers form a cluster for high availability and scalability. The broker is responsible for receiving messages from the producers assigning offsets to them and committing the messages to storage on disk. The primary clients in Kafka are the Producer and the Consumer. Producers write commit messages to a topic and optionally specify a partition. If a partition is not specified, messages are distributed across topic partitions without a guarantee of order. Kafka consumers read messages from topics or partitions and keep track of the offset of read messages. They form consumer groups where one or more consumers work together to consume a topic. The group assures that each partition is consumed by only one member in the group. Messages in Kafka can be persisted for a configurable period. This enables new consumers to read messages from the earliest offset. Kafka provides other client APIs to interact with Kafka topics and brokers. It is commonly paired with other stream processing systems for big data processing. Kafka supports all three levels of delivery schematics of an MOM [61].

2.3.3.2 Apache Spark

Apache Spark is a unified computing engine and a set of libraries for managing and coordinating the execution of parallel data processing on computer clusters [14]. A cluster is managed by a cluster manager like Apache Hadoop YARN¹. Spark Applications are made up of packaged code describing operations on data and are submitted to the cluster manager. While Spark is written in Scala and runs on the Java Virtual

¹YARN (Yet Another Resource Negotiator) is an open-source resource manager and job scheduler utilized in clustered environments

Machine, Spark Applications can be written in Python, Scala, R or SQL. A Spark Application consists of a driver process and a set of executor processes [14]. The driver process on a cluster node maintains information about the Spark Application, responds to user programs and input, and analyzes and schedules work across executors. Executor processes execute workloads assigned by the driver process and returns the results. Apache Spark loads data from a storage system and distributes multiple nodes in the cluster. It supports a variety of storage systems including Azure Block Storage, Amazon S3, distributed file systems like Apache Hadoop, key-value stores like Apache Cassandra and streaming platforms like Apache Kafka [14]. The data is loaded into Data Frames, Resilient Distributed Datasets (RDD) or Datasets data structures that expose APIs for easy transformation and manipulation. Unlike traditional Hadoop Map Reduce, which processes data by writing and reading from disk, Apache Spark can process plain or graph data types in memory, in real time or in batches. Additionally, the available engine libraries include a Machine Learning library, an SQL interface to query data and APIs to manipulate structured streaming data. Support is also provided for third-party community libraries. Operations performed over loaded data range from simple aggregations, grouping, joins and sorting to complex window querying, transformations, regression, and clustering.

2.3.3.3 Logistic Regression

Logistic regression (LR) is one of the statistical or machine learning algorithms supported by Apache Spark. It models the probability of a discrete outcome given one or more input/independent variables. LR is a logit transformation of linear regression using the sigmoid function to restrict the value from a large scale to a range within 0 and 1 [63]. Linear and logistic regression are based on the same assumptions, however, linear regression models continuous dependent outcomes.

Given β as the coefficient of regression and X or X_i as independent/input variables.

(Linear Regression)

$$Y \approx \beta_0 + \beta_1 X$$

(Simple Logistic Regression)

$$p(X) = \beta_0 + \beta_1 X$$

using the sigmoid function

$$p(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}$$
$$\frac{p(X)}{1 - p(X)} = e^{\beta_0 + \beta_1 X}$$

Where $\frac{p(X)}{1 - p(X)}$ is called the odds. Finally,

$$\log\left(\frac{p(X)}{1 - p(X)}\right) = \beta_0 + \beta_1 X$$

(Simple Logistic Regression with Multiple Predictors)

$$p(X) = \frac{e^{\beta_0 + \beta_1 X_1 + \dots + \beta_k X_k}}{1 + e^{\beta_0 + \beta_1 X_1 + \dots + \beta_k X_k}}$$

$$\log\left(\frac{p(X)}{1 - p(X)}\right) = \beta_0 + \beta_1 X_1 + \dots + \beta_k X_k$$

The application of logistic regression to the smart grid and smart grid data include demand response systems [51], consumption pattern analysis [69, 13] and predictive control [9].

2.3.4 Data Storage

Storage of data for future analysis, auditing, error tracing and historical reference may be crucial in some smart grid systems. Data is written and read at high speed, fast, efficient and scalable data storage systems are required to match these needs.

NoSQL databases are commonly employed for data storage in a smart grid because they are more scalable and provide superior performance dealing with big data [18, 5, 42]. A NoSQL database stores data with flexible schema, in contrast to fixed columns and rows in relational databases. The types of NoSQL databases are document based, key-value stores, wide-column database or graph databases. The system and types of data dictate which type to use.

2.3.4.1 MongoDB

Document databases store data in documents similar to JSON/BSON objects. Each document consists of pairs of fields and values. MongoDB is a distributed document-oriented NoSQL database. It is made up of documents and collections. Documents are the basic unit of data and are tantamount to rows in a relational database; a collection is a set of documents, equivalent to tables in a relational database. Documents within a collection are schema-less and have a flexible structure. MongoDB provides a form of data consistency like relational databases. However, it implements BASE properties in place of ACID (Atomicity, Consistency, Isolation, Durability). BASE is made up three attributes: Basically Available, Soft-State and Eventual Consistency. Compared to ACID, this model is less strict and does not offer guaranteed consistency at write time. That trade-off gives MongoDB some advantage; it is fault tolerant and easily scalable, it supports master slave replication and horizontal scaling through sharding. The flexible, fast, efficient and scalable nature of this database makes it suitable for managing large datasets in the smart grid [18].

2.3.5 Security

Data in the smart grid environment is not limited to electrical parameters and values, it could include sensitive user consumption or company proprietary data. Therefore, it is essential to secure the smart grid to ensure integrity, privacy and confidentiality.

We consider two ways to secure data, information and communication in the smart grid, in motion and at rest. This section describes some smart grid data security concepts.

2.3.5.1 Point-to-point and End-to-end Security

Point-to-point encryption (P2PE) is a technology standard created to secure electronic financial transactions between point of interaction and processing. It enforces data security in transport between two directly connected systems. End-to-end encryption is a system of communication that ensures a message is readable to only the sender and the receiver. Additionally, the message is not readable by the medium of communication, a third-party storage service or an attacker secretly eavesdropping the traffic over the network. Only the intended recipient is able to read a message sent over a connection or series of communications. End-to-end encryption supports data privacy, confidentiality and integrity between directly or indirectly connected systems.

2.3.5.2 Transport Layer Security and Secure Sockets Layer

Transport Layer Security is a network-based security protocol designed to provide peer authentication and secure traffic between devices connected over a network. This protocol provides privacy, authentication, and data integrity. The protocol is at the transport layer of the TCP/IP model. It prevents data sent over the internet from eavesdrop, replay and man-in-the middle attacks. TLS is a standardized version and successor to the Secure Sockets Layer (SSL) protocol. SSL/TLS are used to achieve point-to-point encryption. A secure data channel is established through a handshake. In the handshake, the client and the server negotiate on a version of TLS protocol, decide on a cipher suite, exchange and verify SSL certificates and finally exchange a shared key. The SSL certificates are self-signed or signed by a Trusted

Certificate Authority for identity verification of a peer in the connection. SSL/TLS employs both asymmetric and symmetric encryption. Asymmetric encryption is used during the handshake for key exchange and digital signature verification; AES-128 symmetric algorithm and the derived key from the handshake are used to encrypt subsequent traffic.

2.3.5.3 Asymmetric Encryption

Asymmetric encryption is a form of data encryption with two keys, the public or encryption key and the private or decryption key. The public key is made available to the public, and used for encrypting and verifying signed messages. The private key is known only to the recipient of an encrypted message. It can decrypt a message generated by a corresponding public key. The security of asymmetric encryption is computational infeasibility of calculating the private key from the public key. A public-key encryption algorithm requires a trap-door one-way function, a function easy to compute but hard to invert without some secret trap-door [37].

2.3.5.4 RSA Encryption

RSA (Rivest, Shamir, Adleman) is a public-key encryption developed in 1977 based on the large integer factoring problem. Given two large prime integers p, q , the private and public keys are obtained by:

$$n = p * q$$

$$\Phi(n) = (p - 1) * (q - 1)$$

Given:

$$e \in \mathbb{Z} \quad \text{and} \quad 1 < e < \Phi(n)$$

$$\gcd(e, \Phi(n)) = 1$$

And d is the inverse of e calculated by the extended Euclidean algorithm:

$$1 < d < \Phi(n) \quad \text{and} \quad e.d \equiv 1 \pmod{\Phi(n)}$$

Where (n, e) is the public key and (n, d) is the private key, the encryption operation to derive a ciphertext C from a message or plaintext M is:

$$C = M^e \pmod{n}$$

The plaintext can be retrieved by decryption with the private key (n, d) :

$$M = C^d \pmod{n}$$

For the security of RSA, 1024 or 4096 bit keys are used in practical applications.

2.3.5.5 Key Management

Key management is a major problem in cryptography. Encryption and decryption algorithms are public and secured systems depend on the secrecy of the encryption key. Keys should be securely managed with efficient and easy access by authorized users. In public key cryptography, it is essential to make the public key readily available. One way to do this is the use of a Public Key Registry. A Public key registry is a key-value store or database for public keys. Redis data store is a popular example of a key-value store for fast read, write and indexing of data [56].

2.4 Cloud-based Smart Grid Management

Cloud computing is a distributed model of computing to provide on-demand network access to a shared pool of resources that can be rapidly scaled with minimal effort [42]. These resources include hardware and services such as network, servers, storage

and applications. Leveraging on cloud-based infrastructure can provide scalable and flexible computing, storage, and processing resources suited for smart grid big data. There are potential applications of cloud-based technology in many areas of a smart grid including system monitoring and control, information security and storage, power system simulation, and dispatching and planning [23]. There are a great number of cloud solutions that support big data such as Amazon Elastic Compute Cloud (Amazon EC2), Google Compute engine, Microsoft Azure Cloud and IBM Docker Cloud.

2.4.1 Containerization and Docker

Containerization is a form of operating system virtualization that enables running multiple services and applications in isolated environments, using shared resources on the same machine. A container is an isolated environment containing files, libraries and dependencies required for a service to run. In contrast to a virtual machine, a container shares the host operating system kernel. Docker is an open container runtime to build, deploy and run distributed applications in different environments from laptops and virtual machines to cloud environments [21]. Docker makes it easy to ship and scale multiple services in the cloud.

2.4.2 Kubernetes

Kubernetes is an open-source platform for managing containerized workloads, services and for automating deployment, scaling and orchestration of containerized applications [7]. It groups containers that make up an application into logical units for easy management and discovery. Google founded Kubernetes in 2014 to produce an open-source container orchestration tool like Borg, used internally by Google for many years. The rapid development and features of Kubernetes have made it the de-facto tool for container management. A Kubernetes cluster is made up of the

master node (control plane) and a set of worker nodes. The nodes in a cluster host Pods; Pods contain running containers and are the basic unit of deployment in Kubernetes. The control plane manages the nodes and the pods in a cluster. Each node consists of an agent called a kubelet which ensures containers run in pods according to the declared specification. A node also contains a container runtime like Docker responsible for running containers. Kubernetes also has components called Services for managing network communication between Pods and external applications. The features and architecture of Kubernetes is described in detail by the Kubernetes authors in the documentation [7].

2.5 Related Work

In the literature, several works have explored the domain-specific tools and co-simulation frameworks for smart grid systems, communication middleware and cloud-based application and data management for the smart grid. M. Vogt et al. [68] surveyed smart grid simulators and classified them based on the synchronization techniques, chosen power simulation tool, the hardware-in-loop and source availability of the co-simulator. The authors identified the three synchronization techniques explained in Section 2.2.5 above, common co-simulation scenarios and the common tools. Table 2.2 shows some co-simulation from the literature, the application area, the synchronization method of co-simulation and middleware employed.

P. Lipčák et al. [41] discuss the use of big data tools in analyzing and visualizing anomalies in smart grid environments. The application described in this paper maps the reference architecture in P. Pääkkönen and D. Pakkala [49]. In the study, Smart meter data is written into Apache Kafka, which streams and provides the data for storage in a Hadoop Distributed File System. Computation jobs to be performed on this data in Apache Flink² are submitted through a spring boot application.

²Apache Flink is a tool used for stateful computations over data streams.

The results of the computations are visualized in a React Frontend³. Grafana⁴ also handled some of the visualization tasks. A. Aydin et al. [8] described a different architecture for real-time and batch incremental data collection and analytics platform. The software architecture used consists of four layers: application, service, index and storage. The storage layer is based on SQLite for storing application layer data and Cassandra NoSQL database for automatic partitioning of data across a cluster. The index layer incrementally indexes large data sets while collecting data in real time. The service layer is based on RabbitMQ as a message bus, Redis and Apache Spark. The architecture was built for fault tolerance and resilience. R. Shyam et al. [61] explore applicable areas of a big data analytics platform for smart grid data. It compares the differences and advantages of Apache Spark over Map Reduce and Hadoop. The authors classified smart grid data processing into batch processing, real-time processing and iterative processing. In the study, Phasor Data Units (PMU) time series data was analyzed using a 3-node Apache Spark cluster. In the setup, Apache Kafka ingests PMU streaming data into the Apache Spark engine for windowed processing. The results demonstrate the versatility of Apache Spark in different areas of smart grid data analytics. To discover the best technology for data storage, M. H. Ansari et al. [5] explores the performance evaluation of big data frameworks for analysis of smart grid data. It compared the performance of NoSQL HBase, Elasticsearch, MongoDB and Cassandra for data storage. The results favor Cassandra and MongoDB as top choices for data storage.

Some works also research the middleware employed in smart grid systems. J. Rodríguez-Molina and D. Kammen explore the protocols, use cases and evaluations of Message-Oriented Middleware in a smart grid system [58]. The paper highlights the ENCOURAGE platform, a group of user-focused smart grid technologies, for the seamless interaction of existing devices by different vendors, fine-grained remote control

³React is a Javascript framework for building web interfaces and applications.

⁴Grafana is an open-source visualization tool with support for a variety of data sources.

of user's appliances, and energy brokerage. The middleware of the platform delivers data encoded in XML format through an AMQP-based messaging bus. B. Petersen et al. [50] seek to find more optimal middleware than XML. The authors compared multiple serialization formats including MsgPck, JSON, ProtoBuff based on message size, serialization and deserialization time. ProtoBuf performed relatively better. The research also evaluated some message-oriented middleware solutions based on latency and throughput. Finally, B. Fang et al. [23] and H. Farhangi [10] describe how cloud technology can provide scalable infrastructure for smart grid systems and applications. Some cloud supported applications highlighted include Demand Response systems, Peak and dynamic pricing management systems and real time monitoring systems.

Table 2.2: Related Work on Co-simulation

Ref.	Name	Power tool	Network tool	Sync. Method	Application	Middleware
[48]	VPNET	VTB	OPNET	Time stepped	Influence of communication delays in the performance of control and protection functions in a medium-voltage dc (MVDC) shipboard smart grid.	HTTP Request with JSON encoding
[39]	PowerNet	Modelica	NS2	Time Stepped	Influence of communication delays wind turbine voltage controller	
[36]		OpenDSS	OMNeT++	Master-slave	Impact of data rate-based and event-based sensors on reactive control algorithms of plug-in electric vehicles to reduce critical voltage durations.	HTTP Request with JSON encoding
[40]	GECCO	PSLF	NS2	Global event driven	Agent-based remote backup relay protection scheme.	
[11]	EPOCHS	OpenDSS	OMNeT++	Time stepped	Protection scheme for transient power system.	
[65]		OpenDSS	OPNET	Global event driven	Evaluate the reliability of the control strategies in smart grid with DERs (Distributed Energy Resources)	
[20]	Mosaik		OMNeT++	Time stepped		TCP Socket Connections

Chapter 3

Motivation

Many published papers have presented several approaches to solve the three major highlighted issues of smart grid co-simulation, communication and big data management. However, communication overhead in a smart grid system can affect performance and can significantly increase the simulation time in a co-simulation environment for large network sizes. An efficient framework should utilize a protocol and synchronization that minimizes overhead. Some current co-simulation frameworks employ request-response or REST APIs and JSON data formats, others use database polling as a mechanism for data exchange. These methods have strengths and drawbacks. For example, Figures 3.1 and 3.2 show a comparison between the message latency of a simple database polling data exchange and a publish subscribe message-oriented exchange in JSON format. This experiment was conducted on a machine with 32GB of RAM, 2.6GHz 6-Core Intel Core i7 running MacOS 11.01. A MySQL database was used for polling and RabbitMQ version 3.8.19 was the publish-subscribe broker. The publish-subscribe and message polling applications were written in Python. The results reveal that message latency is higher and more inconsistent with database polling. It also shows writing/sending messages are faster with the publish-subscribe system. Additionally, database polling does not

support complex routing of messages between applications, and latency degrades with more records. For this reason, one of the goals of this research is to review existing solutions and present a framework to meet optimal requirements applicable in production systems with minimal trade-offs.

Further, there are many tools and technologies available to manage different aspects of big data in a smart grid. Data collection and aggregation alternatives include Apache Storm, Flume and Kafka. Data storage options include Apache Cassandra and MongoDB. Data analytics options include Apache Hadoop and Apache Spark. However, several factors also influence the choice of the tools utilized. The following are essential:

- Flexibility and scalability for large workloads
- Support for high performance, availability and fault tolerance
- Libraries and integrations should support multiple platforms
- Security
- High throughput and low latency
- Open source or suitable commercial license
- Support for real-time analytics of streaming data

The other contribution of this research is to create a big data technology framework for the smart grid to meet the above requirements.

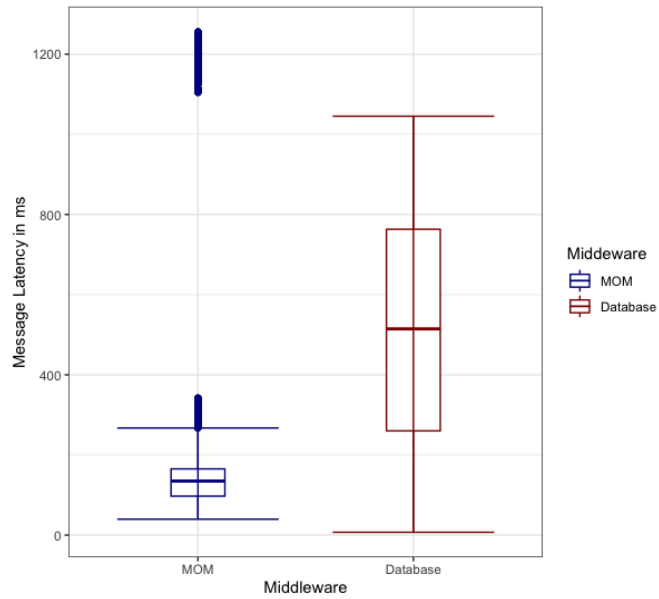


Figure 3.1: Message Latency of 10,000 JSON messages using database polling and publish-subscribe middleware.

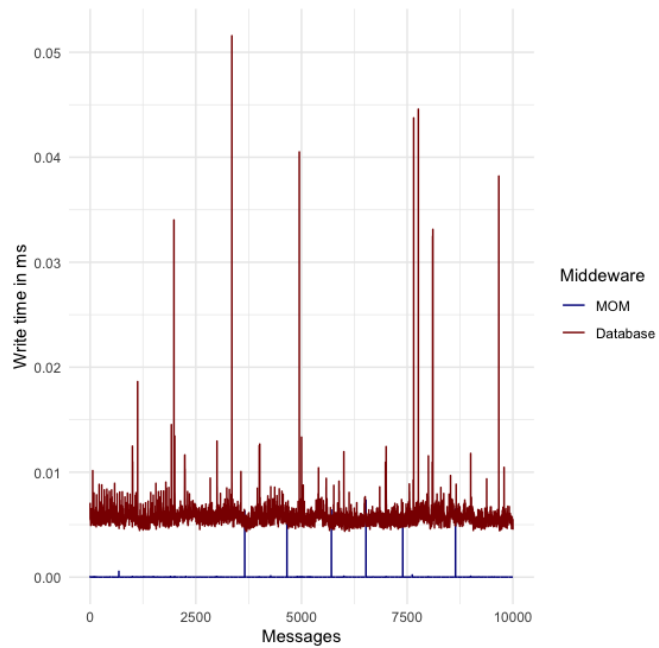


Figure 3.2: Write time of 10,000 JSON messages using database polling and publish subscribe middleware

Chapter 4

Design and Implementation

This chapter describes the design of our smart grid framework and the motivation behind the technologies chosen for the framework. The framework handles two broad functions: efficient data and communication middleware (co-simulation and real systems) and big data management.

4.1 Co-simulation

One of the goals of the co-simulation section of the framework is to provide a time synchronization technique with minimal error and time ratio. The other is to provide a common and efficient means of data exchange with minimized communication overhead. In a co-simulation, the OpenDSS power simulator models the electrical circuits, power devices, utility generation and distribution systems. It supports efficient power analysis of power delivery and smart grid applications. The role of the simulator in the framework is to support various types of power analysis and simulation scenarios including Distribution Planning and renewable energy integration simulations. It solves a given power model and advances a simulation in different simulation modes at various time steps based on input parameters. These modes include daily or yearly power flow, duty-cycle power flow, harmonics and fault study

analysis. The power simulation instance is interfaced and controlled through its COM interface. Supplying input parameters to the simulation and fetching results and data from meters, monitors and other elements is done via the COM interface. The communication section of a model in co-simulation is implemented and run in the OMNeT++ simulator. OMNeT++ models various communication protocols and capabilities of the elements in the power model with a good representation of real anomalies and scenarios such as delays, packet loss and interference.

4.1.1 Time Synchronization

To ensure a seamless co-simulation with minimal errors, the synchronization technique connecting the power and communication models combines the master-slave or primary-secondary synchronization and the time-stepped synchronization technique. In a master-slave time-stepped synchronization technique, a time step and a time interval are defined. The simulation is advanced in units of the time step to push the simulation time and simulation further. The configurable time step determines how fast the simulation progresses in the absence of other events and in effect determines the time ratio. The time interval is a set time frame after which both simulators are synchronized in the absence of a critical event or an error. The primary and secondary simulator(s) are designated, in this thesis, OMNeT++ and OpenDSS respectively. Both simulators are scaled to the same time unit. The primary simulator leads the simulation, it makes requests to the secondary simulator(s) for data, control, synchronization or fault resolution. The primary simulator events are divided into critical and non-critical events. Critical events are time sensitive and are processed immediately. This includes control request events, faults and errors. Critical events mostly require a synchronization before they are processed. Non-critical events are processed with or without synchronization, events like a device or node information request. The classification of events into critical or non-critical are based on the sim-

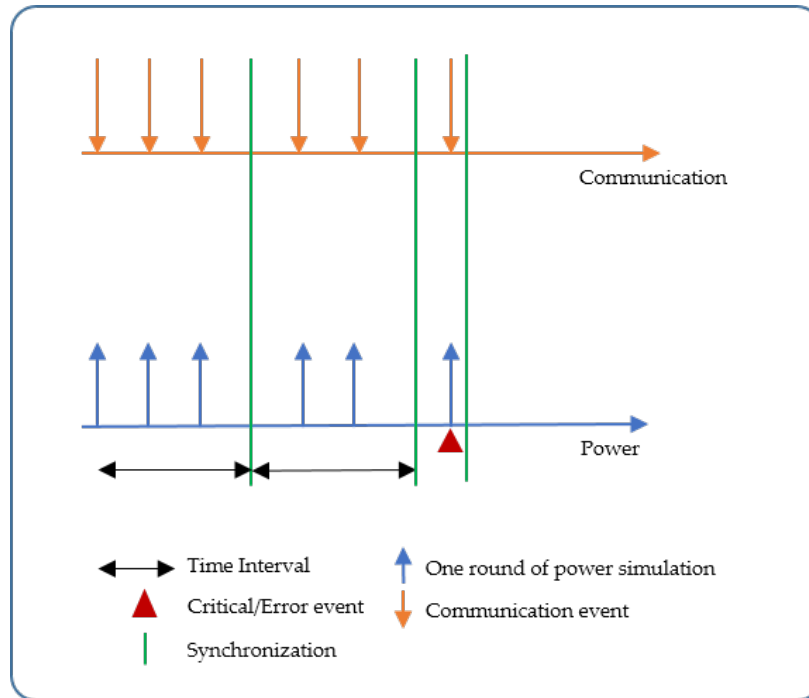


Figure 4.1: Combination of Time-stepped and Primary-secondary Synchronization

ulation scenario and are defined by the designer of the co-simulation. Therefore, a synchronization point in the co-simulation is defined by time interval in the absence of critical events or by the occurrence of a critical event. At synchronization the communication simulator sends a message event to the power simulator to solve for the circuit for x units of time. Figure 4.1 illustrates the technique.

This technique avoids the error accumulation problem in the pure time-stepped technique by handling errors immediately. It also resolves the issue of time resolution in a pure master-slave synchronization technique by scaling all the simulators in the co-simulation to a common time unit. To allow plugging and running a co-simulation with real life systems, the time scale is adjusted and a real-time scheduler that synchronizes the simulation with the real world clock is used in place of a sequential scheduler.

4.1.2 Data Exchange Middleware

While the time synchronization technique defines when data is exchanged, it is important to select an optimal method to exchange the data in co-simulation to reduce communication overhead. The effect of a poor communication medium may be negligible for simple co-simulation scenarios involving a model with a few nodes. However, as the model grows larger the communication overhead becomes significant, increasing time ratio, and affecting accuracy. In addition to communication within the co-simulation, middleware should support data and control interactions with external systems and applications including a real smart grid system. In this thesis, three main data middleware solutions are considered:

- Database Polling
- Request Response
- Message Oriented Middleware: the background section of this thesis discusses this in more detail.

In a smart grid co-simulation, the simulation requirements range from simple to complex analysis, hence multiple factors influence the choice of data middleware. Table 4.1 shows these factors considered and how effectively each middleware solution supports the factor.

Based on its suitability features, the framework in this research employs RabbitMQ as a message-oriented middleware for co-simulation. Figure 4.2 shows the design of a simple co-simulation with RabbitMQ middleware, and Figure 4.3 shows the co-simulation middleware support for data interaction with external systems.

4.2 Data management

The following subsections describe how the framework presented in this research

Table 4.1: Middleware options and support for co-simulation framework requirements.

Feature/ Method	Database Polling	Request Re-	MOM
Complex Routing	Hard	Hard	Easy
Ease of use	Easy	Medium	Easy
Latency	Degrades with more records	Medium/Low	Low
Throughput	Degrades with more records	Medium	High
Scaling	Fair Support	Fair Support	Mostly Out of the box
Message delivery	Manually managed	Manually managed	Mostly Out of the box
Bi-directional Exchange	Possible but complex	Hard	Easy
Data Locality	Easy but resource intensive	Easy	Relatively easier
Supporting for streaming	Fair support. (Cassandra)	Hard	Large integration support
Support for Co-Sync simulation methods	Master-slave	Master-slave, time-stepped	Supports all methods.



Figure 4.2: A simple co-simulation with OpenDSS and OMNeT++

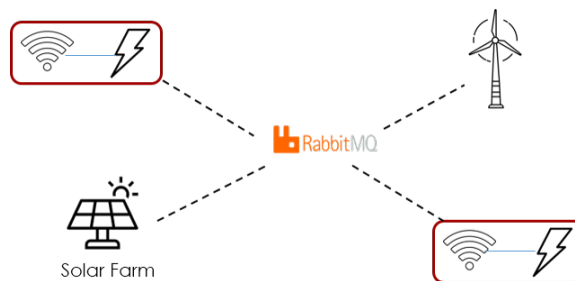


Figure 4.3: A simple co-simulation with OpenDSS and OMNeT++, RabbitMQ as middleware for intra and external data exchange.

handles the different parts of the big data management in a smart grid.

4.2.1 Data Transmission

Due to the constrained computing nature of devices operating in the smart grid, the framework employs the Advanced Messaging Queue Protocol (AMQP) and the Message Queue Telemetry Protocol (MQTT) for uniformly exchanging data between different nodes in the smart grid. Data transmission includes data exchange between real devices and [co-]simulation. These protocols were chosen for their lightweight and low overhead. Other common existing data protocols in IoT (Internet of Things) and smart grid environments include Hypertext Transfer Protocol (HTTP), Constrained Application Protocol (CoAP, a modification over HTTP), Data Distribution Service (DDS), and Extensible Messaging and Presence Protocol (XMPP) [1]. In choosing a protocol, we considered three things: the performance of the protocol; the support for publish/subscribe; and the reliability and message guarantee (Transmission Control Protocol, TCP vs. User Datagram Protocol, UDP). Based on performance, we eliminated XMPP because it uses XML, which is comparatively heavier and has more overhead than JSON-based Representational State Transfer (REST) systems [1]. HTTP also has a considerably higher overhead compared to AMQP and MQTT [62]. In environments or applications with frequent data updates and big data requirements, a publish/subscribe protocol is considered a viable choice [70]. CoAP is RESTful, supports request/response and publish/subscribe, but uses UDP, which is less reliable than protocols that run on TCP. AMQP, MQTT and DDS protocols support publish/subscribe and work over TCP. However, we choose AMQP and MQTT and not DDS based on support and compatibility with other parts of the framework. RabbitMQ for example, has been proven to work very well with both protocols [32] [54].

4.2.2 Data Communication

Data communication covers the middleware for carrying messages and data across the smart grid, the data format and how data schema evolution is managed. In this framework, the data communication middleware in the smart grid co-simulation (discussed in Section 4.1.2) applies to the context of middleware in a big data smart grid. Furthermore, in simulation scenarios where complex routing is not required, Apache Kafka, which can serve as a publish-subscribe system, could replace RabbitMQ as message-oriented middleware. Apache Kafka provides higher throughputs and lower latencies with increased difficulty of handling complex routing as a tradeoff.

In choosing a suitable data format in the framework, the options are compared based on the serialization and deserialization time, compression and decompression time, memory use, and message size. Based on the results of the experiments performed by B. Petersen et al. [50], the options considered for the framework were narrowed to primarily Protocol Buffers and JSON. While JSON was not concluded to have performed best, it was chosen for consideration for its ease of use, current wide usage and compatibility with existing systems. In a single system, both the data formats could be mixed for different parts, however, it is recommended that only one data format is used at a time. The experiments in this thesis apply one at a time.

In the presented framework, there are two recommended ways to manage schema evolution. The first method is message and schema versioning. Each message is tagged with a schema version used by the client to fetch the schema, as well as serialize and deserialize the data in the appropriate format. A schema registry stores, tracks and maintains different versions of data schema. The other method is version control where a repository maintains serialization and deserialization functions. Each application, node or client in the system pulls a copy of the functions for use. When an update to the schema is required, the functions are updated, committed and pushed to the shared repository. All using applications are required to pull the

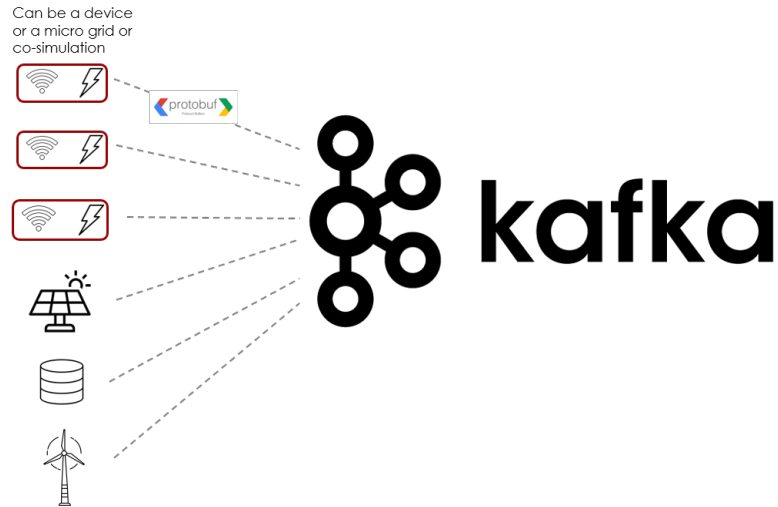


Figure 4.4: Kafka as a tool for aggregation

update. The downside to this method is that for each update, all nodes or applications must be updated. However, data schema is not changed too often in the smart grid environment for this to constitute a major bottleneck. Finally, it is noteworthy that in some cases schema evolution is easier to manage in schema-less data formats like JSON.

4.2.3 Data Aggregation

To support streaming data and multiple data sources, Apache Kafka is employed for aggregating data from nodes across the smart grid including plugged-in simulation instances as shown in Figure 4.4. Apache Kafka is suitable because of its wide range of support for different data sources, streaming and non-streaming, including databases, output from analytics, web, and IoT devices. It also integrates well with streaming data sinks like databases, analytics tools like Apache Spark and visualization tools. Aside from Kafka client libraries for Java, Python and C/C++ among others, Kafka provides a connector API for integrating with different data sinks and sources. Finally, Kafka easily scales horizontally for high availability, high throughput and low latencies.

Table 4.2: A comparison of Apache Spark and Map Reduce for big data analytics

Apache Spark	Hadoop Map Reduce
In memory processing	On disk processing
Process plain data and graphs	Process plain data
Easy to use	Relatively Hard
Large and easy support for Machine Learning	Relatively Less support
Suited for Batch Processing and Streaming/Real time	More suited for Batch processing Require additional tool for real time
Fault tolerant	Better fault tolerance
Provides Spark SQL for easier querying	
Support structured Streaming	

4.2.4 Data Analytics

To support real-time analytics, the presented framework uses Apache Spark. The motivation behind selecting Apache Spark over a close alternative, Hadoop Map Reduce, for analytics in smart grid is shown in Table 4.2. It integrates seamlessly with Kafka for processing streaming data and writing results back into Kafka or storage. Spark is also selected because of its support for multiple data sources including the Hadoop Distributed File System, databases like Cassandra, and others like Azure Block Storage and Amazon S3. This is useful in performing analytics that require historical data. Figure 4.5 shows how Spark fits into our smart grid framework.

4.2.5 Data Storage

Choosing the right database for storing smart grid data is critical. A less than optimal choice could affect the write and read speed by up to 4 times [71]. For this framework, only NoSQL databases are considered because they are more suitable for the smart grid environment due to their flexibility. Furthermore, literature reviews indicate that NoSQL databases are commonly used for smart grid storage. An

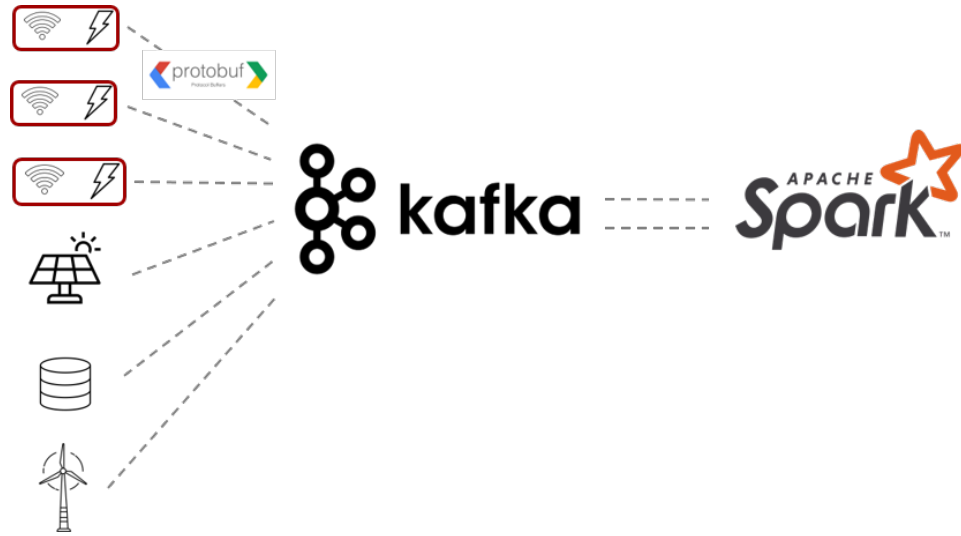


Figure 4.5: The role of Apache Spark for Analytics

experiment by E. Yilmaz et al. [71] comparing different storage options for the smart grid concludes that Apache Cassandra and MongoDB outperform HBase NoSQL database and SQL options like PostgreSQL. This framework has chosen MongoDB.

4.2.6 Data Security

The framework utilizes TLS/SSL to secure data in transmission between services. However, a potential issue is the management of TLS certificates for clients, servers, and brokers. TLS certificates are managed by Kubernetes Secrets. Kubernetes Secrets store and manage sensitive information, such as passwords, OAuth tokens, and TLS credentials. These secrets can be passed to the services hosted in the cluster as files or environment variables. To store data at rest and ensure end-to-end security, messages are encrypted using the RSA algorithm. The RSA asymmetric algorithm was chosen over symmetric alternatives like AES because of key management. The main issue is securely managing a single key between multiple services; the overhead in solving this problem outweighs the benefits of using symmetric algorithms. Another issue is that when one service is compromised, all the services sharing the same

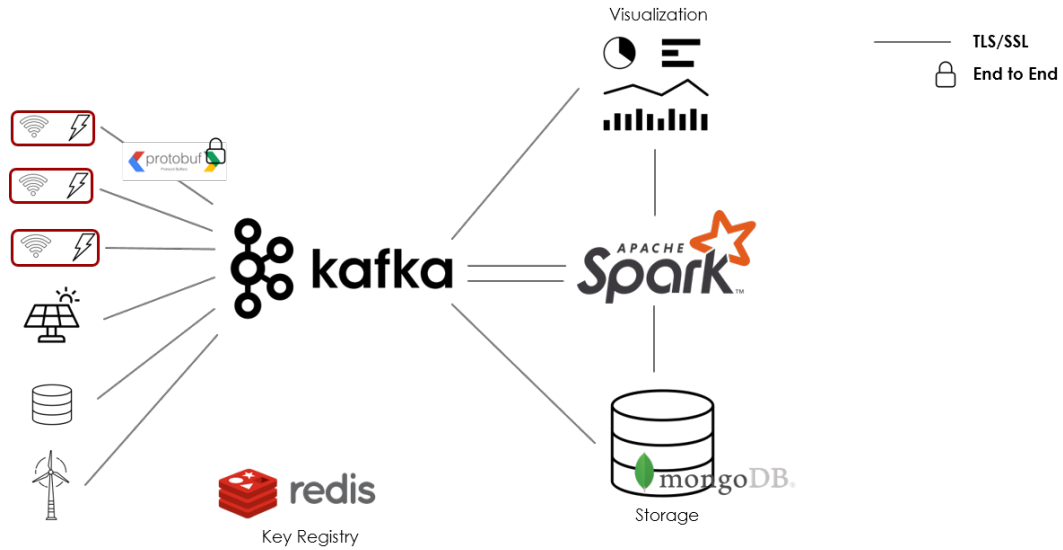


Figure 4.6: A secure and scalable framework for data, communication and co-simulation

key are equally compromised. However, the public and private keys in RSA must also be efficiently and securely managed. A Key Registry in the framework makes the public key of each service centrally accessible to all other services. The registry is a Redis Key value store. Redis was chosen for its fast write and lookup. The private key of each service is managed by Kubernetes Secrets. Figure 4.6 shows a complete design of the framework. An alternative is to combine symmetric and asymmetric encryption. With this method, a symmetric encryption algorithm like Advanced Encryption Standard (AES) is utilized for message encryption because of its speed and to mitigate the computation overhead of asymmetric encryption (RSA). To manage the key, RSA is used to encrypt the symmetric key and is passed with every message or exchanged in a handshake between the two communicating services. For simplicity, in our framework we consider the first option. The second option could be explored in future work to make security more efficient.

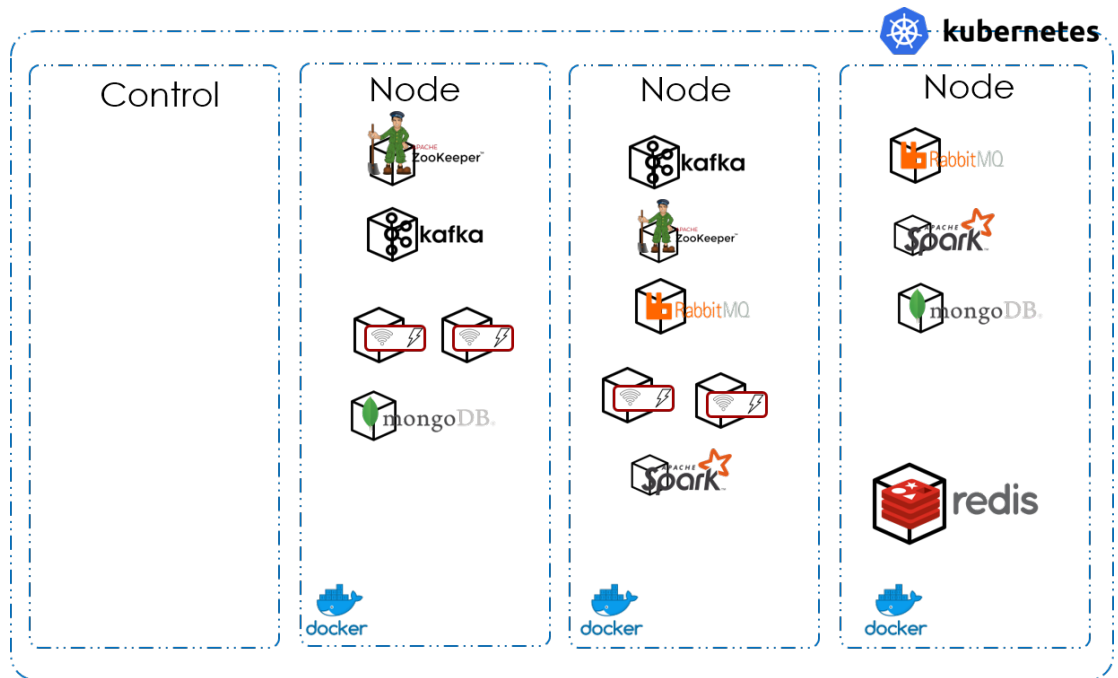


Figure 4.7: A kubernetes cloud-based translation of the framework

4.3 Cloud-based Scaling

The services and tools in the presented framework require a scalable and elastic amount of computing resources to accommodate the dynamic nature of the smart grid. Cloud-based computing provides on-demand access to these resources, scalable with minimal effort. To this end, each service or tool in managing smart grid data is containerized into docker containers. The collection of services (containers) is managed and orchestrated by Kubernetes. Aside from managing upward and downward scaling of services, Kubernetes also participates in security management. Kubernetes Secrets stores private keys and manages certificates. Furthermore, Kubernetes ConfigMap holds variables passed to running containers to modify service configurations. Figure 4.7 shows a translation of the presented framework into a cloud-based environment.

Name	Role	Internal-IP	OS-Image	Container- Runtime
casa39	master	10.31.2.22	Ubuntu 16.04.6 LTS	docker 19.3.12
casa49	worker	10.31.2.21	Ubuntu 16.04.6 LTS	docker 19.3.12
casa50	worker	10.31.2.20	Ubuntu 16.04.6 LTS	docker 19.3.12

Table 4.3: CASA Kubernetes Cluster

4.4 Implementation

The design of the framework can be interpreted and implemented in varied configurations, with possible tool substitutions based on context of use. This section of the thesis presents a micro-service implementation of the framework. The implementation is expanded and applied to the use-cases in Chapter 5 for an evaluation of the framework.

4.4.1 Hardware Environment

This thesis uses a three-node Kubernetes cluster on multi-core servers. Each server connects to the Centre for Advanced Studies-Atlantic laboratory network and is allocated a static IP address. The cluster (Table 4.3) is bootstrapped with Kubeadm [35] version 1.18.4. The specification of each server is as follows:

- **CPU:** Six Intel i7-8700 cores, two hardware threads per core
- **Memory:** 32 GiB
- **Disk:** 500 GiB

Software	Version	Instances	CPU (per in- stance)	RAM (per in- stance)
Kafka	v 2.8.0	3	1.5	4
Zookeeper	v 3.6.3	3	1	4
Redis	v 2.8.0	1 master, 2 replicas	1	2
RabbitMQ	v 3.8.19	3	2	8
Spark	v 3.1.2	1 driver, 2 executors	2	4
MongoDB	v 5.0	3	-	-
Python	v 3.8		-	-
C++	v 11, 14, 17	-	-	-
Docker	v 19.3.12	-	-	-
OmNeT++	v 5.6.2	-	-	-
OpenDSS	v 9.3.0.1	-	-	-

Table 4.4: Software Environment

4.4.2 Software Environment

Table 4.4 presents the basic details of the software tools, compilers, packages and runtime environment utilized in this thesis. In some cases, for high availability, a software may run multiple instances/container in a cluster, each one allocated a specific amount of resources. At the time of this research, version 2.8.0 of Kafka supports deployment without Zookeeper, which is documented to be faster. But we deployed with Zookeeper because standalone Kafka deployment is new, unstable and has limited support for the production environment.

4.4.3 Message Structure

Each message passed between services has a body and a header. The header contains general information and message meta-data, the schema version for schema evolution, the message origin and destination and serialized message body. The header also contains meta data on the encryption of the message body. This separation has

some obvious advantages. First, this structure is agnostic to the serialization and deserialization format. It allows a mix and match of formats for the message header and body. For example the header could be serialized in JSON for easy and schema-less passing of meta-data and the body is more efficiently serialized in Proto format. Second, it supports easy schema change, each message carries information on how to decrypt and deserialize it. However, one disadvantage is the header overhead. Listings 4.1 and 4.2 show a representation of the structure in JSON and Proto. Version 3.15.8 of the Protocol buffer library was used for code generation in C++ and Python.

Listing 4.1: Message Structure in Proto

```
syntax = "proto3";

import 'google/protobuf/timestamp.proto';

message Bundle {

    // header
    google.protobuf.Timestamp src_timestamp = 2;
    string src_id = 1;
    string dest_id = 3;
    bool encrypted = 4;
    string schema_version = 6;

    //body
    bytes data = 5;
}

message Payload {
    string src_id = 1;
    string dest_id = 2;
    string event_name = 3;
    string event_type = 4;
    string element_identifier = 5;
    google.protobuf.Timestamp timestamp = 6;
    string value = 7;
# value could be repeated data type (array)
}
```

Listing 4.2: Message Structure in JSON

```

payload = {
    "src_id": string ,
    "dest_id": string ,
    "src_id": string
    "event_name": string ,
    "event_type": string ,
    "element_identifier": string ,
    "timestamp": int ,
    "value": string | array
}

bundle = {
    "src_id": string ,
    "dest_id": string ,
    "encrypted": boolean ,
    "data": string_of_bytes
}

```

4.4.4 Micro-services, Supporting Libraries and Modules

In this section, we provide details on the services, supporting libraries and modules in the implementation described in this thesis.

- **KafkaClient and KafkaAdminClient:** The *KafkaClient* class is an abstraction that provides methods for sending and receiving serialized messages to and from Kafka. The *KafkaAdminClient* manages Kafka resources like topics and partitions. Both extend classes from the open-source Python Confluent Kafka library (v1.7.0) and are configurable based on given environment variables.
- **RabbitMQClient:** provides an abstraction for routing and receiving messages from RabbitMQ queues. Additionally, it has methods for creating, configuring and managing queues and exchanges. The C++ version of the class used with OmNeT++ is built over Copernica Marketing Software AMQP-CPP open-source library [64], and the Python version over the Pika Python open-source library v1.2.0 [52]. For high availability, and efficiency, the research uses Quorum queues [55], which operates on the Raft Algorithm.

- **E2eRSAEncryption and PublicKeyRegistry:** e2eRSAEncryption provides end-to-end asymmetric encryption for messages exchanged across services. It has methods for generating private and public keys or retrieving PEM format keys from the service environment. It is connected to the PublicKeyRegistry class to fetch public keys for destination hosts/services and publishing the public key from a service. The Python implementation is built over the open-source Pyca Cryptography library v3.4.8 [53].
- **Gateway Service:** The gateway service creates a bridge between simulation or integrated device and the aggregation layer. This abstraction allows the substitution of simulation and real devices without changing the services interacting with the aggregation layer. Another role of the gateway service is to provide bi-directional flow of commands to and from other parts of the system. In addition, this design makes data localization possible; only data sent across the gateway is available outside the simulation or integration module in Figures 4.8 and 4.9. While this design supports both real and simulation scenarios, this thesis only implements the architecture in Figure 4.9 the integration module with Distributed Energy Resources (DERs) can be further explored in future work.
- **OpenDSS Solver:** The OpenDSS Solver and Engine models and solves the power circuit and function in a co-simulation. This implementation utilizes dss-python 0.10.7.post1 library to communicate with the OpenDSS COM interface.
- **Co-ordinator/Manager:** this module manages the interpretation of messages, commands and events, and passes control to the application logic. The application logic depends on the co-simulation use case or system goal and is determined by the developer.
- **Host Startup Service:** is an initialization container that sets environment

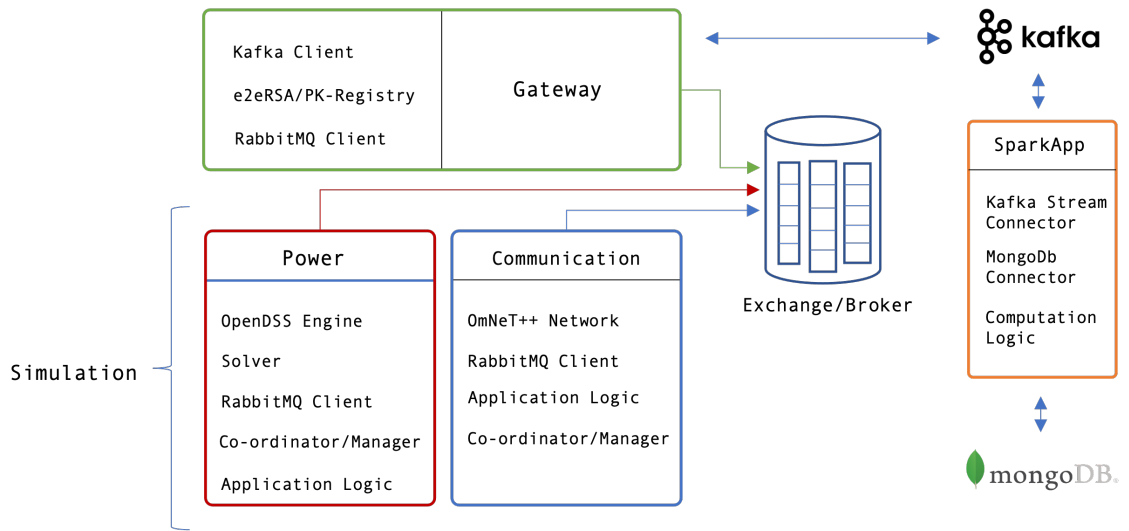


Figure 4.8: Micro-service Implementation of Framework for Simulation

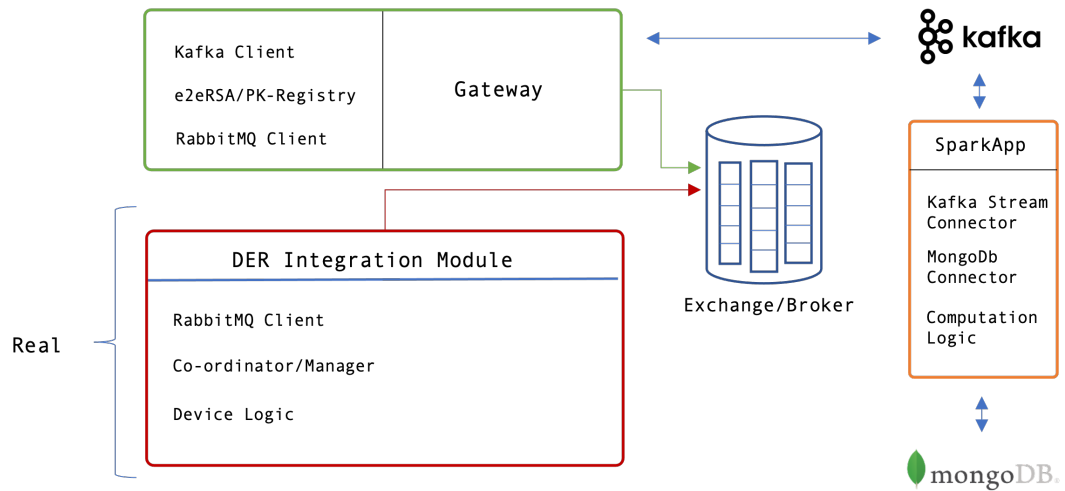


Figure 4.9: Micro-service Implementation of Framework for Real Device

variables, create topics and queues, generates keys and bootstraps resources for the host and system to run.

- **Spark Application:** is a bundle of the complex parallel computation logic (depends on use-case of the developer) and the connectors to fetch data stored in the database and streaming data from Kafka, applies computation to them and writes back the results to either. The connectors utilized here are the *kafka-sql-streaming-2.1.1:3.1*. and *org.mongodb.spark:mongo-spark-connector-2.12:3.0.1*

4.4.5 Cloud Deployment and Supporting Tools

In this section, we provide details on some of the cloud related tools and operations employed in deploying our implementation to a cloud environment.

- **OmNeT++ Base Image:** The OmNeT++ docker image (*omnetpp/omnetpp:u18.04-5.6.2*) served as a basis for running OmNeT++ in a container. We extended this image into another one by installing the required packages to run the Communication service. These packages include, protocol buffer compiler and the libuv event loop. The description of the docker image is presented in Appendix A.
- **Bitnami Redis Operator:** A Kubernetes Operator is an extension of Kubernetes for managing custom application resources and their components. It provides an easy way to automate repeated scaling and deployment tasks. We employ the Bitnami Redis Operator with the helm package manager for deploying and configuring a cluster of Redis key-value database instances. The operator chart allows management of storage and security of the Redis cluster.
- **Strimzi Kafka Operator:** Strimzi Kafka Operator allows easy creation and management of a Kafka cluster in a Kubernetes environment. It supports

configuring security, load balancing and scaling of Kafka brokers, creating and managing topics and provisioning storage. In this implementation, version 0.24.0 is used.

- **RabbitMQ Operator and GCP Apache Spark Operator:** RabbitMQ Operator provided by the RabbitMQ team allows configuration and deployment of RabbitMQ clusters in the kubernetes environment. Apache Spark Operator by Google Cloud Platform (GCP) supports running different configurations of computing workloads on Apache Spark clusters in Kubernetes. While the operator was created and is maintained by GCP, it is open-source and supports on-premise clusters and other cloud providers.
- **Docker and Deployment files:** Some of the Docker files and deployment files for deploying the above services and resources in the Kubernetes cluster are presented in Appendix A.

Chapter 5

Performance Evaluation

In this chapter, we evaluate and report on the viability and performance of the presented framework. First, we describe the performance metrics for evaluating the framework. Next, to validate the framework, three use-cases are implemented: a simple co-simulation application to monitor voltage at different points in a circuit, a wide area monitoring network with varying generation rates and a demand-response control application. We describe a base benchmark circuit, which is modified for each application. Finally, we present and discuss the results.

5.1 Performance Metrics

The metrics to evaluate the performance of our presented use case are described as follows:

- **Accuracy:** the primary goal of any simulation is to be accurate and represent the real world as closely as possible. In validating the framework, we ensure the results of the simulation match the results of the simulation without the co-simulation framework.
- **Latency:** Latency is the time taken for a message to travel from the producer

or sender to the consumer or intended destination. The smaller the latency the better. In this evaluation, the factors that affect latency include, the message format, size of message payload and throughput.

- **Throughput:** Throughput is defined as the amount of data transferred or handled in a given time-frame. The higher the throughput the better. The major factor that affects throughput is the number of instances and amount of data requested.
- **Message Size:** The size of message is critical to the latency of messages and performance of the simulation. The message size depends on the choice of message format (binary or string) and methods of encryption. Message size affects the latency and throughput; the smaller the better. Lastly, lower message sizes reduce storage needs.
- **Scalability:** is a measure of a system's agility and the ability to efficiently respond to increased workload without compromising performance, latency, throughput or response time.
- **Time Ratio:** Time ratio refers to the ratio of wall clock time to simulation time. For example, with a Real-Time Scheduler scaled 400 times, without application logic and requests, the time ratio is 0.0025. This means it takes 9 seconds to simulate one hour (3600 seconds). The time ratio affects latency and throughput. The smaller the time ratio, the faster the simulation, therefore the more messages created and sent per second.

5.2 Experiment

We designed our experiments based on an IEEE Benchmark Circuit. A part of the experiment is the co-simulation of the model of power and communication functions

of the circuit. The variables during the experiment include the message payload, the number of instances and message formats and message generation rates. Each experiment is run at least 10 times excluding warm and cold runs.

5.2.1 Test Case IEEE 13 Node Circuit

The IEEE 13-bus Feeder circuit model is used to test common features of distribution analysis software, operating at 4.16 kV. It is characterized by being short, relatively highly loaded, having a single voltage regulator at the substation, overhead and underground lines, shunt capacitors, an in-line transformer, and unbalanced loading [34]. W. Kersting et al. [34] provide detailed information on the circuit.

5.2.2 A: Simple Voltage Monitoring

In this test-case, we compare the results of a co-simulation with and without the framework to ensure that the accuracy of the simulation is not compromised. To do this, three monitors M1, M2, and M3 are placed at line 684611, load 611 and load 652 of the IEEE 13-bus feeder respectively. The monitors have corresponding nodes represented in the OmNeT++ network. Each monitor is synchronized and sampled at 15 simulation minute intervals. The simulation is run with the communication simulator as the primary and the power as the secondary. Appendix B shows the modifications made to the IEEE 13-bus feeder in OpenDSS and the OmNeT++ simulation configuration. The voltage at each node is measured over 24 simulation hours. The results of the simulation with the co-simulation framework is compared with the results without the framework. Figure 5.1 shows the sampling of the voltage at the monitors without co-simulation. Next, the communication network is run with a sequential event scheduler and the primary simulator is not paused during synchronization (Figure 5.2). This is to highlight the latency of messages in the co-simulation. The results show that the voltage readings follow the same shape and

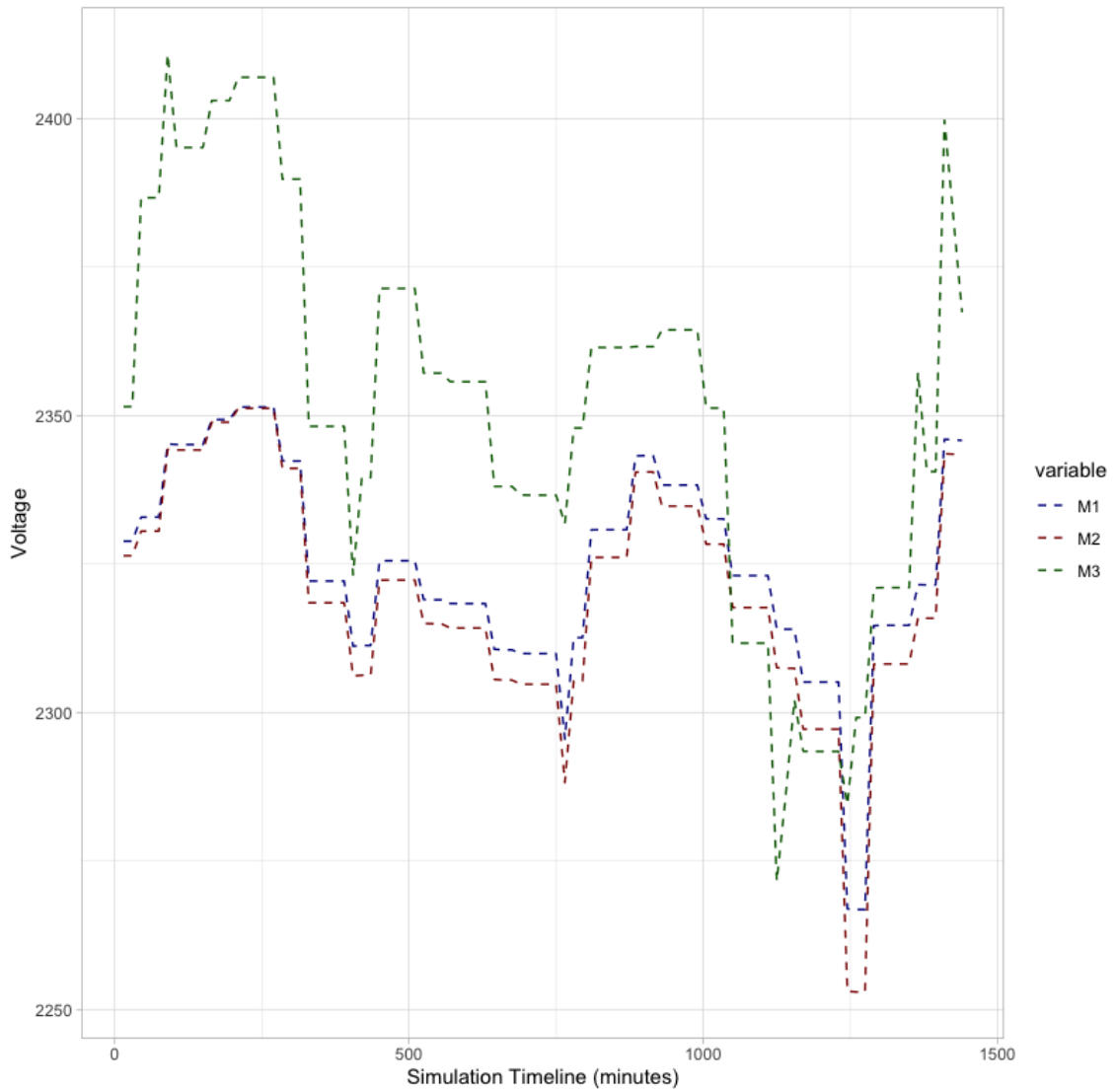


Figure 5.1: Without Co-simulation: Voltage Reading at 15 minute intervals for 24 hours

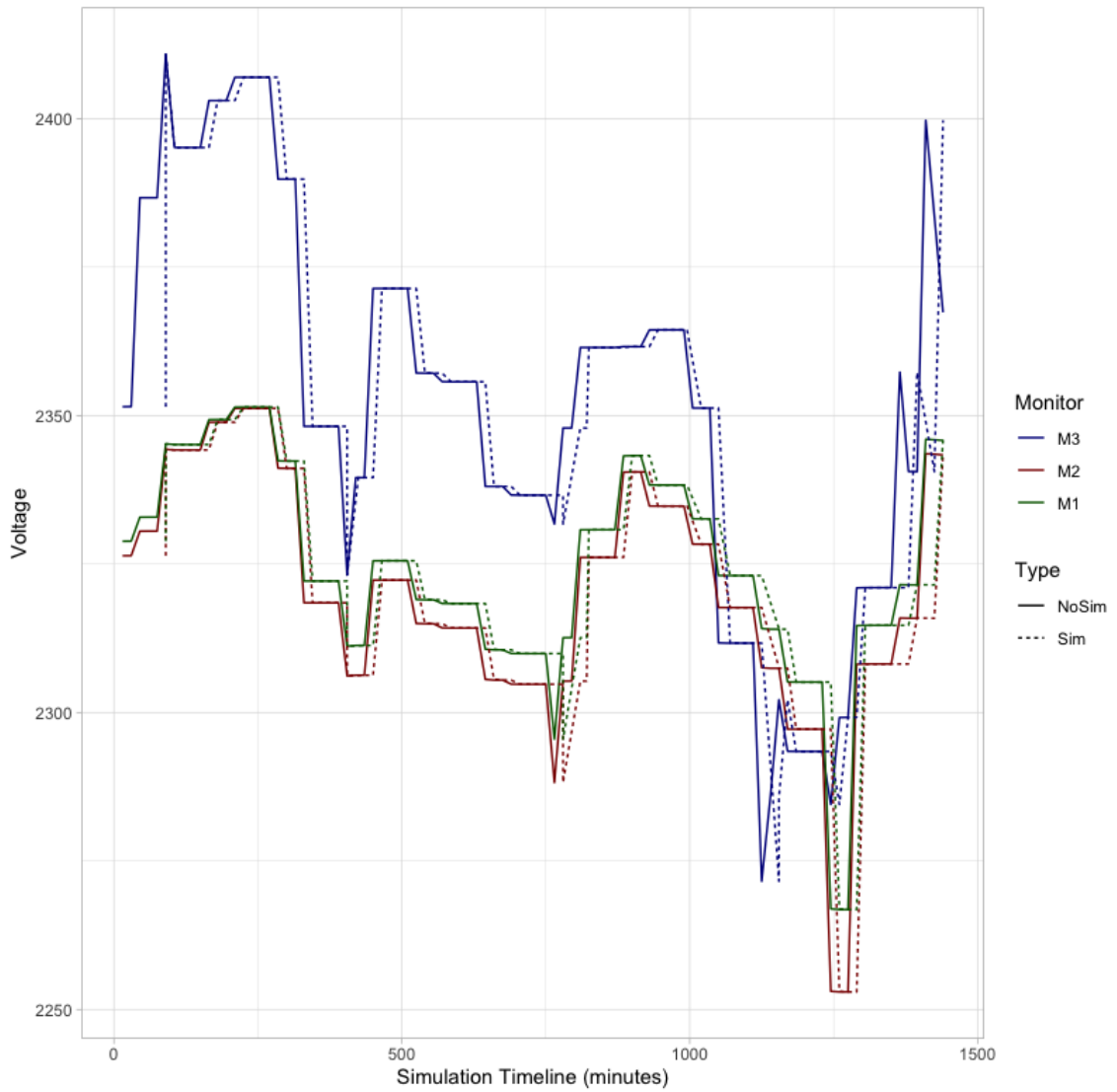


Figure 5.2: Without vs With Co-simulation(Sequential Scheduler): Voltage Reading at 15 minute intervals for 24 hours

are accurate. However, there is some latency between the request to synchronize and the receipt of the voltage reading in the primary simulator. The time t_x when a value is recorded at the primary simulator is defined as $t_x = t_{timeofrequest} + latency$, where latency is the time between request and response. This emphasizes the importance of minimizing the latency in a co-simulation. Therefore:

- If the focus is accuracy, or the simulation is not time-sensitive (i.e not real-time), or has no hardware in the loop, the latency may be less significant. Alternatively, in a time sensitive simulation, the time at the secondary simulator could be sent with every message and referenced regardless of the time at the primary simulator.
- If the simulation has hardware in the loop or is real-time, the Real-Time Scheduler, which can be scaled and synchronized with wall clock time, can be utilized. This is useful when integrating with real DERs.
- Lastly, with the Sequential Scheduler, the primary simulator may be paused at each synchronization until the request receives a response from the secondary. With this method of synchronization, like in Figure 5.3 the time-line and readings of the simulation with the co-simulation (triangle points) align with that of the simulation without the framework. However, this method may increase the time taken to run a simulation.

In all cases, the power readings or voltage shape in both simulations match, showing the accuracy of the co-simulation framework.

5.2.3 B: Wide Area Monitoring Network with Varying Generation Rates

B. Dhananjay et al. designed a co-simulator for a wide area monitoring network

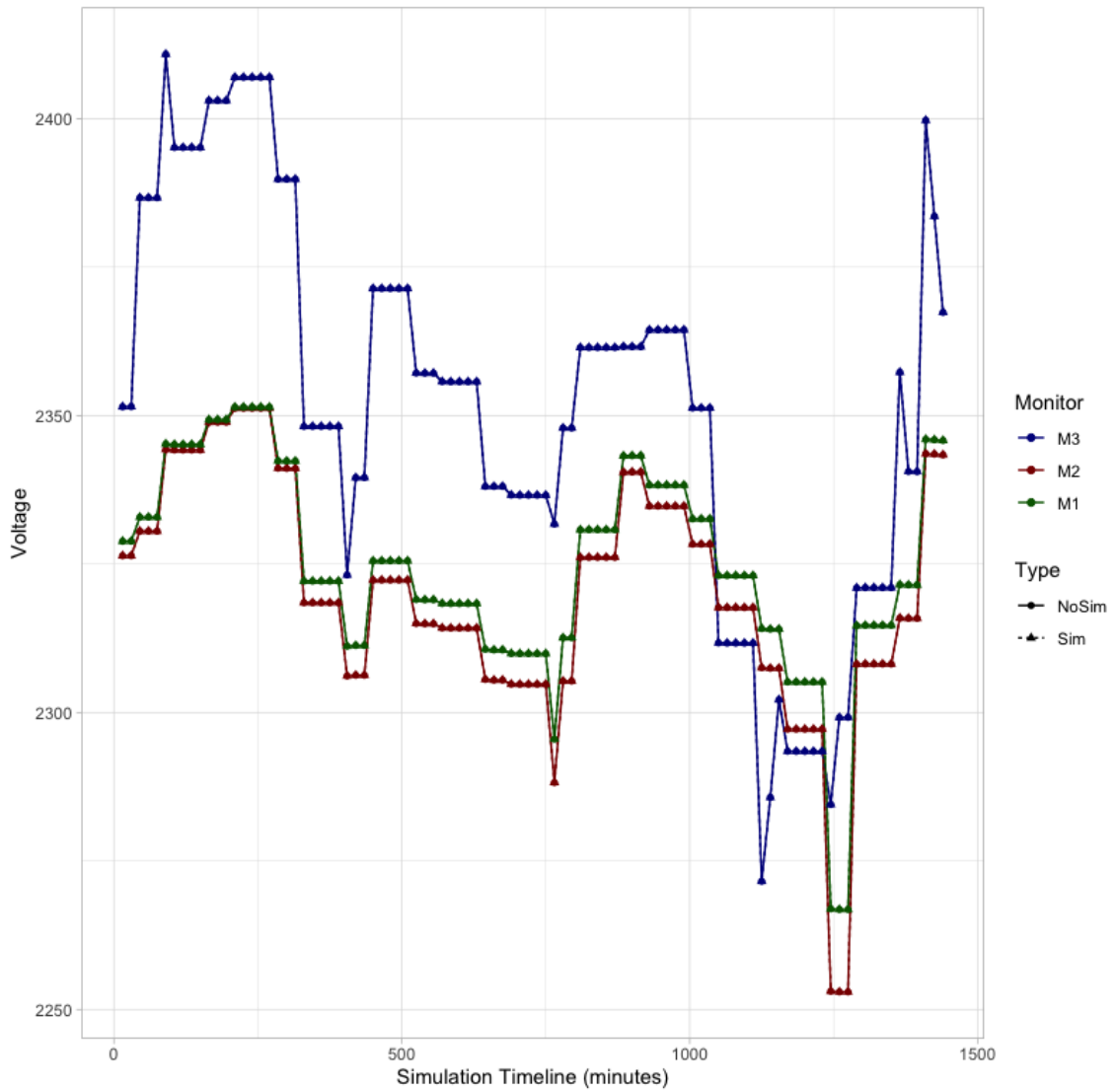


Figure 5.3: Co-simulation with pausing and Sequential Scheduler: Voltage Reading at 15 minute intervals for 24 hours

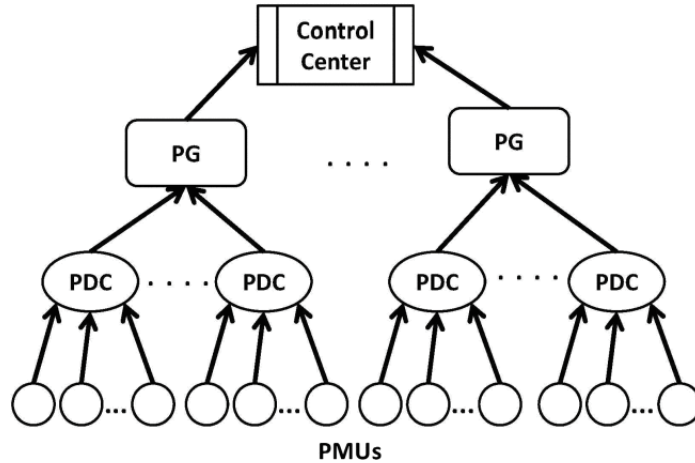


Figure 5.4: Wide Area Monitoring Network [11]

of Phasor Measurement Units (PMU) [11]. Figure 5.4 shows the architecture of the network. In the co-simulator [11], the PMU data is stored in a database by the power circuit and queried by timestamp. Each PMU fetches the data at intervals and generates the PMU packet sent to the Phasor Data Concentrator (PDC). Each PDC performs some operations on the data and forwards the data to the Phasor Gateway. B. Dhananjay et al. [11] compared the running time of the wide area monitoring system with varying PMU generation rates in a one minute simulation. In their implementation, the network is made up of 10 PMUs, 4 PDCs, and 2 PGs. The authors set the synchronization interval between the two simulators to 10 seconds. One observed bottleneck in this implementation is the large latency of PMU readings fetched at intervals. For example, if data is fetched at five second intervals, a value produced a second after a recent fetch cycle, is not delivered until the next cycle with a latency of at least four seconds.

We implement the wide area monitoring system as closely as possible, substituting the database polling in the benchmark implementation with a message-oriented middleware. We increase the number of PMUs to 12, with three attached to each PDC. Without knowledge of the specific size of the message payload in the benchmark system, we choose to send about 100 voltage values per message payload; this amounts

to between 1061 and 2400 bytes when serialized. In a study, by C. Rodriguez et al. [57], that analyzed 78GB of HTTP web and mobile traffic, the median value of JSON payloads in REST-API systems is about 1545 bytes. We assume that the value should be less in smaller IoT devices. We measure the average real time, over multiple runs, required to simulate one minute at each generation rate and compare our results with the reference implementation [11]. In Figure 5.5, both versions of our implementation of the presented co-simulation framework based on a message-oriented middleware outperform the system with database polling. We observe a huge difference in the average time. Aside the efficiency and performance of the queue-based middleware over database polling, the difference also depends on the nature of aggregation at each level of the wide area monitoring network. In the same figure, we observe that serialization with a binary based Protocol buffer performs better than the system with JSON serialization at this payload size.

Next, using the same implementation we investigate the effect of higher workload and throughput on the average latency of messages. The goal is to evaluate the scalability of the framework. The latency in this case refers to the response time defined as the time from the data request to response delivery. This response time includes the message round-trip and the processing time at the destination of the data request. The PMU data generation rate per unit time or simulation throughput is varied from one data request message per PMU per second to 64 messages per PMU per second. Hence, for a sampling rate of 50, the messages per second is 600. From Figures 5.6, and 5.7 we observe the following:

- a. The result supports the fact that, in this case, the MOM with Protocol buffers performs better than JSON across the varied sampling rates with lower latency.
- b. The relationship between the latency of messages and the throughput (messages per second) of the co-simulator is linear. As the workload increases the change in average latency increases accordingly.

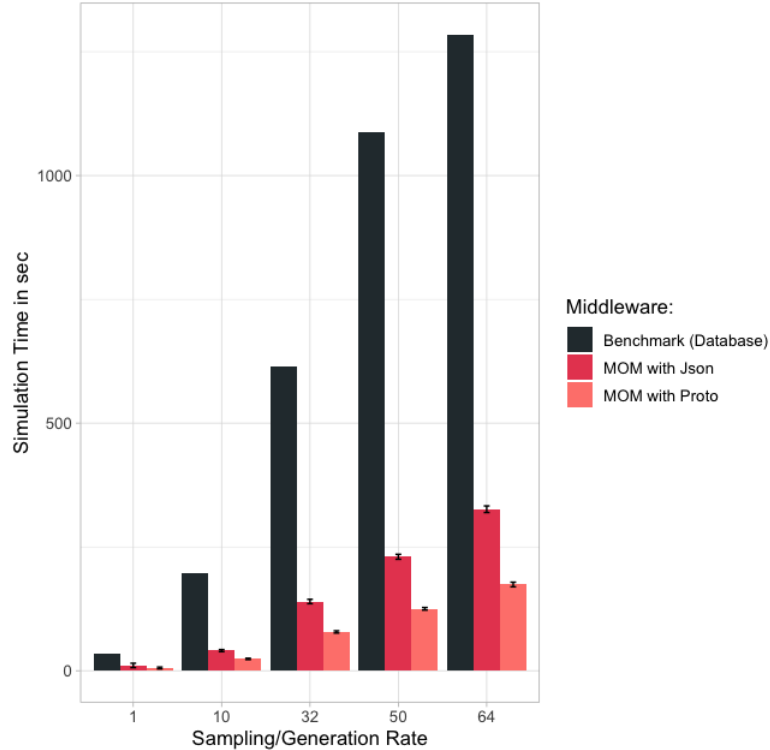


Figure 5.5: Time required to simulate one minute of simulation time with Message-oriented and Database Middleware

- c. While the relationship between latency and throughput is linear as shown in Figure 5.7, the average increase in latency with respect to throughput is greater in the JSON based system.
- d. While error bars representing standard deviation from the mean latency are large, this deviation is $\pm 50 - 150$ ms which could be caused by processing time at the data request target, network fluctuations on the host machine, garbage collection in the middleware, flushing of messages from memory to disk or queue data replication in the middleware¹. The solution with Protocol buffers is more stable from the number of outliers observed in Figure 5.6. We investigate this further in Sub-section 5.2.3.2.

¹For high availability and fault tolerance, queues configured to be replicated in multiple nodes of a RabbitMQ cluster.

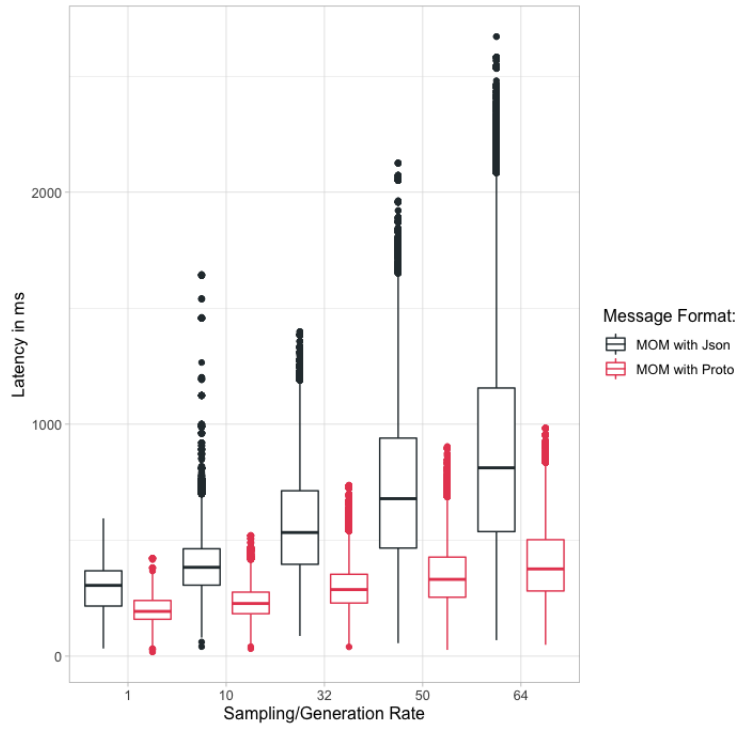


Figure 5.6: Message Latency of message with varied messages per second

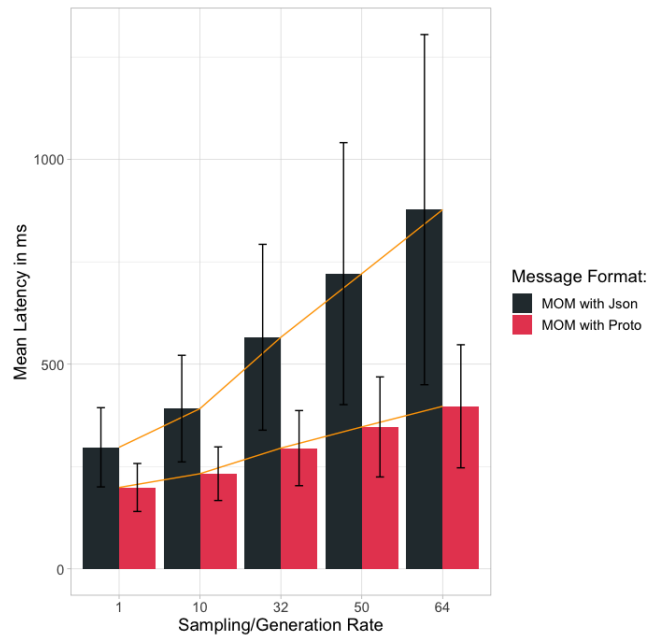


Figure 5.7: Average Latency of Message-oriented Middleware (MOM) with JSON and Protocol buffer with varied messages per second

5.2.3.1 Scaling

To evaluate the scalability of the framework, we run multiple instances of the wide area monitoring co-simulation simultaneously and observe the effect on the performance of each simulation. Each instance is run for one minute of simulation time, with a PMU sampling rate of 10. We run multiple instances at once with a scale factor of two from one to eight instances. With multiple instances the number of messages per second increases. For example, with two simultaneous instances, at a sampling rate of 10, the messages per second is 240. The results, Figure 5.8, reveal that the average time of each running simulation increases as the number of simultaneously running instances increase. The standard deviation of the simulation time is greater with multiple instances running but remains ± 5 seconds at most. The message-oriented middleware efficiently handles the increase in throughput with more instances sending more messages per second. In database polling middleware, for every query/poll to select new data, the size of the table affects the lookup time as the query filters a larger dataset with more instances. In contrast, in a queue-based message oriented system, only the unconsumed messages in the queue are served to the simulator or client. Additionally, in the absence of new data, the queue-based system does not make empty round trips to the data source. While only 12 PMUs are considered, the important scalability metric is the number of messages that can be handled in a unit of time. In many real-world smart grid systems, a smart meter produces one reading between 15 minutes to 8 hour intervals [4]. We can therefore translate scalability of messages handled per second to number of devices.

5.2.3.2 Message Size

Many studies have reported various smart meter message packet sizes: T. Shiobara et al. reports 150 bytes [60], B. Karimi et al., 20 to 500 bytes [33], D. Hartmann et al., 20 to 2,133 bytes [29] and M. Jaber. reports a maximum of about 145,000

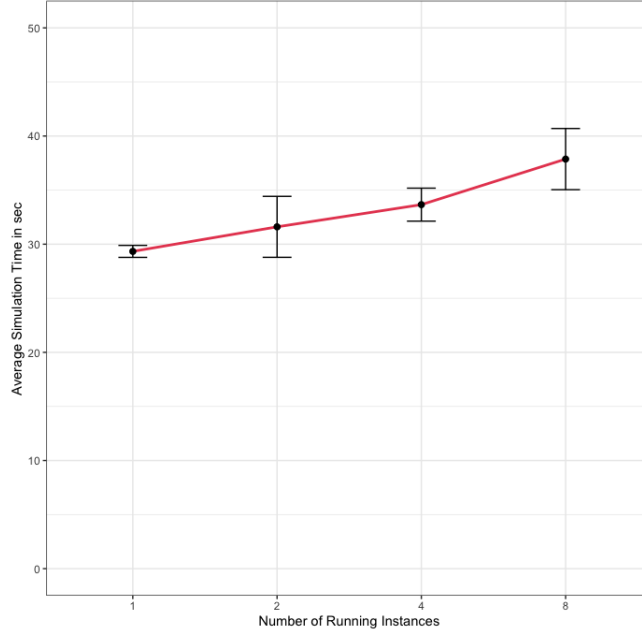


Figure 5.8: Average real time to simulate one minute with varying number of running instances at 10 samples a second

bytes [30]. We consider different message sizes to ensure our framework can handle different smart meter (or grid device) scenarios. In the experiments in previous sections, the Protocol buffer performs better than JSON in the MOMs. Hence, we decide to investigate the effect of the choice of message format on the message size and therefore, the scalability of the framework based on increase in payload, message size and choice of serialization and deserialization format. We run the wide area monitoring co-simulator with a PMU generation rate at two bound, 10 and 50, we vary the number of voltage values sent in each message payload and observe the effect on the size of the message and in turn, simulation time. Figure 5.9 shows the message sizes with 100 to 10,000 values in each payload. First, overall, the message sizes of JSON messages are observed to be more. Second, the change in message size with more values is less in Protocol buffers. This implies that in a simulation or smart grid system with large workloads, Protocol buffers may be more efficient in terms of storage. However, in the results (Figure 5.10) of the effect on simulation

time, we observe the following:

- At a sampling rate of 10 (120 messages per second), with JSON-based messages, the simulation time is longer at 100 values per message, but it performs better with an increase in the number of messages per payload.
- With a larger number of messages per second, Protocol buffers perform better from 100 - 10,000 messages per second. There is a steep increase in simulation time with JSON-based messages at 1,000 messages per second. We believe this is because the average change in message size with the number of samples is greater with JSON. Also, based on the design of RabbitMQ employed for the MOM, with larger message sizes, messages are written from memory to disk as the queue size increases [16]. At lower sampling rates, this effect is not noticed as messages are processed off the queue faster than at higher sampling rates.
- We conclude that: while JSON performs better in some cases, Protocol buffers are a better choice, because of the performance at higher messages per second and lower message sizes for storage.
- Based on the results, at higher throughputs, it is better and more efficient to send multiple smaller data response messages than one large message. For example, sending two messages with 5,000 values instead of a single message with 10,000 values.

5.2.4 C: Cloud-based Demand-Response and Predictive Control

This experiment is targeted at evaluating the analytics and cloud-based section of the co-simulation framework. We modify the benchmark circuit to include generators and monitors on a specific load (Figure 5.11). A monitor is attached at load 611

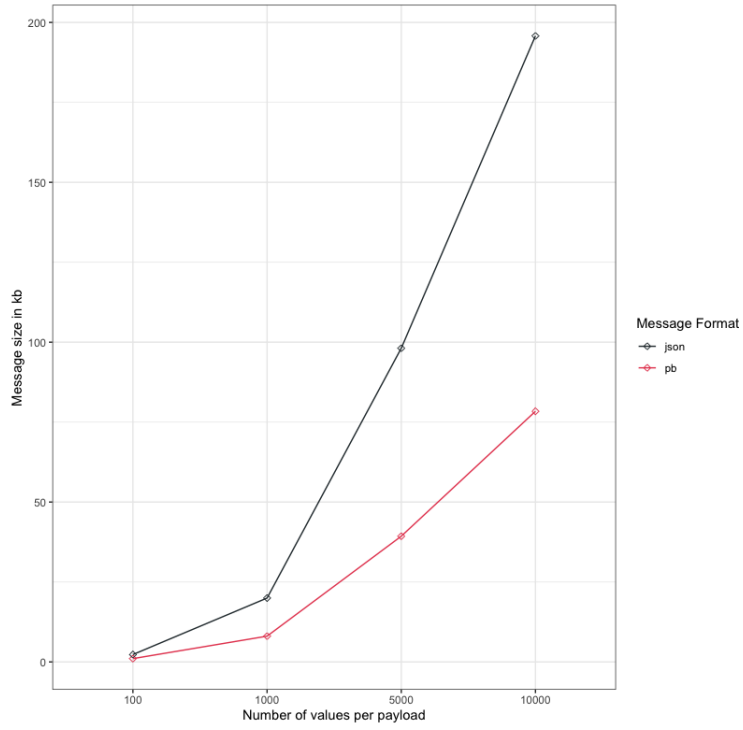


Figure 5.9: Effect of number of values in a message payload on message size

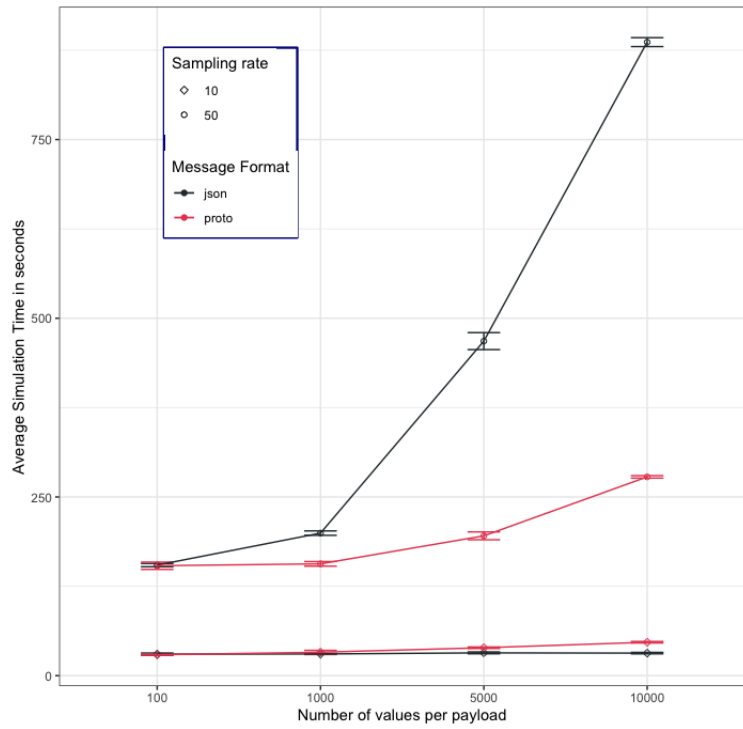


Figure 5.10: Effect of message size on simulation time

to monitor its voltage and a supporting distributed generator, which is initially OFF or inactive on the same bus. The modification to the model in OpenDSS is shown in Appendix C. The voltage at the node is synchronized and monitored in the communication network. The voltage information is streamed through the gateway and the aggregator to the Spark Analytics application. The Spark Application builds or loads an existing logistic regression model from historical data loaded from a MongoDB Database. The Spark Application applies the model to the current voltage readings in the streaming data to decide on an action or command. The command in this evaluation is limited to turning off or turning on the generator to offset high or low voltage at the node.

5.2.4.1 Modelling the Logistic Regression Model

The control algorithm in the demand-response application is based on a simple minimum and maximum threshold. This is similar to the controller algorithm presented by Dhananjay et al. [11]. The difference is that the minimum and maximum control is modelled using logistic regression. A generator is toggled when the voltage is out of bounds of ± 50 of a base voltage of 2,450. We are aware that the algorithm can be implemented with if and else statements. However, we used a model to demonstrate the versatility of the framework. To train the model to do this, we generate a dataset to model this behaviour including the action to take for a particular reading. The data generation script is presented in Appendix C. The format of each row and some examples is presented in Table 5.1. In the dataset label column, 0 stands for Do Nothing, 1 for ON and 2 for OFF. The data is loaded into the MongoDB Database. During training, multiple iterations were made over the dataset to produce an accurate model.

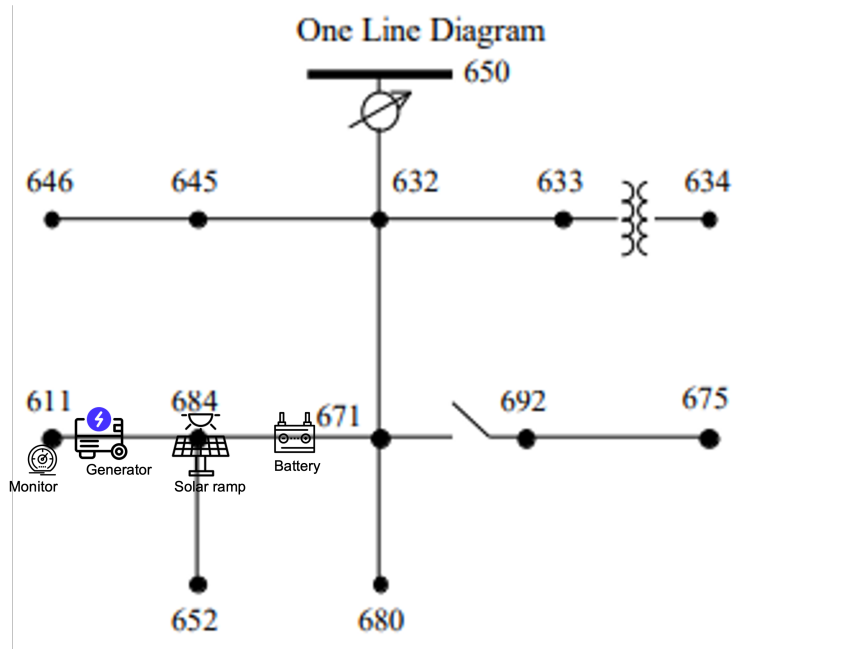


Figure 5.11: A modified IEEE 13-bus feeder with generator and a monitor

Table 5.1: A sample dataset for a minimum and maximum threshold logistic regression model

Voltage	label	command
2400	0	Nothing
2349	1	Generator ON
2451	2	Generator OFF

5.2.4.2 Results

From this experiment, first, we show that the framework supports a bi-directional flow of data and commands in a cloud environment, which is crucial in modern smart grid simulation and real systems. Further we show one simple application in demand-response systems. Figure 5.12 shows the voltage or load shape recorded at Load 611 without the co-simulation or cloud-based demand and response. A voltage drop below the threshold is observed between the 100th and 160th seconds. With the demand-response application in co-simulation (Figure 5.13), the controller consisting of the logistic algorithm applied to the readings in real-time detects a low voltage and issues the command to start the generator. With the generator, the

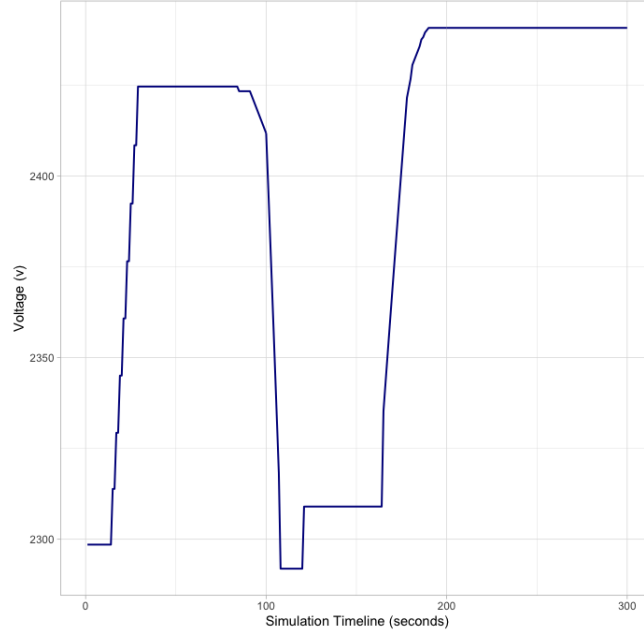


Figure 5.12: Voltage reading at Load 611 for 300 seconds without communication co-simulation

voltage is balanced and kept within the threshold. The bi-directional flow of data and commands with the presented framework is not limited to simple scenarios like the toggling of a generator to balance voltage in a power flow analysis in the demand-response application; it supports more complex functions including but not limited to fault resolution in the power circuit model in a fault study or modifying the power simulation parameters based on input through the communication simulator.

5.3 Summary

In this chapter, we described some metrics for evaluating our framework. We designed and implemented three cases for testing. First, we proved that the co-simulation framework is accurate and can be used with different types of synchronization based on type of simulation, time-insensitive, real-time or hardware in the loop. Second, we showed the framework is easy to scale in a cloud environment,

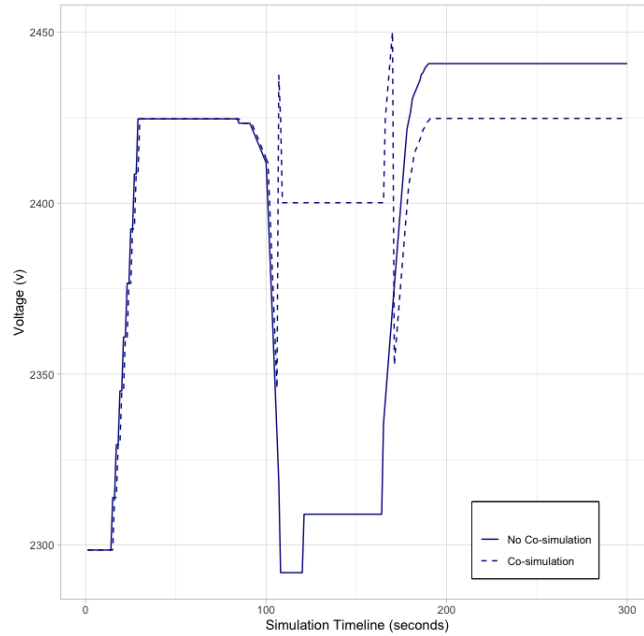


Figure 5.13: Voltage at Monitor with cloud-based demand-response with analytics section of co-simulation framework

running multiple instances of co-simulation. In addition, we showed how an increase in messages per second and how message size affects simulation time. It is better to send smaller messages than single large messages for better performance. Lastly, we show the capability of the cloud-based analytics of the framework by using analytics of historical data and current data to balance the voltage demand and response.

Chapter 6

Conclusions and Future Work

This chapter contains a summary of the project problem statement, objective, solution, evaluation and limitations.

6.1 Summary

We started this project by identifying the differences between the conventional grid and an advanced smart grid with information and communication technology. Further, we identified two major needs: one, the need for co-simulation and efficiently handling communication in a co-simulation environment; two, efficiently managing the large volume of data in the smart grid environment. To this end, we proposed a cloud-centric scalable framework that supports three functions: to efficiently handle data communication in a smart grid system, real, co-simulation and/or both; to enable a bi-directional flow of data and commands between simulation and real systems; and to securely manage aggregation, analytics, and storage of smart grid data. We designed, implemented and presented a framework after comparing different technologies. In our framework, we considered two top serialization and deserialization formats, JSON and Protocol buffer. Next, we designed three co-simulation use-cases based on a benchmark IEEE circuit to evaluate the use, scale and accuracy of the

framework. From the evaluation, we proved the accuracy of our framework with results matching the values of a system without the framework. We showed, in a wide area network monitoring applications with Phasor Measurement Units, that the framework can handle small and larger workloads, and running multiple instances of co-simulation is easy to scale in a cloud environment. In our comparison of data formats for our middleware, while JSON performed better than Protocol buffers in some cases and worse at higher messages per second, we conclude that Protocol buffer is a better choice. In addition, using lines of code as a metric, we noticed that the development effort in using Protocol buffer is greater than using JSON. Lastly, with a demand-response application, we showed the use of analytics for keeping the voltage at a power node balanced using a logistic regression model. In addition, this proved the support for cloud-based bi-directional flow of data and command. Our evaluation showed that we achieved the objective set out by the research.

6.2 Limitations and Future Work

Real life smart grid applications are not limited to demand response systems, they range from simple to complex studies including power flow analysis, customer behavior grouping, fault study, smart grid management and monitoring. We believe our framework can support these applications. However, the evaluation of our analytics section only demonstrates a simple example of threshold predictive control using a logistic regression model. In future work, applications with more complex machine learning algorithms like Support Vector Machine, K-Clustering and Gradient Boost should be explored. In a simple application with simple computation, the latency and processing time may be low, however as queries and computation get complex for different applications, the processing time increases and may affect the latency or response time. To address this issue, windowed queries, which perform computation

over data in time windows could be considered. Also, the chosen middleware and analytics tool are highly configurable. These tools could be tuned to provide better performance based on the use-case. The reusable and extendable implementation modules in this thesis provides a foundation for building other applications. In addition, the data utilized in our evaluation of the analytics is synthetic and may not fully represent real-life data. Future work should explore using publicly available datasets in building applications that require data training.

The design and implementation of our framework supports real-life systems and simulation. It also supports a bridging data and communication between them. However, this thesis evaluates simulation and co-simulation. The software clients, gateway service and the design in Figure 4.9 provide a foundation for future work to explore the Distributed Energy Resource (DER) integration modules. One major challenge in integrating real hardware with simulation would be the time resolution. Various hardware devices and simulators use different time resolution. The Real Time Scheduler in OmNeT++ and the synchronization technique serve as bases for addressing this potential issue. However, we observed that the limited support for multi-threading in OmNeT++ could be a bottleneck.

In our framework, we employed RSA for end-to-end encryption. This method is subject to computational overhead and ciphertext expansion. To further optimize data security in the framework, future work could explore a combination of symmetric and asymmetric encryption. With this method, a symmetric encryption algorithm like Advanced Encryption Standard (AES) is utilized for message encryption because of its speed and to mitigate the computation overhead of asymmetric encryption (RSA). To manage the key, RSA is used to encrypt the symmetric key and is passed with every message or exchanged in a handshake between the two communicating services. However, sending the key with every message may add an overhead to the size of each message header. For example, using a 128-bit key adds about 16 bytes.

As a basis, the interface of the E2eRSAEncryption and PublicKeyRegistry classes presented in our implementation could be implemented and/or extended to use this method.

Lastly, our project was evaluated on an on-premise cluster of machines. Future work should explore cloud providers such as Amazon Web Service, Google Cloud and IBM Cloud. The deployments provided in our framework are reproduceable in these environments with minor configuration changes.

Bibliography

- [1] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash, *Internet of things: A survey on enabling technologies, protocols, and applications*, IEEE communications surveys & tutorials **17** (2015), no. 4, 2347–2376.
- [2] Ahmad T Al-Hammouri, *A comprehensive co-simulation platform for cyber-physical systems*, Computer Communications **36** (2012), no. 1, 8–19.
- [3] Michele Albano, Luis Lino Ferreira, Luís Miguel Pinho, and Abdel Rahman Alkhawaja, *Message-oriented middleware for smart grids*, Computer Standards & Interfaces **38** (2015), 133–143.
- [4] Nikoleta Andreadou, Evangelos Kotsakis, and Marcelo Masera, *Smart meter traffic in a real lv distribution network*, Energies **11** (2018), no. 5, 1156.
- [5] Mohammad Hasan Ansari, Vahid Tabatab Vakili, and Behnam Bahrak, *Evaluation of big data frameworks for analysis of smart grids*, Journal of Big Data **6** (2019), no. 1, 1–14.
- [6] ———, *Evaluation of big data frameworks for analysis of smart grids*, Journal of Big Data **6** (2019), no. 1, 1–14.
- [7] The Kubernetes Authors, *Kubernetes concepts and architecture*, <https://kubernetes.io/docs/concepts/>, 2021, (accessed: 05. 05.2021).

- [8] Ahmet Aydin and Ken Anderson, *Batch to real-time: Incremental data collection & analytics platform*, (2017).
- [9] Ali Kashif Bashir, Suleman Khan, B Prabadevi, N Deepa, Waleed S Alnumay, Thippa Reddy Gadekallu, and Praveen Kumar Reddy Maddikunta, *Comparative analysis of machine learning algorithms for prediction of smart grid stability*, International Transactions on Electrical Energy Systems (2021), e12706.
- [10] Samaresh Bera, Sudip Misra, and Joel J.P.C. Rodrigues, *Cloud computing applications for smart grid: A survey*, IEEE Transactions on Parallel and Distributed Systems **26** (2015), no. 5, 1477–1494.
- [11] Dhananjay Bhor, Kavinkadhirsvelan Angappan, and Krishna M Sivalingam, *A co-simulation framework for smart grid wide-area monitoring networks*, 2014 Sixth International Conference on Communication Systems and Networks (COMSNETS), IEEE, 2014, pp. 1–8.
- [12] ———, *A co-simulation framework for smart grid wide-area monitoring networks*, 2014 Sixth International Conference on Communication Systems and Networks (COMSNETS), IEEE, 2014, pp. 1–8.
- [13] Anna Borucka, *Application of the logistic regression model to study customer loyalty in an online store*, Proceedings of the 3rd International Conference on Business and Information Management (New York, NY, USA), ICBIM '19, Association for Computing Machinery, 2019, p. 21–26.
- [14] Bill Chambers and Matei Zaharia, *Spark: The definitive guide: Big data processing made simple*, ” O’Reilly Media, Inc.”, 2018.
- [15] Selim Ciraci, Jeff Daily, Khushbu Agarwal, Jason Fuller, Laurentiu Marinovici, and Andrew Fisher, *Synchronization algorithms for co-simulation of power grid and communication networks*, 2014 IEEE 22nd International Symposium on

- Modelling, Analysis & Simulation of Computer and Telecommunication Systems, IEEE, 2014, pp. 355–364.
- [16] CloudAMQP, *Rabbitmq best practices*, <https://www.cloudamqp.com/blog/part4-rabbitmq-13-common-errors.html>, 2021, (Accessed on 09/16/2021).
- [17] Andrei Cornel Cristian, Tudor Gabriel, Madalina Arhip-Calin, and Alexandru Zamfirescu, *Smart home automation with mqtt*, 2019 54th International Universities Power Engineering Conference (UPEC), 2019, pp. 1–5.
- [18] Houda Daki, Asmaa El Hannani, Abdelhak Aqqal, Abdelfattah Haidine, and Aziz Dahbi, *Big data management in smart grid: concepts, requirements and implementation*, *Journal of Big Data* **4** (2017), no. 1, 1–19.
- [19] Andrea De Mauro, Marco Greco, and Michele Grimaldi, *A formal definition of big data based on its essential features*, *Library Review* (2016).
- [20] Jens Dede, Koojana Kuladinithi, Anna Förster, Okko Nannen, and Sebastian Lehnhoff, *Omnet++ and mosaik: enabling simulation of smart grid communications*, arXiv preprint arXiv:1509.03067 (2015).
- [21] Docker, *Docker*, <https://www.docker.com/>, 2021, (accessed: 05. 05.2021).
- [22] EPRI, *Epri, open distribution system simulator*, <https://smartgrid.epri.com/SimulationTool.aspx>, 2021, (accessed: 01. 09.2020).
- [23] Baling Fang, Xiang Yin, Yi Tan, Canbing Li, Yunpeng Gao, Yijia Cao, and Jianliang Li, *The contributions of cloud technologies to smart grid*, *Renewable and Sustainable Energy Reviews* **59** (2016), 1326–1331.
- [24] Hassan Farhangi, *The path of the smart grid*, *IEEE power and energy magazine* **8** (2009), no. 1, 18–28.

- [25] Jason C Fuller, Selim Ciraci, Jeffrey A Daily, Andrew R Fisher, and M Hauer, *Communication simulations for power system applications*, 2013 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), IEEE, 2013, pp. 1–6.
- [26] Hanno Georg, Sven Christian Müller, Nils Dorsch, Christian Rehtanz, and Christian Wietfeld, *Inspire: Integrated co-simulation of power and ict systems for real-time evaluation*, 2013 IEEE International Conference on Smart Grid Communications (SmartGridComm), IEEE, 2013, pp. 576–581.
- [27] Google, *Protocol buffers*, <https://developers.google.com/protocol-buffers>, 2021, (accessed: 05. 05.2021).
- [28] Vehbi C Gungor, Dilan Sahin, Taskin Kocak, Salih Ergut, Concettina Buccella, Carlo Cecati, and Gerhard P Hancke, *Smart grid technologies: Communication technologies and standards*, IEEE transactions on Industrial informatics **7** (2011), no. 4, 529–539.
- [29] Detlef Hartmann, Katinka Wolter, and Tilman Krauss, *Ict resilience simulations in small confined smart distribution grids*, Intelec 2013; 35th International Telecommunications Energy Conference, SMART POWER AND EFFICIENCY, VDE, 2013, pp. 1–6.
- [30] Mona Jaber, Nour Kouzayha, Zaher Dawy, and Ayman Kayssi, *On cellular network planning and operation with m2m signalling and security considerations*, 2014 IEEE International Conference on Communications Workshops (ICC), IEEE, 2014, pp. 429–434.
- [31] JSON, *Json*, <https://www.json.org/json-en.html>, 2021, (accessed: 05. 05.2021).
- [32] Supun Kamburugamuve, Leif Christiansen, and Geoffrey Fox, *A framework for real time processing of sensor data in the cloud*, Journal of Sensors **2015** (2015).

- [33] Babak Karimi, Vinod Namboodiri, and Murtuza Jadliwala, *Scalable meter data collection in smart grids through message concatenation*, IEEE Transactions on Smart Grid **6** (2015), no. 4, 1697–1706.
- [34] William H Kersting, *Radial distribution test feeders*, IEEE Transactions on Power Systems **6** (1991), no. 3, 975–985.
- [35] Kubernetes, *Kubeadm*, <https://kubernetes.io/docs/reference/setup-tools/kubeadm/>, 2021, Accessed 05. 05. 2021.
- [36] Martin Lévesque, Da Qian Xu, Géza Joós, and Martin Maier, *Communications and power distribution network co-simulation for multidisciplinary smart grid experimentations*, Proceedings of the 45th Annual Simulation Symposium, 2012, pp. 1–7.
- [37] Ninghui Li, *Asymmetric encryption*, pp. 142–142, Springer US, Boston, MA, 2009.
- [38] Weilin Li, Mohsen Ferdowsi, Marija Stevic, Antonello Monti, and Ferdinanda Ponci, *Cosimulation for smart grid communications*, IEEE Transactions on Industrial Informatics **10** (2014), no. 4, 2374–2384.
- [39] Vincenzo Liberatore and Ahmad Al-Hammouri, *Smart grid communication and co-simulation*, IEEE 2011 EnergyTech, IEEE, 2011, pp. 1–5.
- [40] Hua Lin, Santhosh S Veda, Sandeep S Shukla, Lamine Mili, and James Thorp, *Geco: Global event-driven co-simulation framework for interconnected power system and communication network*, IEEE Transactions on Smart Grid **3** (2012), no. 3, 1444–1456.
- [41] Peter Lipčák, Martin Macak, and Bruno Rossi, *Big data platform for smart grids power consumption anomaly detection*, 2019 Federated Conference on Computer Science and Information Systems (FedCSIS), IEEE, 2019, pp. 771–780.

- [42] Peter Mell, Tim Grance, et al., *The nist definition of cloud computing*, (2011).
- [43] Markus Mirz, Lukas Razik, Jan Dinkelbach, Halil Alper Tokel, Gholamreza Alirezaei, Rudolf Mathar, and Antonello Monti, *A cosimulation architecture for power system, communication, and market in the smart grid*, Complexity **2018** (2018).
- [44] MQTT, *Mqtt*, <https://mqtt.org/faq/>, 2021, (accessed: 05. 05.2021).
- [45] Neha Narkhede, Gwen Shapira, and Todd Palino, *Kafka: the definitive guide: real-time data and stream processing at scale*, " O'Reilly Media, Inc.", 2017.
- [46] Erik Nilsson and Victor Pregén, *Performance evaluation of message-oriented middleware*, 2020.
- [47] OMNeT++, *Omnet++ discrete event simulator*, <https://omnetpp.org>, 2020, (accessed: 01. 09.2020).
- [48] David Oudart, Jérôme Cantenot, Frédéric Boulanger, and Sophie Chabridon, *An approach to design smart grids and their it system by cosimulation*, MODELSWARD 2019: 7th International Conference on Model-Driven Engineering and Software Development, SCITEPRESS, 2019, pp. 372–379.
- [49] Pekka Pääkkönen and Daniel Pakkala, *Reference architecture and classification of technologies, products and services for big data systems*, Big data research **2** (2015), no. 4, 166–186.
- [50] Bo Petersen, Henrik Bindner, Bjarne Poulsen, and Shi You, *Smart grid communication comparison: Distributed control middleware and serialization comparison for the internet of things*, 2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), 2017, pp. 1–6.

- [51] ZX Pi, XH Li, YM Ding, M Zhao, and ZX Liu, *Demand response scheduling algorithm of the economic energy consumption in buildings for considering comfortable working time and user target price*, Energy and Buildings **250** (2021), 111252.
- [52] Pika, *Pika*, <https://github.com/pika/pika>, 2013, (Accessed on 09/04/2021).
- [53] Pyca, *pyca/cryptography: cryptography is a package designed to expose cryptographic primitives and recipes to python developers.*, <https://github.com/pyca/cryptography>, 2021, (Accessed on 09/04/2021).
- [54] RabbitMQ, *Amqp 0-9-1 model explained*, <https://www.rabbitmq.com/tutorials/amqp-concepts.html>, 2021, (accessed: 05. 05.2021).
- [55] ———, *Quorum queues — rabbitmq*, <https://www.rabbitmq.com/quorum-queues.html>, 2021, (Accessed on 09/04/2021).
- [56] Redis, *Redis*, <https://redis.io/>, 2021, (Accessed on 09/16/2021).
- [57] Carlos Rodriguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Trabucco, Luigi Canali, and Gianraffaele Percannella, *Rest apis: A large-scale analysis of compliance with principles and best practices*, 06 2016, pp. 21–39.
- [58] Jesús Rodríguez-Molina and Daniel M. Kammen, *Middleware architectures for the smart grid: A survey on the state-of-the-art, taxonomy and main open issues*, IEEE Communications Surveys Tutorials **20** (2018), no. 4, 2992–3033.
- [59] Edmund O Schweitzer, David Whitehead, Ken Fodero, Paul Robertson, and D Gammel, *Merging sonet and ethernet communications for power system applications*, proceedings of the 38th Annual Western Protective Relay Conference, Spokane, WA, 2011.

- [60] Toshichika Shiobara, Peter Palensky, and Hiroaki Nishi, *Effective metering data aggregation for smart grid communication infrastructure*, IECON 2015-41st Annual Conference of the IEEE Industrial Electronics Society, IEEE, 2015, pp. 002136–002141.
- [61] R Shyam, Bharathi Ganesh HB, Sachin Kumar, Prabaharan Poornachandran, and KP Soman, *Apache spark a big data analytics platform for smart grid*, Procedia Technology **21** (2015), 171–178.
- [62] Lucija Šikić, Jasna Janković, Petar Afrić, Marin Šilić, Željko Ilić, Hrvoje Pandžić, Marijan Živić, and Matija Džanko, *A comparison of application layer communication protocols in iot-enabled smart grid*, 2020 International Symposium ELMAR, IEEE, 2020, pp. 83–86.
- [63] RO Sinnott, H Duan, and Y Sun, *Chapter 15-a case study in big data analytics: exploring twitter sentiment analysis and the weather*, Big Data (2016), 357–388.
- [64] Copernica Marketing Software, *Amqp-cpp*, <https://github.com/CopernicaMarketingSoftware/AMQP-CPP>, 2013, (accessed 18 09 2021).
- [65] Xinwei Sun, Ying Chen, Jiatai Liu, and Shaowei Huang, *A co-simulation platform for smart grid considering interaction between information and power systems*, ISGT 2014, IEEE, 2014, pp. 1–6.
- [66] Gustavo O Troiano, Hélder S Ferreira, Fernanda CL Trindade, and Luis F Ochoa, *Co-simulator of power and communication networks using openss and omnet++*, 2016 IEEE Innovative Smart Grid Technologies-Asia (ISGT-Asia), IEEE, 2016, pp. 1094–1099.
- [67] András Varga and Rudolf Hornig, *An overview of the omnet++ simulation environment*, Proceedings of the 1st international conference on Simulation tools

- and techniques for communications, networks and systems & workshops, 2008, pp. 1–10.
- [68] Mike Vogt, Frank Marten, and Martin Braun, *A survey and statistical analysis of smart grid co-simulations*, *Applied energy* **222** (2018), 67–78.
- [69] Yongxin Xie, Jianlin Liu, Taiyang Huang, Jianong Li, Jianlei Niu, Cheuk Ming Mak, and Tsz cheung Lee, *Outdoor thermal sensation and logistic regression analysis of comfort range of meteorological parameters in hong kong*, *Building and Environment* **155** (2019), 175–186.
- [70] Muneer Bani Yassein, Mohammed Q Shatnawi, et al., *Application layer protocols for the internet of things: A survey*, 2016 International Conference on Engineering & MIS (ICEMIS), IEEE, 2016, pp. 1–4.
- [71] Ercan Nurcan Yilmaz, Hüseyin Polat, Saadin Oyucu, Ahmet Aksoz, and Ali Saygin, *Data storage in smart grid systems*, 2018 6th International Istanbul Smart Grids and Cities Congress and Fair (ICSG), 2018, pp. 110–113.
- [72] Ziming Zhu, Sangarapillai Lambotharan, Woon Hau Chin, and Zhong Fan, *Overview of demand management in smart grid and enabling wireless communication technologies*, *IEEE Wireless Communications* **19** (2012), no. 3, 48–56.

Appendix A

Containerization Configuration and Deployment Details

This section contains the content of some of the docker files utilized in creating the images for some services described in the implementation in Chapter 4. It also contains the deployment details for running the implementation in kubernetes. The deployment details include storage configuration and definition and passing of environment variables to kubernetes pods.

A.1 Docker Files

Listing A.1: "Extended OmNeT++ Docker Image"

```
FROM omnetpp/omnetpp:u18.04-5.6.2 AS stage

ENV LIBUV_VERSION 1.41.0
ENV PROTOBUF_VERSION 3.15.8
ENV BUILD_PACKAGES autoconf automake curl libtool cmake g++ unzip
RUN apt-get update
RUN apt-get install -y $BUILD_PACKAGES

RUN curl -sSL https://dist.libuv.org/dist/v1.41.0/libuv-v${LIBUV_VERSION}.tar.gz | tar xzfv - -C
  /usr/local/src
WORKDIR /usr/local/src/libuv-v${LIBUV_VERSION}/
RUN sh autogen.sh \
  && ./configure \
```

```

    && make && make install \
    && rm -rf /usr/local/src/libuv-v${LIBUV_VERSION} \
    && ldconfig

# install protocol buffers
RUN curl -sSL https://github.com/protocolbuffers/protobuf/releases/download/v${PROTOBUF_VERSION}/
    protobuf-cpp-${PROTOBUF_VERSION}.tar.gz | tar zxfv -C /usr/local/src
WORKDIR /usr/local/src/protobuf-${PROTOBUF_VERSION}/
RUN ./configure \
    && make \
    && make install && ldconfig \
    && rm -rf /usr/local/src/protobuf-${PROTOBUF_VERSION}

#install openssl
RUN apt-get install -y libssl-dev

# compile rmq_lib
WORKDIR /thirdparty
COPY . /thirdparty/

WORKDIR /thirdparty/build
RUN cmake .. -DAMQP_CPP_LINUX_TCP=ON && cmake --build .
RUN cp -r /thirdparty/include/* /usr/local/include/ \
    && cp /thirdparty/build/src/librmq-example.a /usr/local/lib \
    && cp /thirdparty/build/thirdparty/AMQP-CPP/bin/libamqpcpp.a /usr/local/lib \
    && ldconfig

WORKDIR /
RUN apt-get remove --purge -y ${BUILD_PACKAGES} && rm -rf /var/lib/apt/lists/*

```

A.2 Deployment Files

Listing A.2: "Co-Simulation Deployment for Kubernetes"

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: smartgrid-cosimulation
spec:
  replicas: 1
  selector:
    matchLabels:
      app: smartgrid-instance
  template:
    metadata:
      labels:
        app: smartgrid-instance
    spec:
      imagePullSecrets:
        - name: regcred
      tolerations:

```

```

    - key: "node-role.kubernetes.io/master"
      operator: "Exists"
      effect: "NoSchedule"
  containers:
  - name: instance-gateway
    image: adeyemogab/gateway
    imagePullPolicy: IfNotPresent
    resources:
      limits:
        memory: "128Mi"
        cpu: "150m"
    envFrom:
      - configMapRef:
          name: smartgrid-kafka-config
      - configMapRef:
          name: smartgrid-redis-config
      - secretRef:
          name: smartgrid-keypassword-secret
      - secretRef:
          name: smartgrid-redis-secret
    env:
      - name: RABBITMQ_SERVER
        valueFrom:
          secretKeyRef:
            name: smartgrid-rabbit-secret
            key: rabbit
      - name: POD_ID
        valueFrom:
          fieldRef:
            fieldPath: metadata.name
      - name: GATEWAY_ID
        value: "$(POD_ID).gateway"
      - name: RABBITMQ_QUEUE_NAME
        value: $(POD_ID).gateway
      - name: RABBITMQ_EXCHANGE
        value: $(POD_ID)
      - name: RABBITMQ_EXCHANGE_TYPE
        value: direct
      - name: RABBITMQ_ROUTING_NAME
        value: $(POD_ID).gateway
      - name: RABBITMQ_COMM_ADDRESS
        value: $(POD_ID).comm
      - name: RABBITMQ_POWER_ADDRESS
        value: $(POD_ID).power
      - name: RABBITMQ_PUBLISH_INTERVAL
        value: "0"
      - name: KAFKA_TOPICS
        value: $(POD_ID)
      - name: KAFKA_GROUP_ID
        value: $(POD_ID)
  - name: instance-power
    image: adeyemogab/power
    imagePullPolicy: IfNotPresent
    volumeMounts:
      - name: smartgrid-logs
        mountPath: /logs/

```

```

env:
  - name: RABBITMQ_SERVER
    valueFrom:
      secretKeyRef:
        name: smartgrid-rabbit-secret
        key: rabbit
  - name: POD_ID
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
  - name: APPLICATION_ID
    value: "$(POD_ID).power"
  - name: RABBITMQ_QUEUE_NAME
    value: $(POD_ID).power
  - name: RABBITMQ_EXCHANGE
    value: $(POD_ID)
  - name: RABBITMQ_EXCHANGE_TYPE
    value: direct
  - name: RABBITMQ_ROUTING_NAME
    value: $(POD_ID).power
  - name: RABBITMQ_COMM_ADDRESS
    value: $(POD_ID).comm
  - name: RABBITMQ_GATEWAY_ADDRESS
    value: $(POD_ID).gateway
  - name: RABBITMQ_PUBLISH_INTERVAL
    value: "0"
  - name: POWER_MODEL_PATH
    value: /app/models/IEEE13Nodeckt.dss
resources:
  limits:
    memory: "1G"
    cpu: "500m"
- name: instance-comm
  image: adeyemogab/network
  imagePullPolicy: IfNotPresent
  volumeMounts:
    - name: simulation-config
      mountPath: /config/
    - name: smartgrid-logs
      mountPath: /logs/
env:
  - name: RABBITMQ_SERVER
    valueFrom:
      secretKeyRef:
        name: smartgrid-rabbit-secret
        key: rabbit
  - name: POD_ID
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
  - name: APP_NAME
    value: "$(POD_ID).comm"
  - name: RABBITMQ_QUEUE_NAME
    value: $(POD_ID).comm
  - name: RABBITMQ_EXCHANGE
    value: $(POD_ID)

```

```

    - name: RABBITMQ_EXCHANGE_TYPE
      value: direct
    - name: RABBITMQ_ROUTING_NAME
      value: $(POD.ID).comm
    - name: RABBITMQ_POWER_ADDRESS
      value: $(POD.ID).power
    - name: RABBITMQ_GATEWAY_ADDRESS
      value: $(POD.ID).gateway
  resources:
    limits:
      memory: "3G"
      cpu: "500m"
  volumes:
    - name: simulation-config
      configMap:
        name: smartgrid-omnetpp-config
    - name: smartgrid-logs
      persistentVolumeClaim:
        claimName: smartgrid-logs-pv-claim

```

Listing A.3: "Storage Definition for Kubernetes Deployment"

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: smartgrid-log-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: smartgrid-logs-pv-0
  labels:
    name: smartgrid-rabbit
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: smartgrid-log-storage
  local:
    path: /home/sujit/smartgrid/logs/
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - casa50
                - casa49

```

```
—
apiVersion: v1
kind: PersistentVolume
metadata:
  name: smartgrid-logs-pv-1
  labels:
    name: smartgrid-rabbit
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: smartgrid-log-storage
  local:
    path: /home/sujit/smartgrid/logs/
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
            - casa49
```

```
—
apiVersion: v1
kind: PersistentVolume
metadata:
  name: smartgrid-logs-pv-2
  labels:
    name: smartgrid-rabbit
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: smartgrid-log-storage
  local:
    path: /home/sujit/smartgrid/logs/
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
            - casa39
```

```
—
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

```
name: smartgrid-logs-pv-claim
spec:
  resources:
    requests:
      storage: 5Gi
  storageClassName: smartgrid-log-storage
  accessModes:
    - ReadWriteOnce
```

Appendix B

Simulation Configuration for Simple Application

This section contains the configuration details for running the simple application employed in Chapter 5 to evaluate the accuracy of the implemented framework. It also contains some configurations for the wide area monitoring network simulation in the same chapter.

B.1 Configuration for OmNeT++ for Simple Application

Listing B.1: "Base config"

```
network = xpo-one.simulations.XpoOne
cmdenv-express-mode = false
cmdenv-performance-display = false
cmdenv-event-banners = false
cmdenv-autoflush = true
cmdenv-redirect-output = false
**.cmdenv-log-level = info
*.manager.cmdenv-log-level = info
simtime-resolution=ms
sim-time-limit=60s
```

```
#Comment out for sequential scheduler
scheduler-class = "cRealTimeScheduler"
```

Listing B.2: "Setup script to read config from environment variables"

```
#!/bin/sh
# Get all the necessary environment variables and write to omnetpp file
cat /config/omnetpp.ini >> /conf/omnetpp.ini
echo "\n*.manager.appname=\"$APP_NAME\"\n" >> /conf/omnetpp.ini
echo "\n*.manager.queueName=\"$RABBITMQ_QUEUE_NAME\"\n" >> /conf/omnetpp.ini
echo "\n*.manager.url=\"$RABBITMQ_SERVER\"\n" >> /conf/omnetpp.ini
echo "\n*.manager.exchange=\"$RABBITMQ_EXCHANGE\"\n" >> /conf/omnetpp.ini
echo "\n*.manager.exchangetype=\"$RABBITMQ_EXCHANGE_TYPE\"\n" >> /conf/omnetpp.ini
echo "\n*.manager.routing=\"$RABBITMQ_ROUTING_NAME\"\n" >> /conf/omnetpp.ini
echo "\n*.manager.power.address=\"$RABBITMQ_POWER_ADDRESS\"\n" >> /conf/omnetpp.ini
echo "\n*.manager.gateway.address=\"$RABBITMQ_GATEWAY_ADDRESS\"\n" >> /conf/omnetpp.ini
echo "\nncmdenv-output-file=\"/logs/$APP_NAME-log.out\"\n" >> /conf/omnetpp.ini
```

B.2 OpenDSS Modifications to IEEE 13-bus for Simple Application

```
!LoadShapes
New Loadshape.Residential npts=24 interval=1 mult=(0.69000000 0.50999999 0.44999999 0.41999999
0.55000001 0.85000002 1.01999998 0.80000001 0.89999998 0.91000003 1.02999997 1.03999996
1.11000001 0.98000002 0.94000000 0.94000000 1.02999997 1.26999998 1.51999998 1.59000003
1.75999999 1.50999999 1.29999995 0.89999998) ! c?digo=Residential 101-220 kWh
New Loadshape.Residential_Two npts=24 interval=1 mult=(0.69000000 0.50999999 0.44999999
0.41999999 0.55000001 0.85000002 1.01999998 0.80000001 0.89999998 0.91000003 1.02999997
1.03999996 1.31000001 0.98000002 0.74000000 0.84000000 0.92999997 1.06999998 1.21999998
1.39000003 2.15999999 1.20999999 1.09999995 0.69999998) ! c?digo=Residential 101-220 kWh

New Monitor.M1 element=line.684611 terminal=1
New Monitor.M2 element=load.611 terminal=1
New Monitor.M3 element=load.652 terminal=1
```

B.3 Configuration for OmNeT++ for Wide Area Network

Listing B.3: "Base Configuration for Wide Area Network"

```
network = xpo_one.simulations.XpoOne
cmdenv-express-mode = false
cmdenv-performance-display = false
cmdenv-event-banners = false
cmdenv-autoflush = true
cmdenv-redirect-output = true
**.cmdenv-log-level = info
**.pdc[*].pmu[*].sample_per_second = 1
simtime-resolution=ms
sim-time-limit=60s

XpoOne.pdc[0].pmu[0].bus = "675"
XpoOne.pdc[0].pmu[1].bus = "650"
XpoOne.pdc[0].pmu[2].bus = "671"
XpoOne.pdc[1].pmu[0].bus = "692"
XpoOne.pdc[1].pmu[1].bus = "632"
XpoOne.pdc[1].pmu[2].bus = "684"
XpoOne.pdc[2].pmu[0].bus = "633"
XpoOne.pdc[2].pmu[1].bus = "634"
XpoOne.pdc[2].pmu[2].bus = "611"
XpoOne.pdc[3].pmu[0].bus = "670"
XpoOne.pdc[3].pmu[1].bus = "680"
XpoOne.pdc[3].pmu[2].bus = "645"
```

Appendix C

Configuration for Demand-Response Application

This section presents the configuration of, and code snippets from, the demand-response application presented in Chapter 5.

C.1 Modifications to IEEE 13-bus for Demand Response Application

Listing C.1: Message Structure in Proto

```
New LoadShape.SolarRamp npts=300 sinterval=1 mult=(file=solarshape.csv)

! Generator
New Generator.SolarGen Phases=2 Bus1=684.1.3 kV=4.16 kW=1000 PF=1 Daily=SolarRamp

New storage.2MWStorage Phases=2 Bus1=684.1.3 kV=4.16 kwrated=500 PF=1 kWhStored=50
~ model=1 dischargeTrigger =0.0 chargeTrigger = 0.0 state=IDLING

! Monitor Definitions
New Monitor.load_M4 element=load.611 terminal=1

New Generator.ConGen Phases=2 Bus1=684.1.3 kV=4.16 kW=500 PF=1 Model=1 Enabled=No
```

C.2 Logistic Regression Dataset Generation

Listing C.2: Data Generation

```
import random
import pprint
results = []
# Repeated for different ranges
# 2450 - 2500
# 2350 - 2400
# 2300 - 2350
for item in range(50):
    val = random.randrange(2450, 2500)
    if val <= 2350:
        results.append((val, 1))
    elif val >= 2450:
        results.append((val, 2))
    else:
        results.append((val, 0))
```

Vita

Candidate's full name: Gabriel Olawale Adeyemo

University attended: Bachelor of Science, Landmark University, 2018

Publications: None

Conference Presentations: None