A Synchrophasor Stream Processing Pipeline Architecture for Near-Real-Time Applications

by

Daniel Villegas Posada

A thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg.

Copyright © 2025 by Daniel Villegas Posada

Abstract

Since their introduction, synchrophasor measurement networks have been steadily growing and are expected to continue growing as distribution Phasor Measurement Units (PMU) become mainstream. Moreover, recent advances in data-driven methods, in contrast to traditional synchrophasor applications, introduce more demanding data requirements. Together, these factors create the need to research data engineering methods for handling synchrophasor data at a large scale. This research investigates the use of a stream processing framework to build synchrophasor applications and proposes a novel reconfigurable synchrophasor data pipeline for near-real-time applications. The processing pipeline is part of a larger system which comprises data collection at the substation level, data ingestion, processing and a centralized metadata registry. In addition, to overcome some of the technical, cybersecurity and economic challenges of traditional PMU networks, the system is developed using open-source, off-the-shelf technologies and is deployed to a cloud provider. Lastly, the system is tested using a real-time simulation, the data is collected by multiple agents and forwarded to the cloud using different data rates to assess its performance in terms of the latency.

Acknowledgment

I would like to express my sincere and deep gratitude to the individuals that have accompanied me in this stage of growth and learning as well as to the institution that made this possible. I am grateful to the Professor Athula Rajapakse for his support, guidance and for welcoming me in the Intelligent Power Grid Laboratory (iPGL). To the Professors who kindly thought the courses during the program. To my dear wife Valentina who helped me finding strength, discipline and motivation. To my colleagues and friends. And to the Faculty of Graduate Studies and the University of Manitoba. I am fortunate and thankful for this invaluable experience.

I would like to further extend this acknowledgment to the review committee for taking the time to review this thesis and whose valuable questions and observations enriched the current work and provided perspectives that helped me forming a better understanding of future directions.

Dedication

To my family.

Contents

Abstract .... *ii*

Acknowledgment .... *iii*

Dedication .... *iv*

Contents .... *v*

List of Tables .... *viii*

List of Figures .... *ix*

List of Acronyms and Abbreviations .... *xiii*

Chapter 1 Introduction *15*

1.1 Background 17

1.2 Problem Statement 19

1.3 Research Motivation 21

1.4 Research Objectives 21

1.5 Thesis Organization 22

Chapter 2 Literature Review *24*

2.1 Synchrophasor Technology Motivation and History 24

2.2 Evolution of Synchrophasor Standards and Protocols 25

2.3 Introduction to Data Engineering 33

2.4 Synchrophasor Big Data Platforms 42

Chapter 3 Cloud Platform Overview *49*

3.1 Overall Structure of the Measurement System 49

3.2 Testing and Validation Setup 52

3.3 Summary 54

Chapter 4 Substation Agent *55*

4.1 Design and Architecture 56

4.2 Capabilities and Usage 73

4.3 Testing and Validation 77

4.4 Summary 83

Chapter 5 Cloud Infrastructure *84*

5.1 Challenges integrating synchrophasor protocols with the cloud 84

5.2 Cloud Infrastructure Resources 85

5.3 Cloud Application Design 86

5.4 Database Schema 87

5.5 Summary 88

Chapter 6 Data Pipeline Design *90*

6.1 Exploration and Design 91

6.2 Data Model 95

6.3 Synchrophasor Data Pipeline 97

6.4 Latency Measurement 102

6.5 Scalability, Fault Tolerance and Testing 109

6.6 Summary 109

Chapter 7 Conclusions and Future Work 111

7.1 Conclusions 111

7.2 Future work 114

References 116

List of Tables

Table 1 Classification of Synchrophasor Network Objectives by Timeframe [5]........................ 18

Table 2 IEEE 1344-1995 Synchronization Requirements [19] 26

Table 3 IEEE Std 1344-1995 Communication Frames [19] 26

Table 4 IEEE Std C37.118™-2005 Communication Frames IEEE Std C37.118™-2005 [20] ... 29

Table 5 IEEE Std C37.118™-2011 Configuration Frame 3 [4] 31

Table 6 GEP Message format [22]... 32

Table 7 IoTDB Design Constraints... 48

Table 8 Command Line Interface Tool Commands to Connect to a PMU 75

Table 9 Substation Agent Validation Measurements... 80

Table 10 Provisioned Cloud Resources 85

List of Figures

Figure 1 Hierarchical Synchrophasor Network ... 19

Figure 2 TVE Illustration ... 28

Figure 3 Dataflow Model Example ... 38

Figure 4 Stream Processing Primitive Operators [41] ... 40

Figure 5 Tumbling Window ... 41

Figure 6 Session Window ... 41

Figure 7 Sliding Window ... 42

Figure 8 FNET/GridEye Analytics Architecture ... 44

Figure 9 k-ary tree Illustration ... 45

Figure 10 System Overview. a) substation b) distributed message broker c) stream processing application ... 52

Figure 11 Test Power System ... 53

Figure 12 Simulation Setup ... 54

Figure 13 Substation Agent Layers ... 57

Figure 14 IEEE Std C37.118™-2005 Client and Frame Synchronization ... 59

Figure 15 Protobuf Message Models ... 60

Figure 16 Protocol Translation and Batching ... 61

Figure 17 Library Enumerations 62

Figure 18 Abstract Base Frame and Unbound Frame 63

Figure 19 Header Frame Class Diagram 64

Figure 20 Command Frame Class Diagram 65

Figure 21 Configuration Frame Class Diagram 66

Figure 22 Configuration Frame Object Diagram 67

Figure 23 Data Frame Class Diagram 68

Figure 24 Data Frame Object Diagram 69

Figure 25 Serialization and Deserialization Class Diagrams 70

Figure 26 Struct Module Illustration 70

Figure 27 Command and Header Frame Serialization and Deserialization Format Strings 71

Figure 28 Configuration Frame Serialization and Deserialization Format Strings 72

Figure 29 Data Frame Serialization and Deserialization Format Strings 73

Figure 30 YAML configuration fragment 74

Figure 31 Abstract Substation Agent Event Handlers 76

Figure 32 Configuration frame update 76

Figure 33 Unit Tests Coverage 78

Figure 34 Substation Agent Testing and Validation Output 79

Figure 35 Percentage of Error in the Time Between Frames and the Expected Period 81

Figure 36 Accumulated Average Time Difference 82

Figure 37 Cloud Application Architecture 87

Figure 38 Database Schema 88

Figure 39 Simple Processing Job 92

Figure 40 Simple Processing Job Snippet and Execution Plan 93

Figure 41 Simple Processing Job with Operation Stream 94

Figure 42 Simple Processing Job with Operation Stream Snippet and Execution Plan 95

Figure 43 Pipeline Data Model 96

Figure 44 Output Data Model 97

Figure 45 Synchrophasor processing pipeline data flow diagram 99

Figure 46 Data transformations in the pipeline 100

Figure 47 Example Implementation of the Phase Difference Between Synchrophasors 102

Figure 48 System Latency 103

Figure 49 Latency measurement path 104

Figure 50 Arrival Latency Results 105

Figure 51 Data Ready for Processing Latency 106

Figure 52 Latency variation in time 108

Figure 53 Latency Deviation From the Mean 108

List of Acronyms and Abbreviations

API	Application Programming Interface
CD	Continuous Delivery
CI	Continuous Integration
FDR	Fault Detection and Recovery
GEP	Generic Event Protocol
GFS	Google™ File System
GPS	Global Positioning System
HDD	Hard Disk Drive
IEEE	Institute of Electrical and Electronics Engineers
MLPS	Multi-Protocol Label Switching
NASPI	North American SynchroPhasor Initiative
PDC	Phasor Data Concentrator
PMU	Phasor Measurement Unit
PPS	Pulse Per Second
RDD	Resilient Distributed Dataset
RMS	Root Mean Square
SCADA	Supervisory Control and Data Acquisition
SQL	Structured Query Language
STTP	Streaming Telemetry Transport Protocol
TCP	Transmission Control Protocol
TSSC	Time Synchronization Status Code
TVE	Total Vector Error

UDP	User Datagram Protocol
UML	Unified Modeling Language
UTC	Universal Coordinated Time
YAML	Yet Another Markup Language

Chapter 1 Introduction

Power systems are one of the key drivers of industrialization, and their continuous operation is paramount to sustaining modern societies and living standards. The vast majority of modern technology is powered by electricity supplied through power systems making all essential infrastructure such as water supply, hospitals, telecommunications, airports, emergency services, etc., susceptible to failures in the continuous supply of energy. The perception of ordinary public of the power systems is generally limited to net power production and they tend to overlook the systems and effort required for maintaining a reliable power supply. In reality, reliability is potentially one of the most complex and expensive aspects of power systems and power systems engineering.

Electric grids interconnect many loads and generators, forming an extensive, complex, dynamic system that must adapt to changes in load, generation, topology, and disturbances to remain reliable and stable. Power system operators are responsible for taking action to keep the system operating in this manner, and knowledge of the current state of the electric grid is indispensable for taking the correct decisions.

Even from the beginning of power systems, there has always been some level of supervision and monitoring. Over the years, this has evolved from local electromechanical sensors in generating stations and the most important substations to far-reaching digital sensor networks, bringing with it a plethora of possibilities for system operators as well as technical challenges. The advent of digital communications made it possible to expand measurement networks well beyond the local measurements and gave system operators a global view of the power system. This evolution

brought Supervisory Control and Data Acquisition systems; which are better known as SCADA and have been used for many years. While the definition of SCADA systems is broad and makes no assumptions on how measurements are taken or the sampling rates, they are often associated with slow sampling rates and low-resolution root mean squared (RMS) measurements of voltage, current, active power, reactive power and frequency over time windows. The low time resolution and nature of the measurements make it impossible to observe fast dynamic phenomena and require non-linear computations to understand the operating state of the system [1].

Synchronized phasor measurement, first introduced in the 1990s [2], is a new type of measuring technology that tries to overcome some of the limitations of SCADA systems by providing higher-time resolution phasor measurements. Phasor measurements are advantageous over RMS measurements not only because of their high time resolution (30 or 60 times per second or more) but also because they provide meaningful phase angle measurements allowing direct application of many power system theories formulated in terms of phasor quantities. One of the scientific and engineering advancements that made phasor measurement possible was the introduction of the GPS system [2], which made it possible to have widespread and accurate clock synchronization, hence synchronized phasor, which is vital in determining the angle of phasor measurements. Synchronized phasors are then time-tagged measurements of power system complex quantities, mainly voltages and currents, that present an accurate snapshot of the power system operating state at any given point in time.

Synchrophasor networks have gained popularity over the last twenty years, offering power system operators new opportunities to better understand the power system. This popularity is expected to continue growing as substations and power system assets are renewed, their monitoring equipment is retrofitted, and the technology expands from transmission to distribution systems [3]. Evidently,

one of the most difficult aspects of adopting synchrophasor technologies is and will continue to be data management and data engineering. This is manifested in the evolution of the standards, which adapt to emerging issues, and also the incentives and efforts made by governments, manufacturers and utility companies to advance the adoption of synchrophasor technology. One of the most relevant joint efforts in this direction is made by the North American Synchrophasor Initiative (NASPI), which emphasizes the importance of advanced synchrophasor architectures as a key research stream.

This research investigates some of the data engineering aspects of managing real-time data for synchrophasors applications in distributed computing systems and aims to shed some light on the future of synchrophasor technologies as the data volumes start surpassing the capabilities of existing systems.

1.1 Background

The IEEE Std C37.118.2™-2011 [4] standard formalized the idea of a hierarchical synchrophasor network to transport synchrophasor measurements from the measuring nodes to centralized locations where the applications reside. The NASPInet 2.0 Architecture Guidance [5] defines four clear objectives for synchrophasor networks: observability, advanced grid capabilities, improved protection functions, and cybersecurity. Moreover, architectural guidelines are specified to ensure the network meets these objectives.

Table 1 classifies applications by the timeframe in which each is expected to be executed as per the NASPInet 2.0 Architecture Guidance [5]. Real-time refers to applications that require low and deterministic latency; near-real-time refers to low but non-deterministic latency, i.e., can tolerate

eventual communication delays or event-driven behaviour; and offline applications that are executed over stored data that are not constrained to a specific timeframe.

<table>
<tr><th></th><th>Real-time (<100ms)</th><th>Near-real-time (<1min)</th><th>Offline</th></tr>
<tr><td rowspan="3">Observability</td><td></td><td>Operator decision support</td><td></td></tr>
<tr><td></td><td></td><td>Post-event analysis</td></tr>
<tr><td></td><td></td><td>Planning</td></tr>
<tr><td rowspan="2">Advanced grid capabilities</td><td colspan="3">Management and integration of stochastic grid resources</td></tr>
<tr><td colspan="2">New forms of grid control and stabilization</td><td></td></tr>
<tr><td>Improved protection</td><td>Flexible protection schemes</td><td></td><td></td></tr>
<tr><td></td><td colspan="2">Adaptive protection schemes</td><td></td></tr>
<tr><td rowspan="2">Cyber-physical security</td><td></td><td>Cyber-physical security</td><td></td></tr>
<tr><td></td><td></td><td>Forensic analysis</td></tr>
</table>

Table 1 Classification of Synchrophasor Network Objectives by Timeframe [5]

Traditionally, synchrophasor hierarchical networks were thought of as multiple layers of Phasor Data Concentrators (PDC) stacked on top of each other to perform various aggregation levels until data arrived at the final destination in what had been thought of as a super PDC; this is shown in Figure 1. NASPInet 2.0 advises against PDC stacking and redefines PDC as a function and not a specific device; this adds some flexibility to the definition as it is not prescriptive in the network structure but only in its functional aspects. One of the proposed architectures consists of locating PDC functions on the edge (substations); however, this approach may have horizontal scalability and security issues.

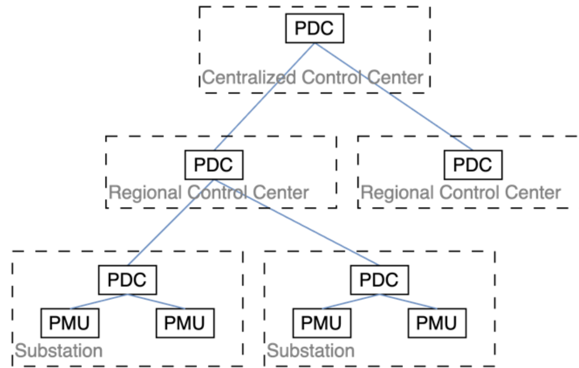


Figure 1 Hierarchical Synchrophasor Network

To comply with the strict real-time communication application requirements, synchrophasor networks often need dedicated and expensive network infrastructure. Moreover, some authors have found current synchrophasor technologies to pose a challenge for modern data analytic applications [6]. Lastly, security, cost, and bandwidth availability may become limiting factors in the types of applications that can be deployed, closing the door on non-mission-critical applications that need near-real-time data access.

1.2 Problem Statement

Synchrophasor networks are designed to meet strict requirements imposed by real-time mission-critical applications. They are expensive to build because they need dedicated infrastructure and are challenging to operate because they have special security considerations. Moreover, PDCs are usually single-node systems, which limits the number of Phasor Measurement Unit (PMU) streams that can be ingested and the number of applications that can consume synchrophasor data, i.e. they have limited horizontal scalability. This has an important implication, as non-critical applications may not be viable because they compete with critical applications for resources. On the other hand, offline applications often operate on the data collected from channels used for real-time

applications; this means that they operate on the data that is available and may not have access to measurements that are, in principle, not critical for maintaining the power system's integrity.

Real-time applications are designed with supporting real-time operation as one of their main objectives. Data analytics applications often have a different set of requirements, one being that they benefit from having more data sources. This is not necessarily true for conventional synchrophasor applications, as they do not leverage the stochastic nature of data. Current synchrophasor networks may fall short in front of the growing number of sensors [7] and emerging techniques that can benefit from more data [8-10]. Likewise, they may exhibit limited capabilities to meet the data exchange objectives of NASPInet 2.0, which seeks interoperability [5]. Moreover, synchrophasor networks often rely on proprietary hardware that makes integration with existing tools, applications and data sources difficult. This has been identified as hindering the development of data-driven applications for the grid [6].

Lastly, the growing number of measurement devices will force PDCs to evolve from single-node devices providing PDC functionality to distributed systems capable of handling an arbitrary number of measurements. This issue has not been sufficiently explored and only a couple of solutions that provide horizontal scalability to synchrophasor systems [11-13] have been reported in the literature.

This research investigates the technical aspects of creating a cloud-based synchrophasor processing data pipeline to enable near-real-time applications. The system is built using open-source and off-the-shelf technologies, and the objective is to assess the readiness of available software for this application. In addition, synchrophasor measurements are sent to a partitioned system with the aim of overcoming the current scalability limitations.

1.3 Research Motivation

The data engineering ecosystem contains a broad set of mature, well-developed, and well-supported technologies that have proven to be successful across many industries, particularly in IoT applications. The big-data ecosystem is not only fueled by communities of contributors but also supported by large companies that rely on it and support its development.

The synchrophasor data engineering ecosystem has not evolved parallel to data engineering in other industries. This can be mainly attributed to the unique synchrophasor characteristics that make the adoption of generic technologies challenging [14]. Furthermore, often proprietary and closed-source nature of synchrophasor technologies has contributed to this.

This raises two questions about the readiness and adequacy of big-data frameworks for the synchrophasor use case. The literature has not widely addressed these questions, and only a few cases have attempted to use big-data frameworks for synchrophasor data processing. Furthermore, many synchrophasor data applications opt for standalone implementations, making the use of stream processing frameworks for synchrophasor data not a widely explored topic.

1.4 Research Objectives

The main objective is to develop a cloud-based synchrophasor data platform capable of ingesting data from multiple substations and to implement a generic reconfigurable stream processing pipeline for implementing near-real-time synchrophasor applications. The following secondary objectives derive from the primary purpose:

1. To review the evolution of synchrophasor data platforms from its beginning to the most recent implementations as well as synchrophasor data applications to understand their performance requirements.
2. To develop an IEEE Std C37.118™-2005 compatible tool to collect synchrophasor data from measurement units and send it to a cloud computing system.
3. To implement a cloud system capable of ingesting synchrophasor data for near-real-time applications using off-the-shelf technologies.
4. To evaluate the adequacy of the synchrophasor processing pipeline from the point of view of the synchrophasor data application requirements and the existing technology.

1.5 Thesis Organization

In Chapter 2, the literary review begins with a brief recount of synchrophasor technology and its evolution, as well as the evolution of synchrophasor protocols with the objective of trying to understand the direction of the technology. Then, it follows with a brief introduction to data engineering and the principles of stream processing. Lastly, it finishes with a recount of existing synchrophasor big-data platforms and synchrophasor data analytic applications.

Chapter 3 presents an overview of the system design and the simulation model that will be used to validate the data system. This section begins with the implementation of a real-time simulation and an overview of the setup to capture and forward the synchrophasor streams to the cloud.

Chapter 4 contains a detailed description of the synchrophasor agent's design and implementation, as well as the methodology used for testing, validation, and results.

Chapter 5 presents the design and implementation details of the cloud infrastructure used to host the data pipeline.

Chapter 6 provides a detailed description of the design of the processing pipeline using Apache Flink. This chapter begins with the design goals, a step-by-step description of the implemented pipeline, validation and performance testing methodology, and lastly, it presents the testing results.

Chapter 7 presents a discussion of the findings, conclusions and future work.

Chapter 2 Literature Review

2.1 Synchrophasor Technology Motivation and History

As power systems grew, the need for wide-area monitoring became evident. The history of synchrophasor technologies and wide-area measurements is described in detail by Prof. Arun G. Phadke, a prominent researcher and proponent of the symmetrical component distance relay [15-16]. As a result of the Northeast blackout on November 9 and 10, 1965, the investigating commission made multiple recommendations to improve power systems stability in future events, the periodic re-evaluation of dynamic performance of power grids, periodic adjustment of relay settings and assessment of automation technologies during emergency conditions [17]; these opened the door to the development of modern wide-area measurement systems.

Later, the symmetrical component measurement algorithm for distance relaying was introduced in [16]. This algorithm allows measuring the positive sequence component of a three-phase signal sampling data from a single cycle, opening the door to other applications such as higher time resolution measurements [18]. The development of Global Positioning System (GPS) technology helped overcome the technical challenges of clock synchronization, and the first PMU was introduced in 1988 [15].

Before the introduction of synchrophasors, power system operators relied on non-linear state estimation to determine the current state of the power system. Synchrophasors can improve non-linear state estimation by directly measuring the system's state. Furthermore, synchrophasors can provide a higher-resolution look into power system dynamics than traditional SCADA technologies as they have higher data rates. Other early applications for synchrophasor

technologies were adaptive out-of-step relaying, voltage and rotor angle stability assessment, and improved power system visualizations for system operators [2].

2.2 Evolution of Synchrophasor Standards and Protocols

This section presents the synchrophasor standards in chronological order to understand how they have evolved and what problems they aimed to solve.

2.2.1 IEEE Std 1344 (1995) [19]

IEEE Std 1344, published in 1995, was the first synchrophasor standard. This standard defines a communication protocol to enable communication with PMU and the synchronization requirements.

Two processes are involved in the time synchronization of PMU measurements: the distribution of the synchronizing signal and the mechanism for synchronizing each measurement to the internal PMU clock. First, the standard allows the synchronizing signal to be distributed from a central location using microwave links, fibre optic links, AM radio, or GPS satellites as long as some minimum synchronization requirements are met. Second, the measurement process consists of taking samples at a fixed rate; each measurement is indexed with a number from 0 to the sampling rate minus one. The measurement trigger signal must be phase-locked to the synchronizing signal to a maximum sample point error, and the measurement with the index 0 must occur at the beginning of the second-of-century within a maximum allowable error. Synchronization requirements are summarized in Table 2. A detailed description of what a synchrophasor is not provided; the signal $v(t) = \sqrt{2}V\cos(\omega_0 t + \varphi)$ is used as a convention.

Parameter	Value
Repetition rate	1 PPS
Minimum stability	0.1µs
Minimum availability/reliability	99.87%
Maximum synchronization error	1µs
Sample point error	1%
Second of century rising edge error	±100ns

Table 2 IEEE 1344-1995 Synchronization Requirements [19]

The standard defines an application layer protocol consisting of 4 types of frames or messages used to transmit phasor measurements. The frame structure is defined in Table 3.

Header frame		
Field	**Size (B)**	**Description**
SOC	4	Second-of-century
SMPCNT	2	Sample number / frame count
STAT	2	Status word
DATA	n	Payload
CRC_{16}	2	Checksum

Received messages		
Field	**Size (B)**	**Description**
SOC	4	Second-of-century
IDCODE	8	PMU Id
CMD	2	Command
CRC_{16}	2	Checksum

Data frame		
Field	**Size (B)**	**Description**
SOC	4	Second-of-century
SMPCNT	2	Sample number / frame count
STAT	2	Status word
PHASOR	n x 4	Phasor channels
FREQ	2	Frequency
DFREQ	2	Rate of change of frequency
DIG	n x 2	Digital channels
CRC_{16}	2	Checksum

Configuration frame		
Field	**Size (B)**	**Description**
SOC	4	Second-of-century
SMPCNT	2	Sample number / frame count
STAT	2	Status word
STN	16	Station name
IDCODE	8	PMU Id
PHNMR	2	Number of phasors
DGNMR	2	Number of digital channels
CHNAM	n x 16	Phasor channel names
PHUNIT	n x 4	Phasor channel conversion factor
DIGUNIT	n x 2	
FNOM	2	Nominal frequency
PERIOD	2	Sampling period
CRC_{16}	2	Checksum

Table 3 IEEE Std 1344-1995 Communication Frames [19]

2.2.2 IEEE Std C37.118™-2005 [20]

IEEE Std C37.118™-2005 [20] provides a more detailed definition of a phasor, introduces an accuracy measure for synchrophasors, and improves the communication protocol for correlating phasors from multiple sources.

A reference cosine function at nominal frequency synchronized to the Universal Coordinated Time (UTC) is defined as a frame of reference to measure the synchrophasor phase of synchrophasors. A synchrophasor of a signal $x(t) = X_m \cos(\omega t + \varphi)$ is then defined as $X = \left(\frac{X_m}{\sqrt{2}}\right) e^{j\varphi}$, where φ is measured relative to the common frame of reference.

IEEE Std 1344 did not impose any constraints on the accuracy of the (TVE), defined in (1).

$$TVE = \sqrt{\frac{(X_r(n) - X_r)^2 + (X_i(n) - X_i)^2}{X_r^2 + X_i^2}} \quad (1)$$

Where X_r and X_i are the true real and imaginary components of a signal and, $X_r(n)$ and $X_i(n)$ are the real and imaginary components of the signal in steady-state. TVE is used to define compliance limits. The TVE concept is illustrated in Figure 2, the phasor *n* is compliant as it lays inside the circumference defined relative to the theoretical value of the measured signal. On the other hand, phasor *k* is not compliant because it lays outside of the circle.

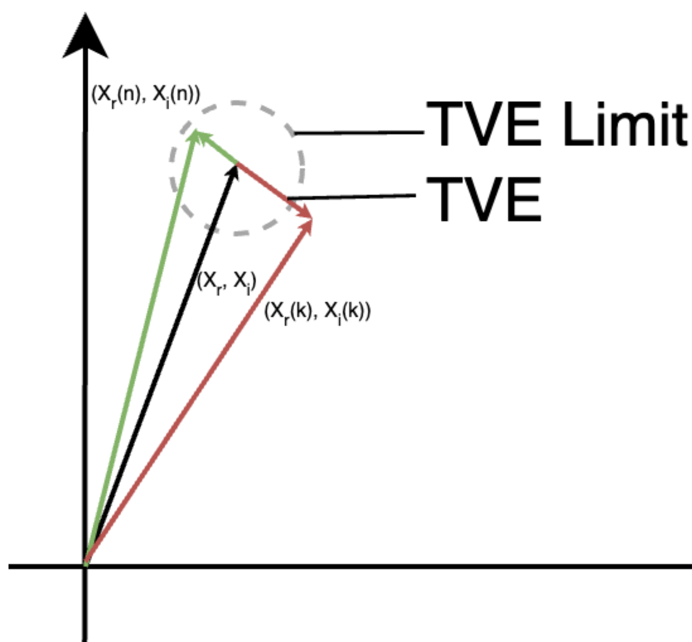


Figure 2 TVE Illustration

The message format is improved by introducing a common frame structure shared by all frame types; this structure wraps a binary payload dependent on the frame type and allows data frames to carry data from multiple PMUs. The latter makes way for a new topology where numerous streams can be concentrated on a single device, while IEEE Std 1344 was better suited for point-to-point communication. In addition, it defines two types of configuration frames, CFG-1 and CFG-2; the former denotes PMU capability, while the latter defines the schema of the currently transmitted stream. The IEEE Std C37.118™-2005 communication frame formats are shown in Table 4.

Common Frame		
Field	**Size (B)**	**Description**
SYNC	2	Synchronization word and frame type
FRAMESIZE	2	Number of bytes in the frame
IDCODE	2	PMU Id
SOC	4	Second-of-century
FRACSEC	4	Fraction of second and time quality
DATA	n	Payload
CRC_{16}	2	Checksum

Header frame payload		
Field	**Size (B)**	**Description**
DATA	n	Arbitrary ASCII characters

Command frame payload		
Field	**Size (B)**	**Description**
CMD	2	Command
EXTFRAME	n	User defined extended frame

Configuration frame payload		
Field	**Size (B)**	**Description**
TIME_BASE	4	FRACSEC divisor to compute timestamp fractional part
NUM_PMU	2	Number of PMUs
Repeat NUM_PMU times		
STN	16	Station name
IDCODE	2	PMU Id
FORMAT	2	Numeric format
PHNMR	2	Number of phasors
ANNMR	2	Number of analog measurements
DGNMR	2	Number of digital words
CHNAM	16 x (PHNMR + ANNMR + 16 x DGNMR)	Channel names
PHUNIT	4 x PHNMR	Conversion factor
ANUNIT	4 xANNMR	Conversion factor
DIGUNIT	4 x DGNMR	Normal status and mask
FNOM	2	Nominal frequency flag
CFGCNT	2	Configuration count / version
DATA_RATE	2	Data rate

Data frame payload		
Field	**Size (B)**	**Description**
STAT	2	Flags
Repeat NUM_PMU times		
PHASORS	4 x PHNMR 8 x PHNMR	Phasor measurements
FREQ	2/4	Frequency
DFREQ	2/4	Rate of change of frequency
ANALOG	2 x ANNMR 4 x ANNMR	Analog measurements
DIGITAL	2 x DGNMR	Digital status words

Table 4 IEEE Std C37.118™-2005 Communication Frames IEEE Std C37.118™-2005 [20]

2.2.3 IEEE Std C37.118.1™-2011 [21] and IEEE Std C37.118.2™-2011 [4]

In the 2011 revision, the standard is divided into two parts: IEEE Std C37.118.1™-2011 concerns the metrological aspects of the standard; most notably, it defines two performance classes, P and M. The P class is intended for fast applications and defines some response requirements. The standard does not impose specific requirements on the type of filter to be used, the manufacturer is free to implement filtering to meet the accuracy requirements. In contrast, the M class is designed for applications that require a higher accuracy but can tolerate filtering delays. The M performance class sets minimum filtering requirements for its type.

IEEE Std C37.118.2™-2011, on the other hand, concerns communications and data architecture. From the point of view of data architecture, IEEE Std C37.118.2™-2011 introduces two main changes to the standard. First, it formalizes the concept of a synchrophasor network as a hierarchically organized network made of multiple PMUs whose data is aggregated in local PDCs and forwarded to other PDCs to be used by various applications. Second, it introduces a third configuration frame (Table 5) type that contains additional metadata while keeping the whole protocol backward compatible.

Configuration frame 3 payload		
Field	**Size (B)**	**Description**
CONT_IDX	2	Continuation index for fragmented configuration frames
TIME_BASE	4	FRACSEC divisor to compute timestamp fractional part
NUM_PMU	2	The number of PMUs in the data frame
STN	1-256	The station name
IDCODE	2	PMU ID
G_PMU_ID	16	18 bit Global Unique PMU ID
FORMAT	2	Numeric format
PHNMR	2	Number of phasors
ANNMR	2	Number of analog channels
DGNMR	2	Number of digital channels
CHNAM	1-256	Channel names
PHSCALE	12	Phasor scaling constant
ANSCALE	8	Analog scaling constant
DIGUNIT	4	Normal status and mask
PMU_LAT	4	Latitude
PMU_LON	4	Longitude
PMU_ELEV	4	Elevation
SVC_CLASS	1	Service class M or P
WINDOW	4	Measurement window length
GRP_DLY	4	The estimated delay of the measurement based on existing filters and processing.
FNOM	2	Nominal frequency
DATA_RATE	2	Data rate
CFGCNT	2	Configuration frame version count

Table 5 IEEE Std C37.118™-2011 Configuration Frame 3 [4]

Configuration frame 3 introduces metadata which were not present in previous protocols, such as geolocation and information on the specific PMU hardware. It can also define a global PMU ID, which is an index that uniquely identifies each PMU. The continuation index allows for large-size configuration frame fragmentation. Still, it does not address the issue of uncontrolled network fragmentation of large data frames, a phenomenon that can lead to lateness and data loss.

2.2.4 GEP Gateway Exchange Protocol and STTP [22]

One of the most critical limitations of the IEEE Std C37.118™-2005 and 2011 protocol series is that it is an application-layer protocol that only defines the data structure used to transmit synchrophasor data and does not deal with how data is transmitted or any other low-level networking mechanisms. This makes IEEE Std C37.118™-2005 and 2011 protocols unable to control packet fragmentation, which leads to data loss when transmitted using UDP or lateness due to retransmission when transmitted using TCP.

GEP is a transport-layer, open protocol designed to transmit large volumes of data between utilities. It follows a publish/subscribe architecture pattern, which means that producers publish messages to an intermediate layer where subscribers can consume messages. Consumers, on the other hand, subscribe to this intermediate layer, and messages matching specific criteria are forwarded to them without the need for them to poll them. Unlike IEEE Std C37.118™-2005 and 2011, which are frame-based protocol, GEP is measurement-based, which means that each message carries one measurement rather than a group of structured measurements. The message format is presented in Table 6. This allows the protocol to transmit data without fragmentation (or rather, by controlling fragmentation), which reduces latency and data loss [23]. Under GEP, two channels are established: a TCP channel for commands between publisher and subscribers and a UDP channel for data transmission.

Subscriber Command Format		
Field	**Size (B)**	**Description**
Payload Marker	4	A fixed hex string to mark the beginning of the message
Payload Length	4	Length of the message
Subscriber command code	1	Subscriber command code

Publisher Response Format		
Field	**Size (B)**	**Description**
Response Code	1	Response code
Command code	1	Command code in the subscriber request
Response length	4	Length of the message
Response payload	n	

Data Package Format		
Field	**Size (B)**	**Description**
Point ID	16	A 128 bits UUID that identifies a measurement channel
Timestamp	8	64 bit timestamp measured in hundreds of nanoseconds.
Value	4	Measurement value
Flags	2	Quality bit indicators

Table 6 GEP Message format [22]

A subscriber can subscribe, unsubscribe, request metadata and control some aspects of the communication by sending a Subscriber Command to a publisher. The publisher responds with the appropriate response codes and metadata. Once a subscriber subscribes to a data stream, the publisher begins the data transmission through a UDP channel. The data package format used allows for multiple optimizations to be made on the data stream by reducing the size of the packets.

Streaming Telemetry Transport Protocol (STTP) is a standard was under development and formally adopted towards the closure of the research in 2024. IEEE Std 2664™-2024 [24] was constructed upon the experiences with GPE. STTP implements two channels: a data channel and a command channel. However, unlike GEP, the data channel can be established through TCP as well. TCP-based data channels can use TLS encryption. Lastly, STTP utilizes a compression algorithm known as Time Series Special Compression (TSSC), which uses an XOR compression technique to get the changes between consecutive messages [25].

2.3 Introduction to Data Engineering

This section aims to provide the reader with a brief introduction to data engineering and the technological landscape. It will explore the techniques used to address large-scale problems and show how they could improve synchrophasor processing technology.

Data engineering is a discipline within computer science that focuses on collecting, processing, storing, and serving large volumes of data. The objective of the data engineering process is to provide high-quality data, enabling data analysis applications [2]. To achieve this, data engineering relies on techniques to manipulate large-scale datasets in distributed environments [26] and known patterns that aim to guarantee some desired constraints [9].

The process of moving raw data from the source to its destination in the form of high-value data is known as the data engineering lifecycle. The lifecycle starts with data generation, ingestion, and processing and ends with serving the resulting data to make it available for further analysis [28]. Moreover, multiple forms of data storage are present during the process to support each stage.

Data engineering and the broader field of Big-data have been steadily evolving during the last two decades. While databases have been around since the beginning of the computing era, storage was costly and limited for the first years. It was in the dawn of the public internet services after 1994 that the scale of the data produced by them began challenging existing data systems and gave birth to the concept of Big-data, bringing with it the development of the data engineering field [29].

Data engineers' main task is to retrieve data from multiple heterogeneous sources, preprocess it, curate it, and make it available for analysis by domain experts. Data engineering began with the task of cleaning and preprocessing data as part of the data science process. Then, data engineering evolved into a field of its own and started developing tools for data scientists to carry out these tasks themselves. Throughout its evolution, some of these tasks have been identified and formalized: data understanding, cleaning, correction, and transformation. Later, the tools evolved into pipelines that combine multiple tools into reusable pieces of software [30]. From this perspective, data engineering is concerned with creating tools and methodologies to deal with low-level data manipulation and the issues that arise from scale.

The evolution of data engineering and Big-data systems is thoroughly detailed in [31]. It traces the developments from its beginnings at Google™, where engineers had to create large-scale parallel processing jobs to handle large volumes of data, to the state-of-the-art tools used today. Examining data engineering tools from the point of view of the problem they were designed to solve, provides excellent insights into the motivation behind them as well as how they contributed to the field.

Google™ File System (GFS) is a large-scale distributed file storage system. It was designed to store large datasets in multiple nodes. By then, some technologies allowed redundant network storage using specialized hardware. GFS, however, provided a fault-tolerant, distributed file system to support data-intensive operations while running in commodity hardware [32]. The first processing jobs were standalone pieces of software independent of each other and had to deal explicitly with the complexities of interacting with a large distributed system; soon, they noticed similarities between their programs and a common pattern emerged. This pattern is known as MapReduce [33], and it is a layer of abstraction that allows developers to separate the processing logic from the complexity of dealing with a distributed system. Realistically, data processing requires complex assemblies of multiple MapReduce processes; the concept of a processing pipeline was introduced as a higher level of abstraction to deal with complex MapReduce job assemblies.

Hadoop was released as an open-source project. It combined a distributed file system with a framework for running MapReduce jobs [34]. MapReduce is aimed at processing large batches of data stored in distributed file systems, making it more suitable for high-latency jobs that produce periodic results rather than real-time analytics. Early stream processing engines were introduced to handle real-time and near-real-time processing scenarios and were first used together as a complement to batch processing jobs.

Apache Storm was introduced on Twitter for stream processing; however, it lacked some features, such as a notion of event time and watermarking to handle out-of-order events. In the context of synchrophasor data, synchrophasors are tagged with a precise timestamp at the source, which represents the time at which the sensor took the measurement; this time tag is known in the context of stream processing as event-time. The lack of this feature is inconvenient if not unsuitable for

handling time-tagged time series because it requires an additional effort to handle out-of-order data and time alignment.

The gap between stream processing and batch processing technologies shaped most architectures. It posed a problem as it meant that two radically different paradigms had to coexist in the same system. Apache Spark was introduced as a successor or legacy MapReduce data processing system and aimed to improve computation times by providing an abstraction for large distributed datasets called "Resilient Distributed Datasets" (RDD); these datasets could exist in memory or be persisted to hard disk drive (HDD) [35]. Spark Streaming allows for the running of streaming jobs leveraging a technique known as "discretized streams," which periodically creates small batches and processes them.

Streaming systems continued to evolve independently, and some developments shaped them into their current form. Early stream processors used message brokers to send messages between the processing stages of a stream processing job. Message brokers and direct TCP connection messages are ephemeral, which means that once a message is sent or read, it is deleted. Apache Kafka was introduced in part to overcome this issue, which was one of the shortcomings in the early days of stream processors, by providing a distributed message broker with the ability to persist data [36].

Today's stream processing frameworks look much different from their predecessors. Apache Flink is a modern unified stream and batch-processing framework introduced in 2014. Before it, batch processing and stream processing were thought of as different and isolated paradigms. Moreover, Apache Flink was designed to overcome the limitations that existed in other frameworks, such as the lack of a notion of event-time, watermarking, fault-tolerance capabilities, etc. [37].

2.3.1 Dataflow Computing

Dataflow computing is an abstraction in which a computing system is described by a directed graph, where vertex represent operations and edges describe data paths describing a dependency relation between operations.

The dataflow model was initially proposed as a processor architecture to improve parallel processing operations. However, the development of hardware to support the dataflow model stagnated and was never used in practice. Years later, the dataflow model found its use in data engineering, describing systems capable of processing large volumes of data [38].

Consider the following problem to illustrate the idea of the dataflow computing model: A group of sensors emits data to a data broker; each message consists of a channel ID, group ID, timestamp, and value. An event or alarm must be emitted every time more than a given number of sensors in a group contain anomalous data during a time window. Figure 3 depicts a dataflow program example to address the former scenario. The first column represents the sensors producing data parallel to each other; the second column represents a group by sensor ID operation that sends each sensor ID to the corresponding partition so that they are processed by the same instance every time; the third step implements an anomaly detection algorithm, this operates independently on each of the sensors and outputs an event when an anomaly is detected on a sensor data stream; the fourth column represents a second group by group ID that routes the events to the corresponding windows; lastly, the events inside a time window are counted, and an event is created when the given condition is met. Each column represents a step in the process and rows represent the parallel instances of the task. This simple example illustrates the dataflow abstraction, showing why it is a powerful tool to represent large-scale programs and how it makes it easier to conceptualize parallel operations.

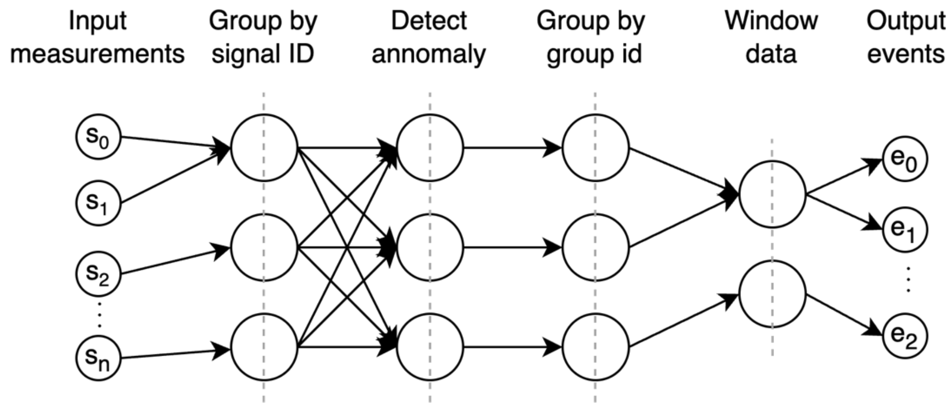


Figure 3 Dataflow Model Example

2.3.2 Stream Processing and Apache Flink

The demand for obtaining faster results from large amounts of data and the need for interactive, close to real-time processing drove the development away from batch processing systems such as Apache Hadoop and micro-batching like Apache Spark towards what is known as stream processing systems [39]. At a high level, stream processing systems can process data as it arrives and control the size of batches based on a time heuristic; in contrast, batch and micro-batch systems process groups of data once the whole dataset or micro-batch is available. While this difference may seem subtle, it is not trivial; stream processing systems must provide consistency and correctness guarantees that are difficult to realize in practice. A relevant instance of one such mechanism can be seen in the way Apache Flink deals with failures to provide exactly-once-processing consistency [37]. This section introduces the core concepts of stream processing and streaming analytics as they are implemented in Apache Flink.

In a stream processing job, the time at which an operator observes an event may have no relation with the actual time at which the event was emitted at the source. This idea is well understood in the context of synchrophasor data and motivates time synchronization at the source. In the context

of stream processing, the time at which the source emits an event or measurement is known as event time. In contrast, the time at which an event is observed is known as processing time. Unlike previous stream processing engines, Apache Flink distinguishes between event-time and processing time [31] [37].

Watermarking is a mechanism used by many stream-processing frameworks to deal with events that arrive in disorder. A watermark represents the earliest time for which a stream processing pipeline assumes it has seen all events. This means that if an event has a timestamp lower than the current watermark, the processing job can consider it to be a late arrival. Watermarks are updated periodically based on the latest seen timestamp and propagated through the stream processing job [40].

Stream processing jobs consist of streams and operators arranged in a directed acyclic graph that represents the flow of the data [37]. An operator can be any function that applies a transformation to one or more data streams and can produce one or more data streams.

Nevertheless, generic primitive operations have been built into most stream-processing frameworks; some of the most common are depicted in Figure 4. A map operator (a) takes a data stream and transforms it into another data stream, emitting an output event per input event. A flat map operator (b) takes a data stream, applies a transformation and produces one or more output elements; it is commonly used when flattening data streams. A filter operator (c) takes an event and emits one or none depending on a condition being evaluated to be true or false. A “key by” (d) operator assigns a key to an event, creating a partitioned stream. Reduce (e) transforms a partitioned stream into a data stream by performing an aggregation function on the elements; for instance, after an ID keys a stream, the reduce operator can be used to compute a rolling sum on

each partition. Lastly, a windowing operator (f) creates groups of events that an aggregation operator processes together, for instance, to calculate a rolling average.

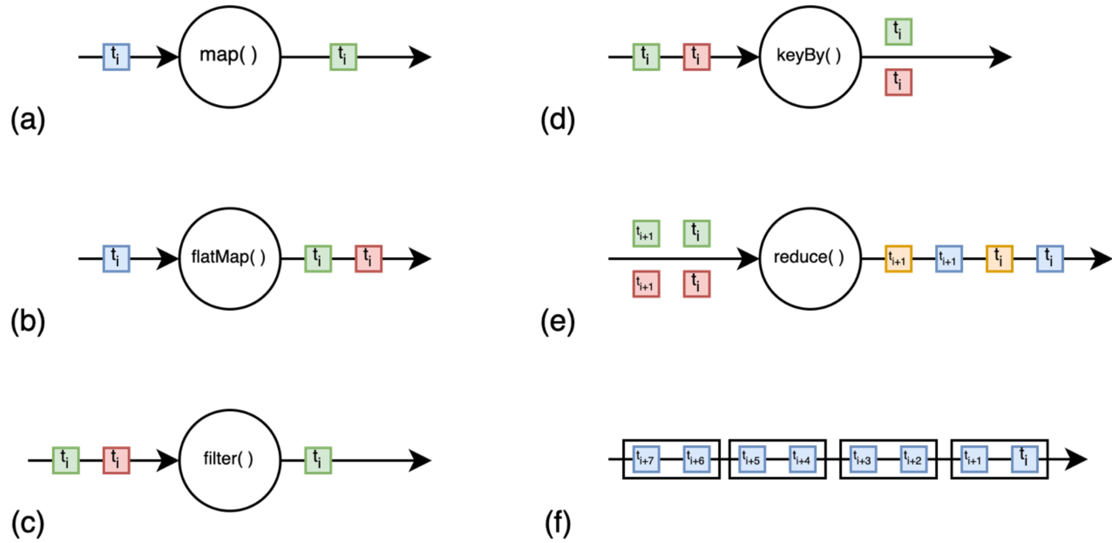


Figure 4 Stream Processing Primitive Operators [41]

Window operators are essential because they allow the application of an aggregation function to groups of consecutive elements. While windows can be created under arbitrary conditions, the three most common are tumbling windows, sliding windows, and session windows [41].

Tumbling windows are of fixed length and do not overlap. Figure 5 depicts the progression of a tumbling window in time. White squares represent events arriving at the window, and gray squares represent past events that are discarded after the window advances.

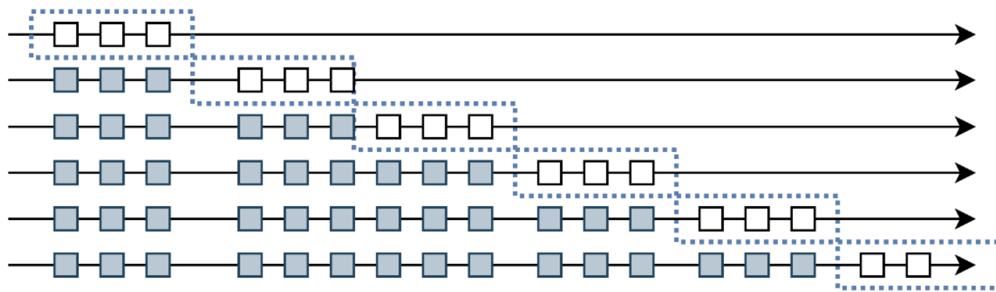


Figure 5 Tumbling Window

Session windows start when an element is received and remain open until no elements are received for a fixed amount of time; session windows cannot overlap. Figure 6 depicts a session window; the width of the window increases as new events arrive, and the window is closed after a time of inactivity.

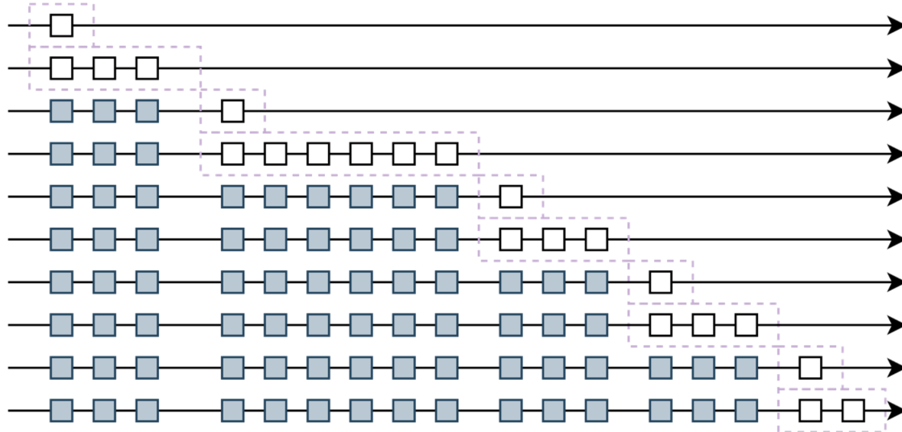


Figure 6 Session Window

Sliding windows, on the other hand, are of fixed size, but consecutive windows overlap. Figure 7 depicts a sliding window; as the window progresses, events in the overlapping segment are not discarded.

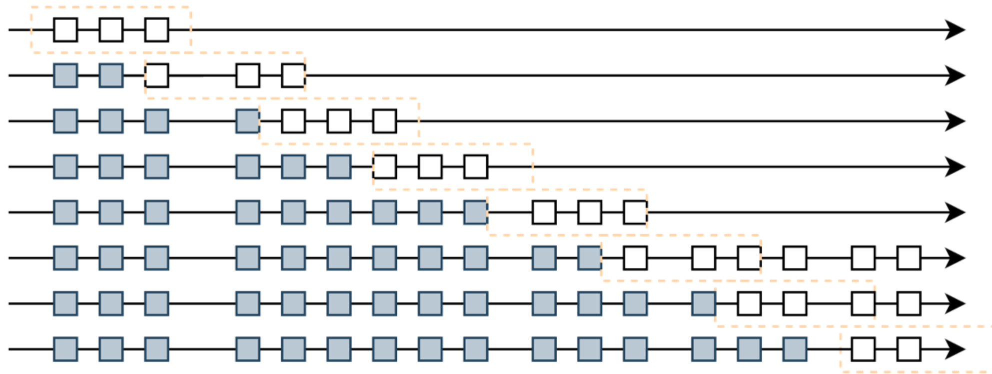


Figure 7 Sliding Window

Event-time windows use watermarks to determine whether an event belongs to the window or not. As explained before, watermarks originate at sources from the timestamp of arriving events. A watermark is an event that is propagated through the processing pipeline to other operators. Window operators will emit a watermark to downstream operators when the window is completed. Operators with two inputs propagate the minimum of the two received watermarks [42].

2.4 Synchrophasor Big Data Platforms

2.4.1 The Frequency Monitoring Network (FNET/GridEye)

FNET was introduced in 2003 and is a cost-effective and easy-to-deploy solution for monitoring frequency in a power system. It was proposed as a higher-resolution alternative to the existing steady-state frequency measurements and to the existing synchrophasor networks, which had limited coverage at that time and were typically expensive to install. The fundamental idea of FNET is that the frequency can be accurately measured at power outlets as it corresponds to the frequency measured at the transmission grid node to which the distribution system is connected. A frequency disturbance recorder (FDR) with a GPS and internet connection is used to measure the frequency of the power outlet [43].

FNET started as an SQL database with a web interface and a service to handle communication with the FDRs. The service received data from up to 50 FDRs [35-36]. Then, it evolved into a multi-server system consisting of a data concentrator and data storage using Microsoft Access. The Microsoft Access database is used as a historian; batches are created when multiple records arrive in the database. Every night, the database was offloaded to a permanent storage server [45].

Lastly, Figure 8 depicts the most recent architecture reported for the FNET/GridEye system. Data is ingested through openPDC and recorded in openHistorian, where it can be accessed by the analytics cluster for near-real-time applications (below 2 minutes) and offline applications. Real-time applications are implemented directly by extending openPDC exposed interfaces. These are meant to be simple computations (below 10 seconds) that can trigger more near-real-time applications. This system architecture was motivated in part to support the ingestion of around 200 FDRs [46].

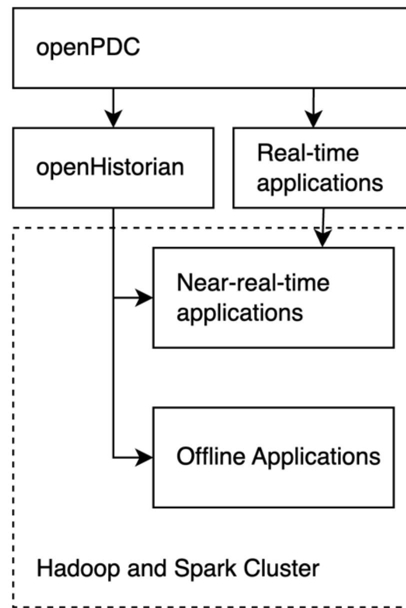


Figure 8 FNET/GridEye Analytics Architecture

2.4.1 BTrDB and Distil [11-14]

BTrDB is a time series database for high-frequency, real-time telemetry data, such as distribution synchrophasor data. It was motivated by the low throughput and low time resolution available in typical time-series databases. The authors report that its success is derived from their novel data structure, which is described as a "*time-partitioned copy-on-write version-annotated k-ary tree*". This can be broken down to understand what each part means and what they aim to solve.

Searching for a value in a large sequence of records is computationally expensive because, without an additional data structure, it is necessary to look at each of them to find the subset of values that match specific criteria. An index is a data structure used to store metadata that describes the underlying structure of the record set to decrease the computational cost of looking for one or more values. A tree is a data structure often used to create indices, as it allows for the division of a set

of records into branches, each pointing to smaller, organized subsets of records. A k-ary tree [47], like the one depicted in Figure 9 is a type of tree where each node can have at most k branches.

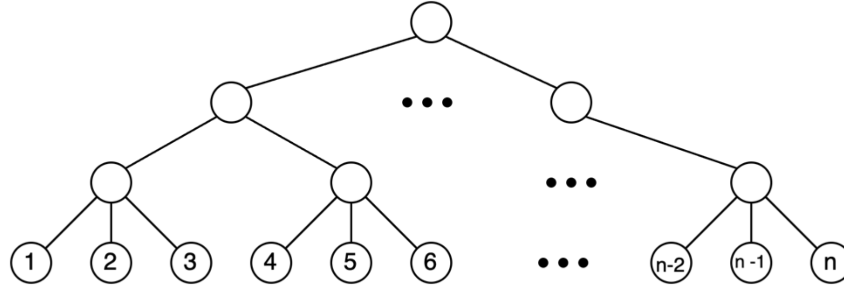


Figure 9 k-ary tree Illustration

Time-partitioned refers to the fact that each node in the tree represents a time interval and branches into smaller intervals. The tree leaves contain individual data points at a particular time, and each node can store statistical measures of the interval it contains.

Each time data is inserted into a tree, BTrDB tags it with a version. This allows the tree data structure to store information about the arrival order and record multiple versions of the stream that appear after it is cleaned or processed. Lastly, new versions form an overlay tree over the existing ones, with the inserted data only.

DISTIL is a framework for creating dataflow graphs for processing time series streams built on top of BTrDB. The framework consists of a set of stateless data processing functions and a file that describes the dataflow graph and how processing nodes are connected [14].

2.4.2 GE Research PMU Dataset Cluster [48]

In this paper, the authors discuss the design of a compute cluster to perform offline analysis of massive synchrophasor data sets. The cluster is a 63-node cluster that stores 18.5 TB of

synchrophasor data collected over two years from 443 PMU. This cluster relies on Apache Hadoop HDFS for data storage, YARN for job scheduling, HIVE for querying the data, and Apache Spark for processing and analytics.

The main objective of data engineers in designing this cluster is to provide power engineers with effective software abstractions to access the data in the cluster without having to deal with detailed data engineering manipulations such as handling data partitions. To achieve this, the authors first identified the most common use case for the cluster, which is feature generation over sliding windows and data exploration. The authors report the software abstractions, setup and measures they took to improve performance.

The authors describe their query strategy as a predefined set of Spark SQL query templates. Spark SQL is a native Spark capability that results in optimized queries and execution plans. Moreover, the templates are optimized for each use case and only need to be parametrized for each query. This not only results in higher-performance queries but also creates a high-level abstraction where engineers do not have to deal with complex data engineering issues such as partitioning.

Feature computation is described as a layered process where these calculations are organized in a way that the cluster can handle parallel processing jobs to compute them. The first layer is referred to as the feature function, which is a primitive processing function that operates locally on arrays and can compute statistical measurements like mean, standard deviation, etc. The second layer is called the feature wrapper, which consists of multiple feature functions grouped; these are applied to data frames to compute features from numerous variables. The third layer is the feature generation engine, whose functions are to extract time-windowed slices from the data set, preprocess the data, apply the feature wrappers, perform post-processing functions and lastly, organize the output into output feature data frames. The final layer is a lower-level layer that

distributes the processing job across multiple nodes and is referred to as a feature generation executor.

2.4.3 Apache IoTDB [12]

IoTDB is a novel time-series database designed to support time-series data ingestion from IoT devices. IoTDB proposes a general approach to handling real-time time-series data ingestion, and unlike the other platforms presented here, it is not uniquely aimed at synchrophasor data. Moreover, unlike the previously mentioned platforms, IoTDB places some near-real-time functionalities like late arrival handling on edge servers and historical data management on cloud servers.

The creators of IoTDB highlight four pervasive data characteristics in IoT applications. These are the absence of a fixed schema due to possible replacement in sensors or redundancies, periodic data with the possibility of slight variations in their timestamps, delayed data arrivals, and omitted points. These characteristics served either as design constraints or were exploited to optimize the performance of IoTDB; they are summarized in Table 7 and thoroughly discussed by its authors in [12].

Feature	Description	IoTDB approach
Device-defined Ever-evolving schema	Sensor replacement or data redundancy can lead to changes in the names that identified a measure. Traditional databases are ill equipped to deal with this situation. IoTDB sought to address this issue in its design.	IoTDB organize time series metadata in trees that allow to group different channels under the same branch, this can be used to indicate a device replacement or redundancy of the same measurement.
Periodical data collection: regular time interval	IoT measurements can be expected to be generated with fixed time steps.	A regular data rate is assumed, and measurements are aligned to it. This fixes small variations by rounding measurement timestamps to the previous timestep. In addition, groups of measurements in the same file or partition are aligned so their timestamps are only stored once.
Periodical data collection: equal measurements are skipped	IoT sensors often avoid sending data if a variable has not changed to reduce network bandwidth.	IoTDB fundamental data file supports storing repeated measurements as empty data points on subsequent timestamps.
Time series correlation	As it is the case for power systems with variables in nearby nodes, large groups of IoT sensors often produce correlated time-series.	This is exploited for data encoding. A regression is calculated over the timeseries and only the difference between the regression values and the true value are stored.
Delayed data arrival	Buffering and partitioning can lead to late or out of order arrivals.	When late arrivals occur, these can be grouped either with other data points arriving at the same time or by creating groups of late arrival data points only. IoTDB is capable of determining which strategy to use depending on the nature of the delayed arrivals.
High ingestion concurrency	It is well known that groups of IoT sensors and sensor networks produce large amounts of data.	Data replication limits the parallelism in the cluster, the creators propose a novel non-blocking consensus algorithm to achieve a high ingestion throughput.

Table 7 IoTDB Design Constraints

Chapter 3 Cloud Platform Overview

This section presents an overview of the cloud platform for synchrophasor data processing, its components and the testing setup used for validation. The platform comprises all the elements needed to forward measurements from the PMUs to the cloud. Moreover, a high-level explanation and overview of each will be provided.

In practical terms, moving from traditional hierarchical synchrophasor networks to a cloud-based systems have its own trade-offs, and it is not a one size fits all solution. The main advantages of creating a cloud-based platform are flexibility, interoperability with existing big-data frameworks and often lower capital expenses because it requires less physical infrastructure. On the other hand, the disadvantages of a cloud-based system in contrast to using traditional hierarchical networks are increased operating costs depending on the amounts of data, reduced ability to control latency as data is sent through a shared medium (this can be overcome by using private channels which are offered by some providers) and the inability to exploit flexible cost models because the stream of data is constant. The advantages of using a cloud-based platform can outweigh its disadvantages in scenarios where the cost of building physical infrastructure is too high, as it can be the case for some distribution substations, when the equipment will be deployed temporarily or when the type of data analysis requires a variety of tools and data sources that may not be readily available close to the PDC physical machines.

3.1 Overall Structure of the Measurement System

In computer networking, the client-server communication model determines which actor in a network, known as the client, is responsible and has the ability of stablishing the communications

with another agent known as the server which has the ability to allow connections from one or more clients. In the context of synchrophasor communications, a PMU acts as a server and a PDC is both a client that connects to remote PMU and a server to which other PDC can connect to consume the output data streams. Traditional synchrophasor networks are often architected, as shown in Figure 1, so that substation PDC, the ones in charge of aggregating all the relevant measurements from the substation, are reachable by their corresponding regional PDC. This is often achieved by building a last mile dedicated link from the substation to an MPLS [5] network that can be used to connect it to the remote regional PDC. Without the former setup, it is not possible to reach a substation PDC either from a regional PDC or the data centers of a cloud service provider. To overcome this without relying on the aforementioned complex solutions, an interface that initiates the connection to the PMUs or substation PDC and then forwards the data to the cloud is necessary.

The high-level arrangement of the cloud-based synchrophasor data collection and processing platform is illustrated in Figure 10. The interface that collects data inside the substation and forwards it to the cloud is referred to as the substation agent and it is described in detail in Chapter 4. The substation agent, shown in Figure 10 , is designed to be a tool that allows to stablish a communication channel between local PMU or PDC inside a substation and to forward the data to a system in the cloud. To this end, an IEEE Std C37.118-2005 library and client are developed to collect the data and then a data transformation and forwarding stage pushes the data stream packages to a cloud service. The advantage of this approach is that the communication can be set up to use a public internet channel without special firewall configurations that allow connections to be initiated from outside the substation; for instance, from the regional PDC. Furthermore, the substation agent has some tools to help stablishing a connection to local PDC and PMUs, a way to

create and store channel mappings to keep the channel IDs and can be extended to forward the data streams to different data systems.

An ingest interface, shown in Figure 10, is necessary to receive data from multiple substation agents. Amazon Kinesis, a distributed message broker is used for this end. Amazon Kinesis was chosen due to its simplicity; however, other distributed message brokers are also suitable for this task. The main objective of this layer is to receive measurements packets from the substation agent and to make them available for the pipeline to consume them when needed. The broker is setup with multiple partition which increases its throughput by allowing more producers to send data simultaneously. The message broker and how it is connected to other components of the system are shown in detail in Chapter 5.

Lastly, the data in the broker is consumed by a streaming application running on an Apache Flink cluster that spans across multiple compute nodes, this is shown in Figure 10. In Figure 10, block (a) depicts the substation agent and data collection at the substation level from PMU/PDC, block (b) depicts the distributed message broker used for data ingestion inside the cloud platform, and block (c) depicts the stream processing application to process the data inside the cloud platform. The streaming application is responsible for routing, windowing, processing the data and finally of forwarding it to its destination on a persistent storage, analytics database or the broker itself to be consumed by other near-real-time applications. This layer is described in detail in Chapter 6.

One advantage of this setup is that it takes advantage of the security mechanisms inherent to the cloud systems for transmitting data. The AWS Kinesis Distributed Message Broker uses HTTPS encryption by default, this is secure enough for most applications as it can guarantee integrity and confidentiality.

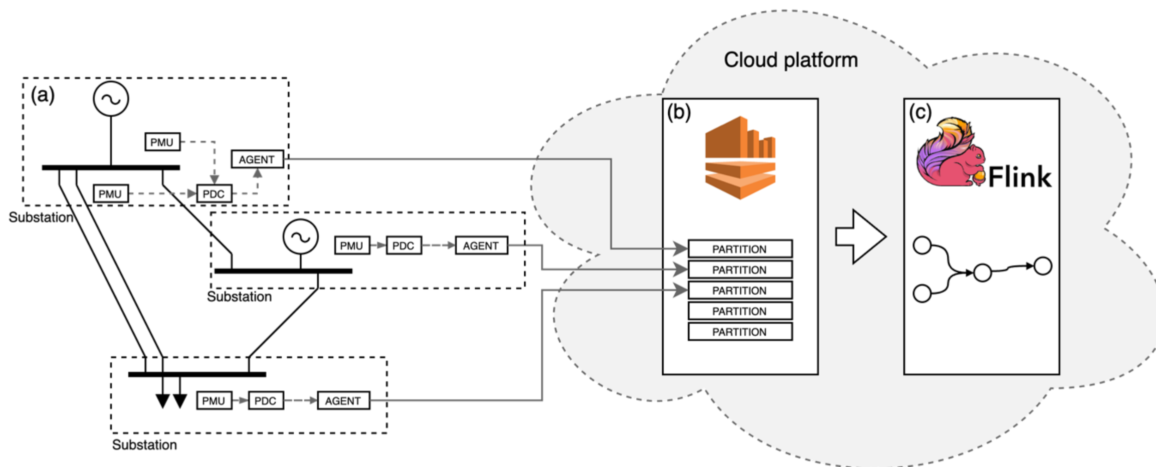


Figure 10 System Overview. a) substation b) distributed message broker c) stream processing application

3.2 Testing and Validation Setup

Testing requires either the use of real PMUs or a real-time simulation with emulated PMUs; there are two main reasons for this. The first reason concerns the behaviour of PMUs. Even though some existing PC-based PMU emulators were explored, they failed to reproduce the exact behaviour of a PMU. The PC-based PMU emulators generated the FRACSEC value from the timestamp at the time the message was generated rather than at the regular periods expected for a given data rate. The second reason is that data needs to be generated at regular intervals, and while this can be achieved with a PC-based PMU emulator, real-time simulation with PMUs better reproduces this behaviour. The difference between periodic data generation in the real-time simulator and generating all the testing data at the same time and sending it to the queue is that stream processing frameworks often can optimize the transmission of larger batches. In contrast, periodic generation represents the worst-case scenario as it has a higher network overhead. The concept is better

illustrated by describing how data is consumed from Amazon Kinesis™, where consumers read up to a maximum number of records and make them available for processing. The broker client requests all available records up to a maximum value. When data is generated periodically, the number of available records is small requiring the client to make frequent requests to the broker. In contrast, when a large number of messages are available in the broker, fewer and less frequent network transactions are required, making it more efficient. This is, however, less realistic and justifies the need of periodic data generation by means of a real-time simulation.

A real-time simulation model of a power system was implemented on RTDS real-time digital simulator. The power system conceptual diagram is presented in Figure 11. The system is composed of two large generators connected to a large randomly varying load, a distribution feeder with a load, and an inverter-based generator. The objective is to generate typical power system waveforms, capture them with PMUs, and send them to be processed in the data pipeline. Five PMUs were placed at multiple nodes, as shown in Figure 11.

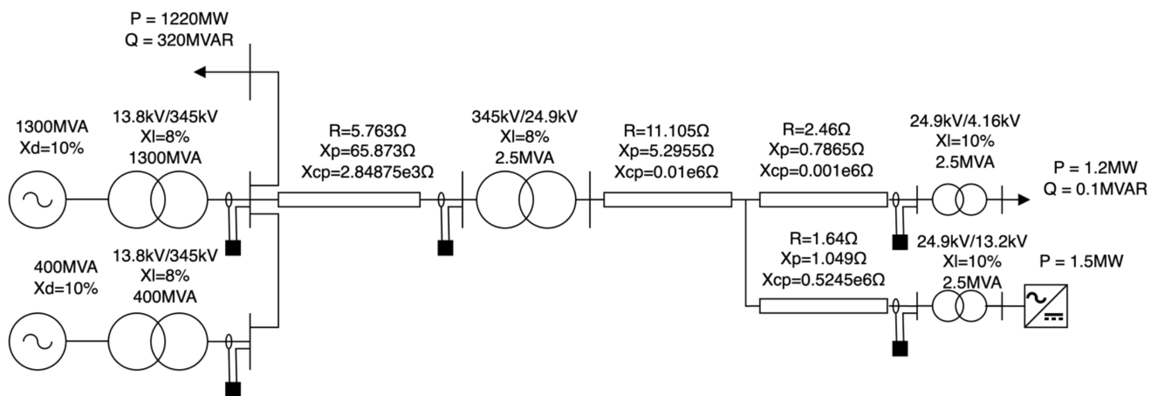


Figure 11 Test Power System

The simulation setup is shown in Figure 12. An RTDS simulator running the system shown in Figure 11, the simulation clock is GPS synchronized via a GTSYNC card connected to a SEL 2407

Satellite clock. Emulated PMUs in RTDS function exactly the same as a real PMU except that input signals are taken from the simulated waveforms. These PMUs, which provide their outputs through the GTNETx2 card of RTDS, and a local computer which hosts the substation agents are connected to the same network through a network switch. Various substation agent instances run concurrently in the local computer. The substation agent instances initiate the connection to the emulated PMUs and forward the data to the cloud, where it is ingested and processed. For simplicity, a PDC was not explicitly included in this test setup, but can be easily accommodate if needed.

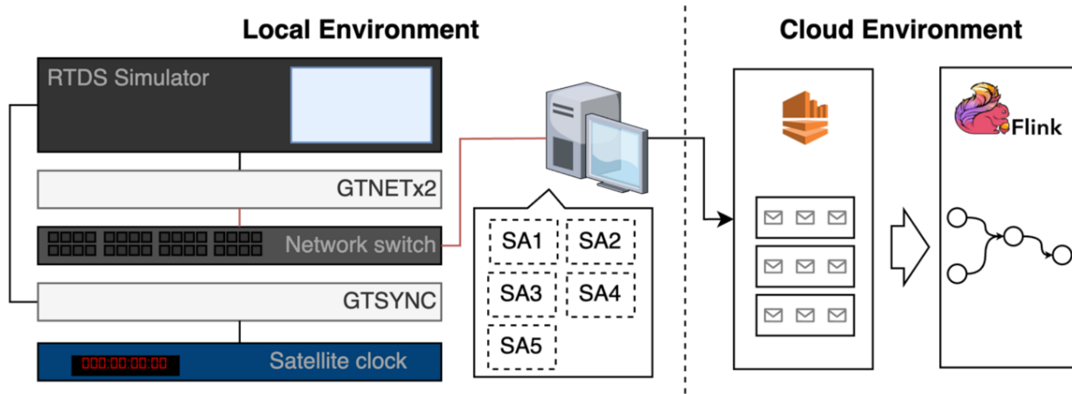


Figure 12 Simulation Setup

3.3 Summary

This chapter presents the overall structure of the setup to extract PMU/PDC data from substations to the cloud for further processing. The components forming the system are presented as well as the concepts that are relevant to the implementation. Finally, the test and validation setup using real-time simulation is presented.

Chapter 4 Substation Agent

The substation agent is a modular Python library and command-line tool. Its primary purpose is to forward data from PDC or PMU to the cloud for data collection. Nevertheless, it can be used to support other use cases such as PMU simulation, IEEE Std C37.118™-2005 server, etc. This section describes how it is architected, methods of testing, validation strategies, and how it can be extended.

The substation agent is software designed to run in a computer inside a substation and whose objective is to capture synchrophasor data from a PMU or PDC inside a substation, manage the communication with the measurement device, transform and forward the data to the next layer.

This chapter begins with an in-depth description of the design and architecture of the substation agent; this subsection presents both a detailed description of the way in which it was implemented as well as an explanation of the design decisions that were taken. Following this, it presents how the substation agent is intended to be used, this subsection delves into the user facing interfaces that are implemented.

Finally, this chapter concludes by demonstrating the proper operation of the substation agent. It describes the testing and validation procedures carried on the substation agent and discusses the results to ensure the software is functioning as intended.

4.1 Design and Architecture

The design can be better understood by dividing it in two layers. Each layer carries out a different and identifiable function. The first layer manages the connection with PMU and PDC by keeping the connection state, overseeing frame synchronization and deserializing the frames from a binary string to a Python object.

Connection state management is responsible to send the appropriate command frame, set the client state to receive either the configuration frames or data frames and to restart the connection when it fails.

Frame synchronization is responsible for detecting the beginning and end of each binary frame as it is received. This involves seeking through the received bytes until the SYNC word is detected, discarding the necessary bytes, performing CRC check and lastly calling the deserializer to parse the received binary string.

Finally, the next three layers are responsible for transforming individual measurements to Protobuf objects, creating small batches to be transmitted and forwarding them to the cloud. A layered overview of the process is depicted in Figure 13.

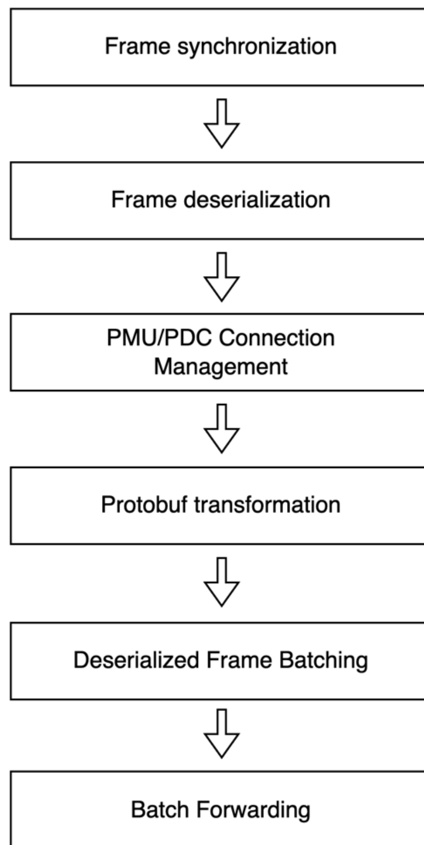


Figure 13 Substation Agent Layers

4.1.1 Synchrophasor Client and Frame Synchronization

This section describes the higher-level implementation details of the client library used to read synchrophasor data from PDC or PMU. The client is implemented in multiple layers: the agent controls the communication and operates on deserialized frames, the client handles the communication with the PDC or PMU, and the IEEE Std C37.118™-2005 handler implements lower-level frame synchronization and deserialization.

Figure 14 (a) is a flow chart depicting the process run by the substation agent. Once a communication channel is open between the agent and the PMU or PDC, the agent begins by

requesting and storing the configuration frames; then, it sends a command frame to initialize the data transmission. Every frame that is received is sent to an asynchronous handler that operates on the frame; the handler is application-dependent and can be set by extending the base agent. At the agent level, frames are bound and deserialized into Python objects.

Figure 14 (b) depicts the internal process of synchronizing and performing the deserialization on the reception buffer. This stage transforms the incoming stream of bytes into deserialized Python objects. The process starts with an empty buffer and a position indicator pointing at position 0. When bytes are received, they are stored in the buffer, and the received amount increases the position. The buffer is then cleaned to ensure the first byte matches the protocol synchronization byte, i.e. 0xaa; this updates the position accordingly to account for the deleted bytes. Once the first byte matches the synchronization byte and the third byte is received, it can extract the FRAMESIZE. Lastly, it waits until the buffer contains at least FRAMESIZE elements, and the first byte is 0xaa when it is deserialized using the IEEE Std C37.118™-2005 deserializer that is introduced in later in subsection 4.1.4. The appropriate configuration frame is set by the agent but is used in the deserialization when a data frame is received.

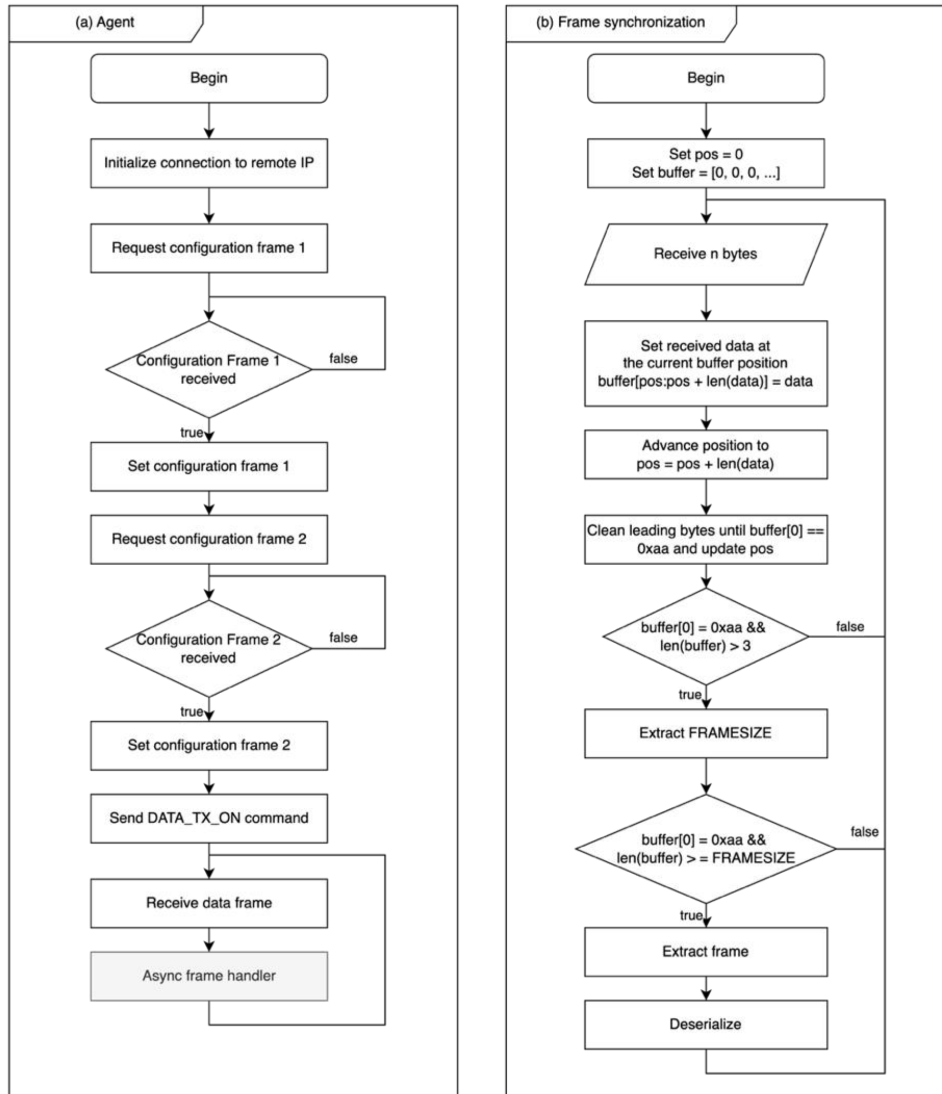


Figure 14 IEEE Std C37.118™-2005 Client and Frame Synchronization

4.1.2 AWS Kinesis Agent and Protobuf Serialization

Frames are serialized back into a binary string to be sent to the distributed message broker. The format utilized to serialize measurements is called Protocol buffers or Protobuf for short [49]. Protobuf is an open-source structured data serialization format developed by Google™.

A concrete implementation of the substation event was made to forward synchrophasor data batches to an AWS Kinesis stream for further processing in the cloud. Protobuf [49] is a data format widely used to send data and is commonly found in data-driven applications. The Kinesis substation agent transforms IEEE Std C37.118™-2005 data frames into Protobuf messages that are forwarded to AWS Kinesis. The defined data model allows for each Protobuf message to carry one or more data points with its stream ID and timestamp; this allows data to be sent in batches to optimize network transmission. The Protobuf models are depicted in Figure 15.

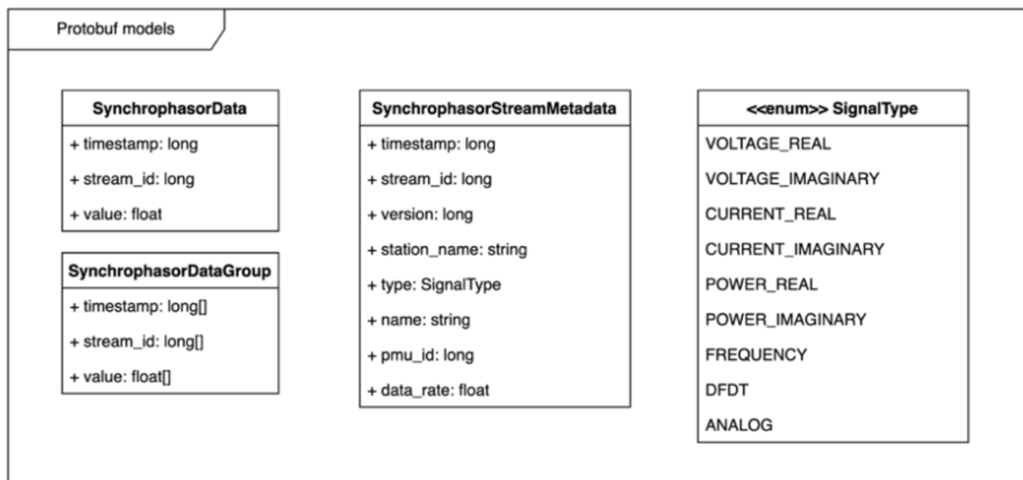


Figure 15 Protobuf Message Models

Data is sent to AWS Kinesis in batches created at uniform time intervals. Figure 16 shows the process of transforming multiple data frames in a batch of measurements stacked on top of each other. There are various reasons for choosing this structure. The first reason is that it follows the same general structure as GEP and STTP; this ensures that the software can be easily adapted in the future to support them. The second reason is that the tabular structure allows for some optimizations, such as only sending a timestamp when it changes and the use of Protobuf variable size encoding [50]. This effectively transforms data from a frame-based format to a measurement-based one.

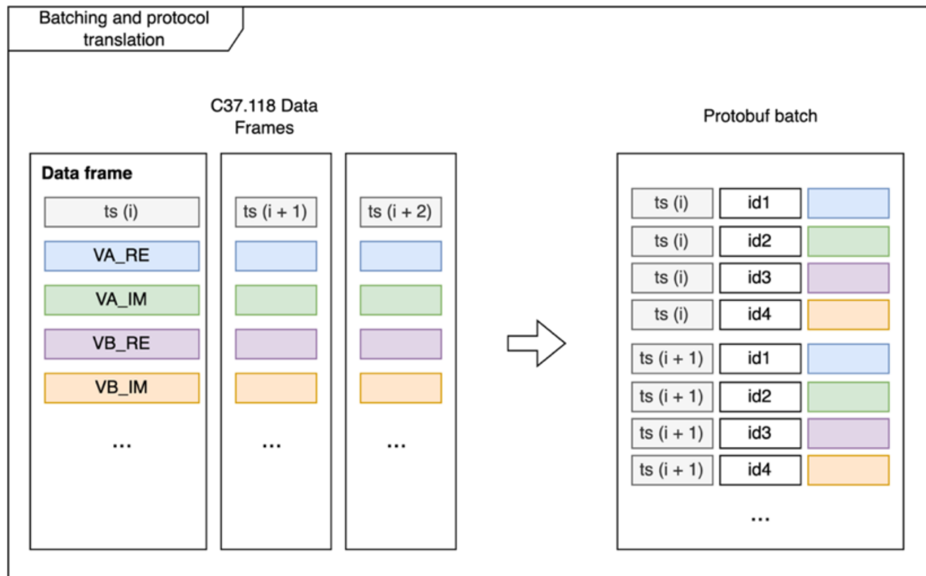


Figure 16 Protocol Translation and Batching

4.1.3 Data Models

The IEEE Std C37.118™-2005 library implements a data model that reflects the protocol message structure. This section presents the Unified Modeling Language (UML) diagrams and design. UML [51] is a graphical language for designing, documenting and describing software applications. Figure 17 contains the enumerations used throughout the models, these are shown for completeness and represent enumerations present in the IEEE Std C37.118™-2005 standard. Figure 18 presents the base abstract frame from which the other frames are derived and the unbound frame, which is a helpful way of thinking about partially parsed frames. An arrow with an empty tip represents an inheritance relation, in this case the UnboundFrame inherits the abstract base Frame. The TimeQuality class is related to the Frame through a composition association as it is the type of one of its attributes; this is represented by a solid line terminated with a diamond. TimeInfo is related to the class though a dependency association which is represented by a dashed line with a solid arrow; this means that the Frame depends on it through the get_time_information

method. Figure 19, Figure 20, Figure 21 and Figure 23 present the structure of header frames, command frames, configuration frames and data frames.

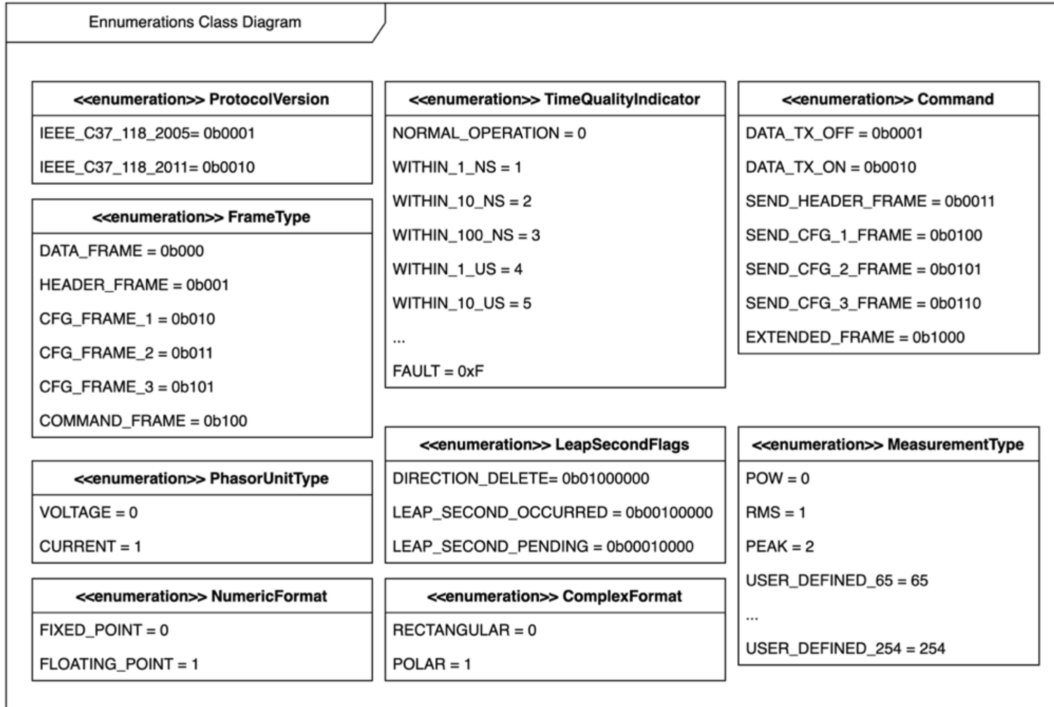


Figure 17 Library Enumerations

Figure 18 shows the base abstract frame; it defines the base fields that are common for every frame in the protocol. These fields (or words in the protocol context) are SYNC, FRAMESIZE, IDCODE, SOC, FRACSEC, and CHK. The remaining data are the set of unparsed words between the FRACSEC and CHL. An additional field called original is defined to store the byte array from where the frame was parsed. The unbound frame is a concrete implementation of a partially parsed frame where the DATA words unique to each type of frame are still unknown.

The *set_original* method defined in the Frame is responsible for setting the original byte array to the original attribute and the parsed checksum bytes to the checksum attribute as an integer. The second, *get_time_information,* is a static method that takes a float timestamp in seconds, time quality flags, and a time base and returns an object with the second-of-century and the

corresponding fraction of second for the time base. Lastly, the *from_frame* class method builds a new frame from an existing one.

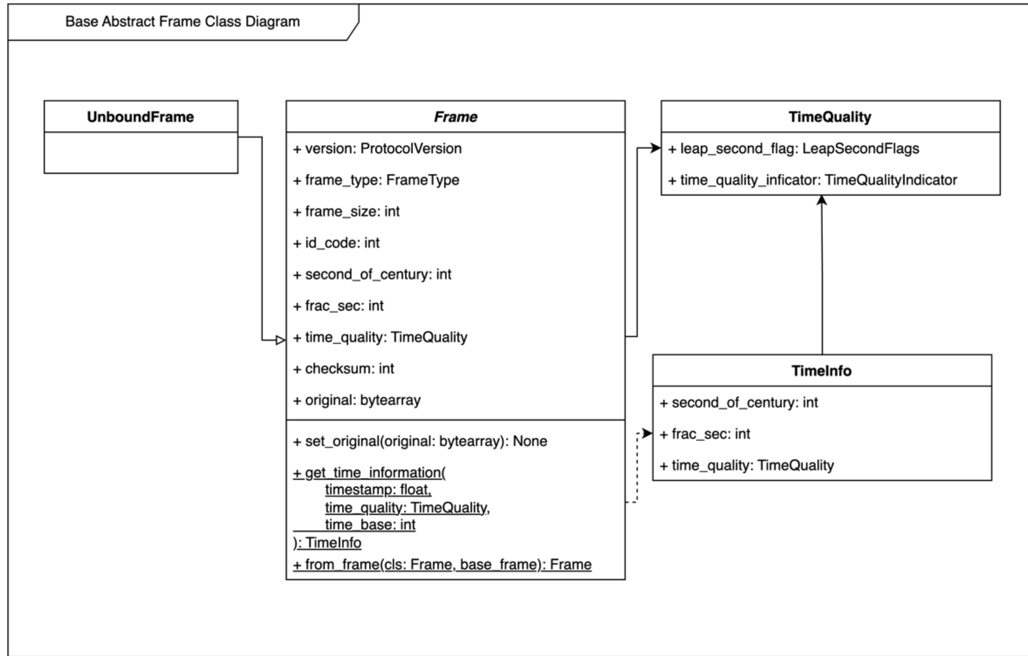


Figure 18 Abstract Base Frame and Unbound Frame

Figure 19 shows the definition of the header frame, the simplest of all the frames. The header frame extends the frame and defines a string attribute called data with the header. With the exception of the unbound frame, all concrete frames define two methods: create and *from_unbound_frame*. The create method is used to create an instance of the frame from an appropriate set of inputs, e.g. *id_code*, timestamp, *time_base*, *time_quality* descriptor and *data*. The *from_unbound_frame* is used to build a frame from an unbound frame and deserialized fields during the deserialization process.

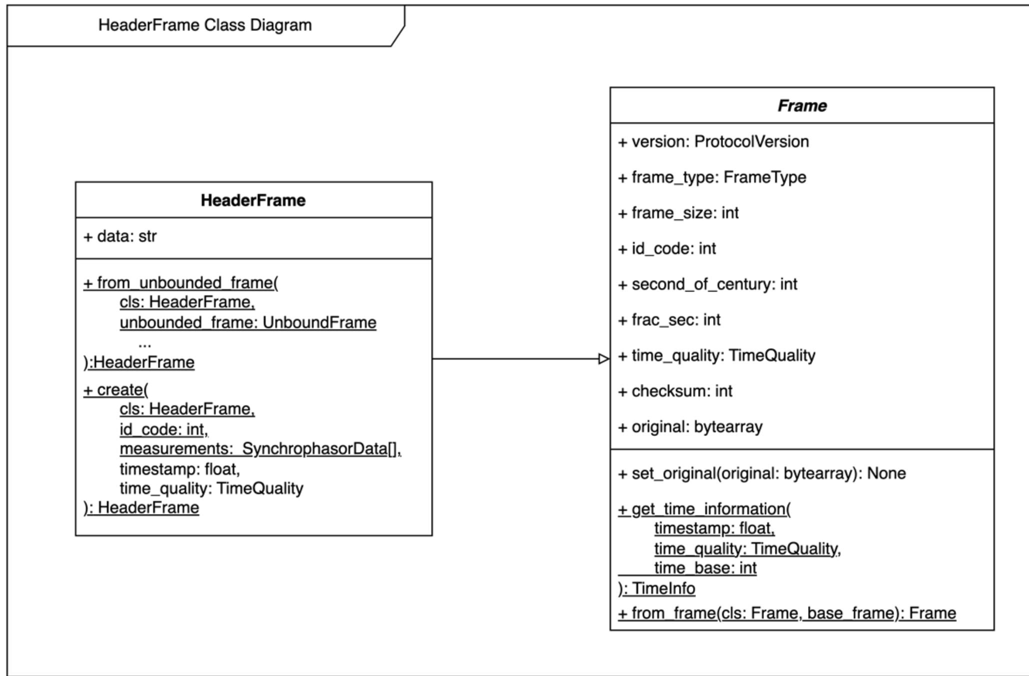


Figure 19 Header Frame Class Diagram

The command frame shown in Figure 20 is defined similarly to the header frame. It represents the attributes *command* and *extended_frame*. Command frames, unlike other types, are most commonly not parsed but generated by the client that wants to connect to a PMU or PDC; the *create* method is used here to build the command frame instances.

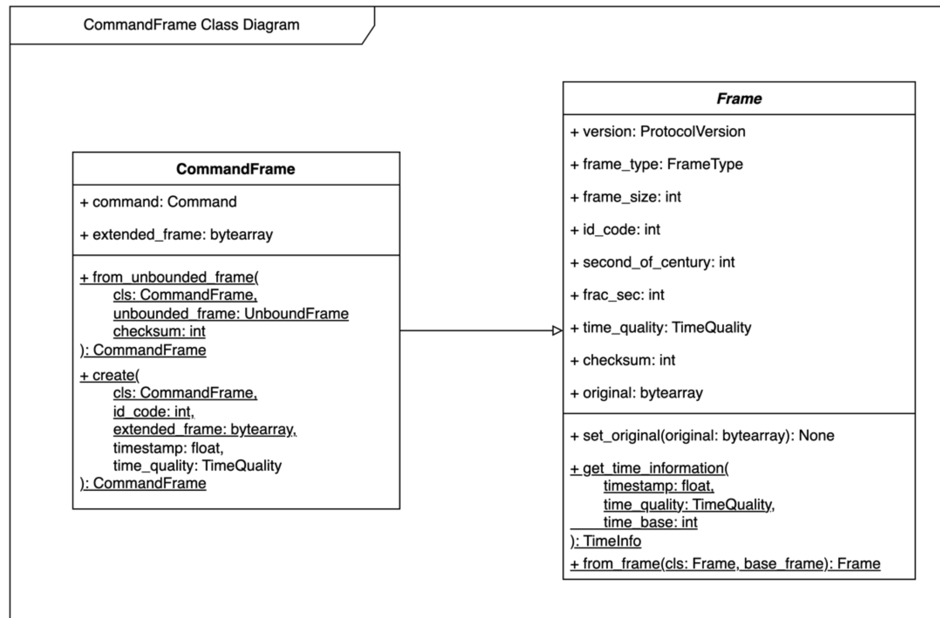


Figure 20 Command Frame Class Diagram

Figure 21 shows the configuration frames 1 and 2 structure. The configuration frame defined a *time_base*, *pmu_configuration*, *data_rate* and a private *pack_fmt*. The former three correspond to the words in the protocol structure; the latter is a string that is used to unpack the data frame payload, and a value is assigned to this attribute when the object is created so data frames can be efficiently parsed. Apart from the common *create* and from_*unbound*_frame methods, the configuration frame defines two other methods, *data_frame_unpack_format* and *get_data_frame_size*. The *data_frame_unpack_format* method uses the number of fields in the configuration frame to create a string that is used to unpack the attributes contained in the byte array payload coming in the data frame. The *get_data_frame_size* method returns a number representing the size of a data frame with the given configuration frame; the size is shown in bytes.

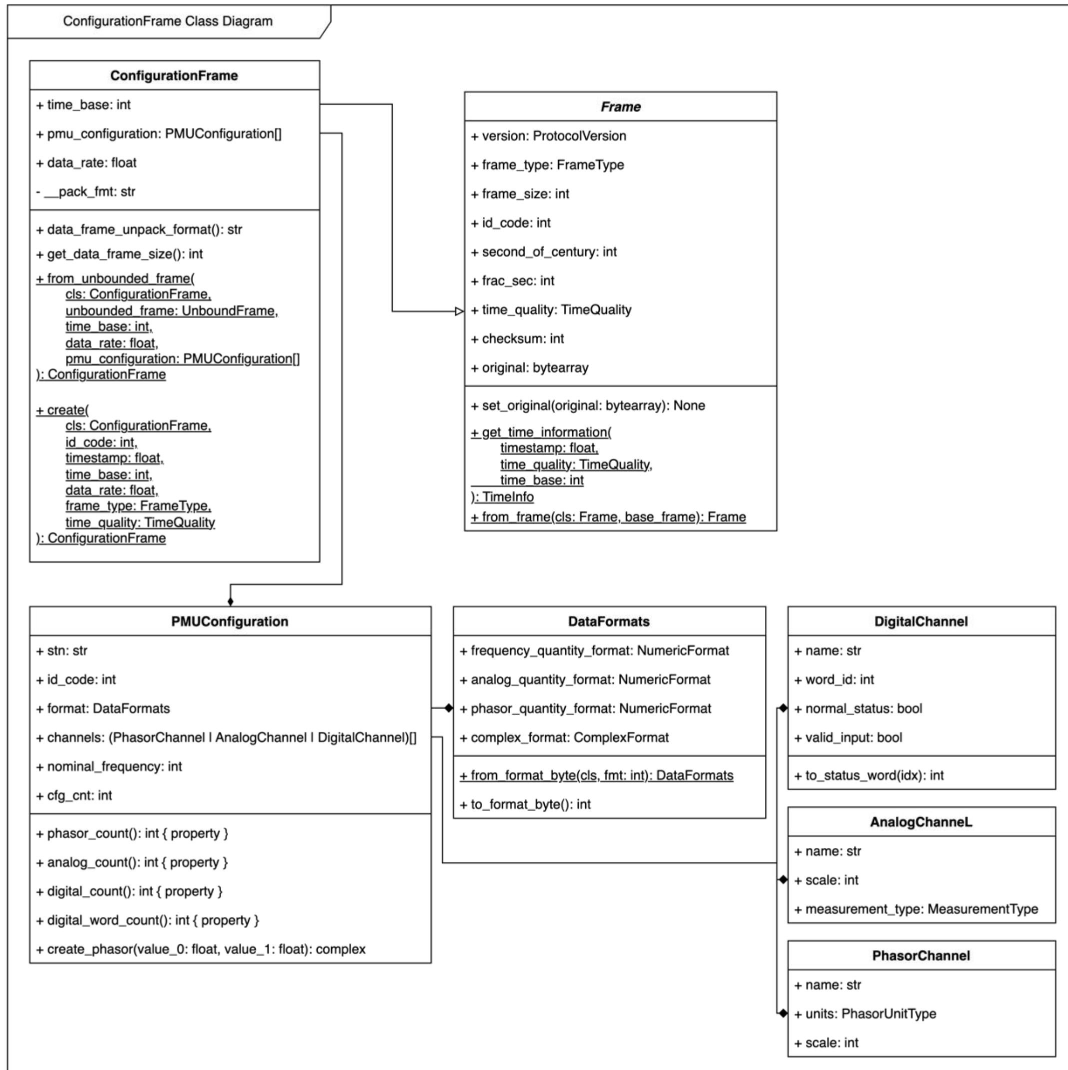


Figure 21 Configuration Frame Class Diagram

A configuration frame contains the definition for one or more PMUs in the data frame types. The PMU Configuration class represents the definition of a single data frame. The PMU Configuration class defines a special method, *create_phasor*, that creates phasors using rectangular or polar coordinates, depending on the configuration. In addition, it defines properties to count the number of channels of each type in the configuration. These properties are used in the serialization and deserialization processes. Figure 22 presents an object diagram containing an example of a

ConfigurationFrame. The object diagram presents the structure of an instance of the classes described by the corresponding class diagram.

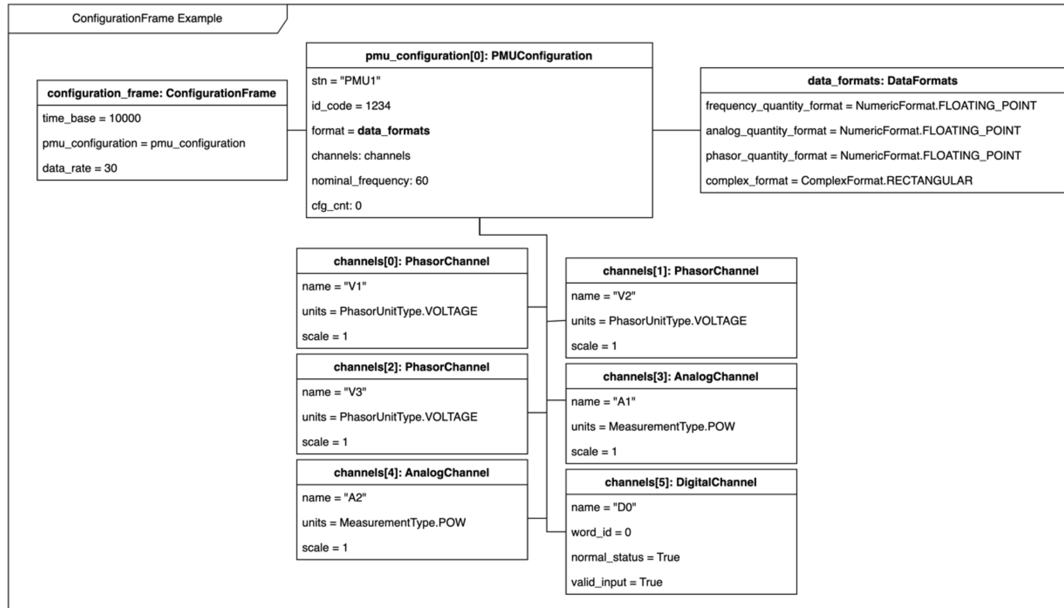


Figure 22 Configuration Frame Object Diagram

The data frame model definition is shown in Figure 23. The Synchrophasor Data class represents the data contained in each PMU, and the phasor, analog, and digital attributes contain ordered dictionaries that map channel names to parsed values. This means that the user can directly access the values contained in the data frame without having to deal with low-level deserialization details such as numeric types, scaling, etc. Phasor channels map a string key to a complex value, analog channels map a string key to a float value, and digital channels map a string key to a Boolean value. The data frame's timestamp can be accessed via the timestamp property; the timestamp is calculated using the time base defined by the configuration frame that contains the data frame configuration. Lastly, the data frame defines special methods for creating simulated data frames; these methods can be used to debug some synchrophasor data applications. However, they do not contain values that hold a physical meaning and are only correct from the point of view of the

protocol. An example of a data frame is shown in Figure 24, which corresponds to the configuration frame shown in Figure 22.

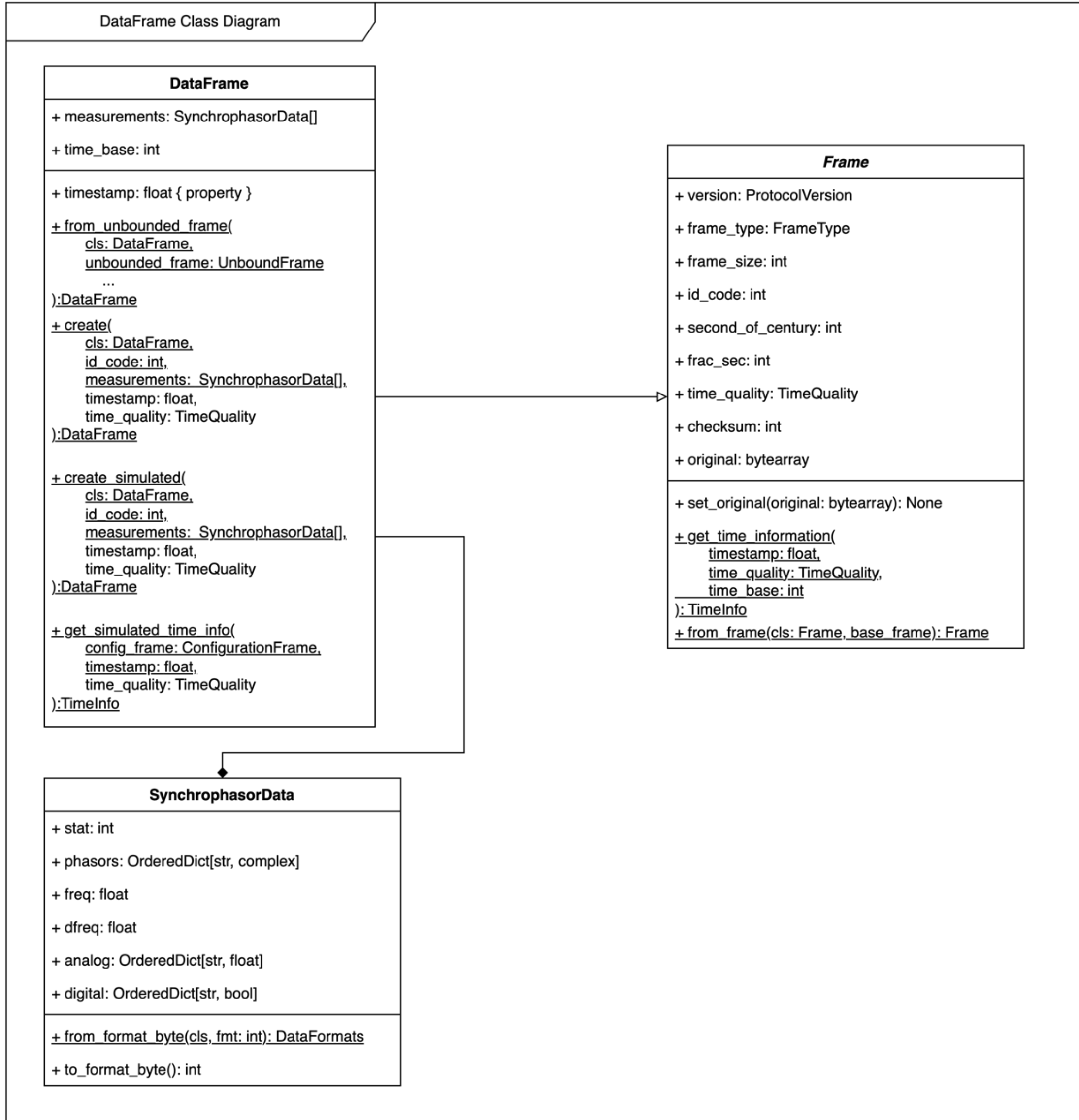


Figure 23 Data Frame Class Diagram

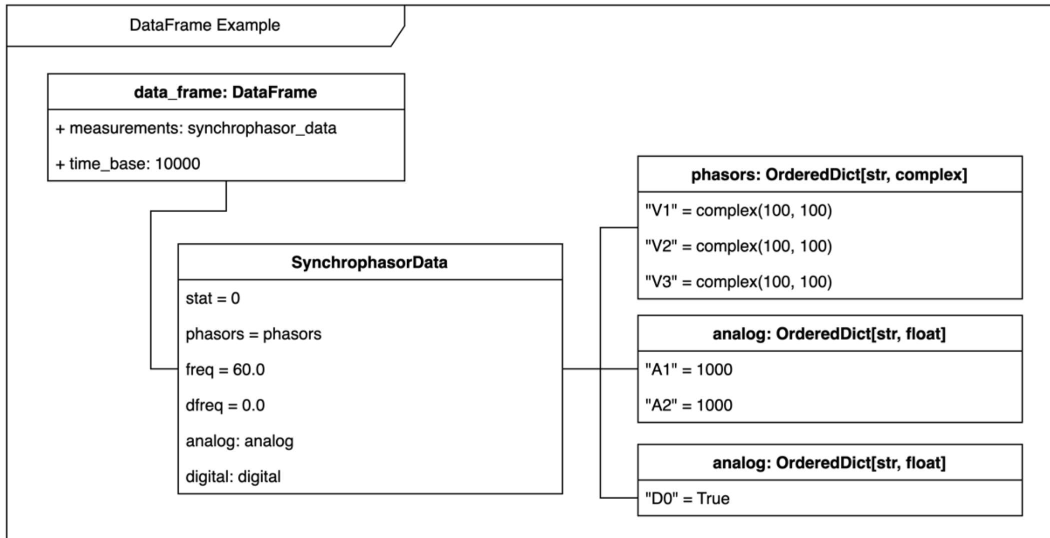


Figure 24 Data Frame Object Diagram

4.1.4 Serialization and Deserialization

Serialization refers to the process of transforming an in-memory object with a defined structure to an appropriate format for storage or data transmission, i.e. IEEE Std C37.118™-2005 byte arrays. The inverse process, transforming a byte array to a structured in-memory object, is known as deserialization.

Figure 25 shows the serialization and deserialization classes for IEEE Std C37.118™-2005 frames. These classes define a single method for performing their respective operations without knowledge of the type of frame that is passed to it. Command frames, header frames, and configuration frames can be serialized and deserialized in a stateless manner; this means that the object does not need to have its state set to perform the transformation. On the other hand, the serialization or deserialization of a data frame is, in general, a stateful operation because the transformation needs knowledge of the configuration frame that was used to create the data frame. To allow stateless manipulation of all frames, the user has the option to force unbound serialization or

deserializations; this always returns an unbound frame or performs only partial deserialization of a byte array and does not require a configuration frame to be set.

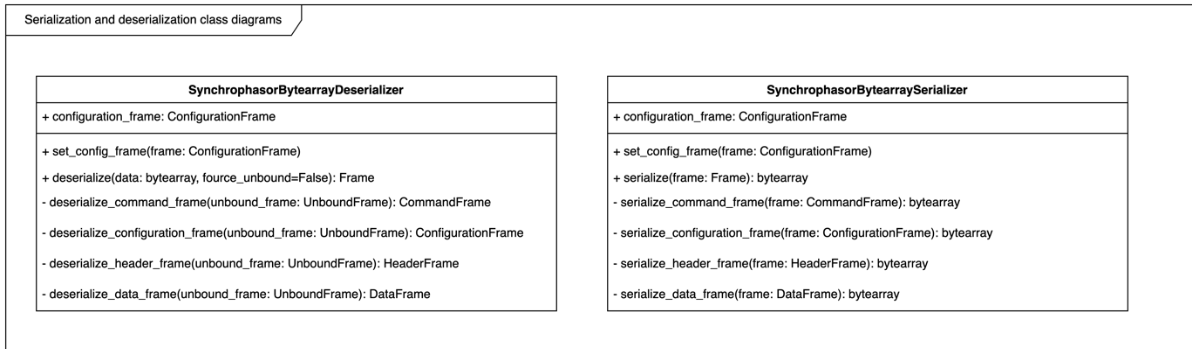


Figure 25 Serialization and Deserialization Class Diagrams

Serialization and deserialization utilities are a wrapper around the Python struct module [51]. The struct library is a low-level utility module that is part of the Python standard library that is used to serialize and deserialize variables of one of the built-in primitive data types (int, float, str, byte) in byte arrays. Figure 26 illustrates the process of packing and unpacking primitive data types into arrays of bytes. The string "i6sf" in the example defines a sequence of bytes composed of a four-byte integer represented with an "i," a twelve-byte or six-ASCII character string represented by "6s", and a 32-bit IEEE 754 floating point number represented by "f." These values are unpacked following the same principle.

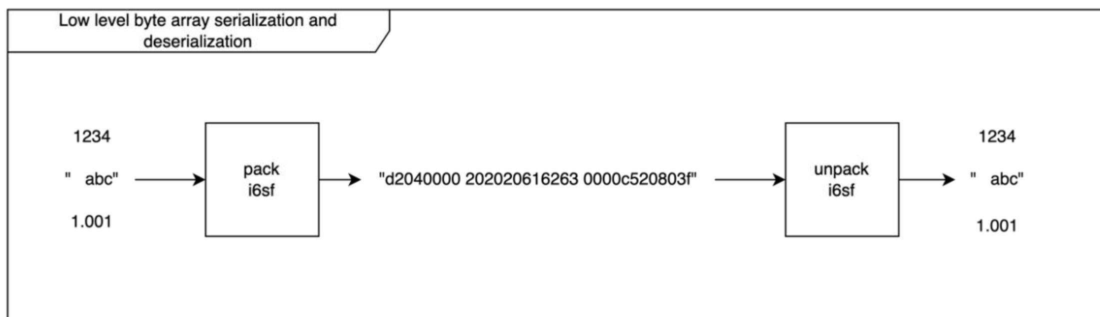


Figure 26 Struct Module Illustration

The five leading fields in a IEEE Std C37.118™-2005 frame are serialized or deserialized using the formatting string "!2BHHII," where the exclamation mark denotes big-endian endianness, and the following define the serialized frame structure as shown in Figure 27. After the leading bytes are deserialized, the size of the payload can be determined by the FRAMESIZE field, and the checksum can be deserialized.

The header frame payload can be easily deserialized or serialized by transforming it to and from a string, respectively. Similarly, the command frame has one field and an internal payload; this process is depicted in Figure 27. On the other hand, the configuration and data frames need to be dynamically serialized or deserialized, and the structure of the formatting string is determined from the data in the configuration frame and by the configuration frame itself, in the case of the data frame.

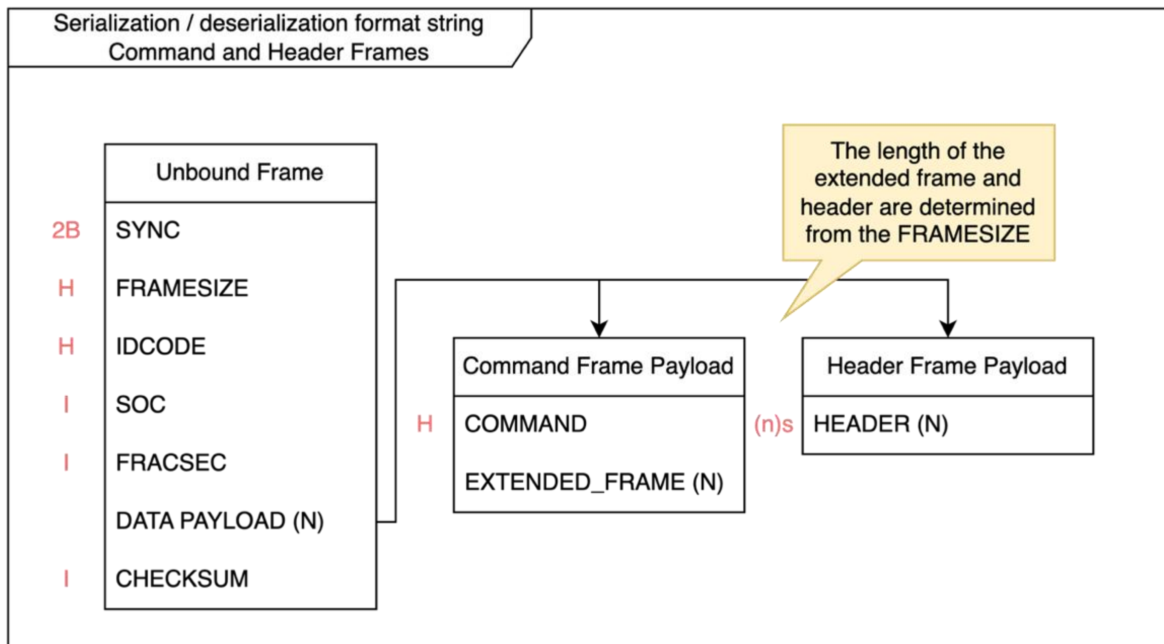


Figure 27 Command and Header Frame Serialization and Deserialization Format Strings

The serialization of a configuration frame can be simply done by creating the format string from the available information in the object and repeating the process for each synchrophasor stream configuration and each channel; this process is depicted in Figure 28. In contrast, the deserialization needs to be done partially to determine the internal structure of the stream configuration dynamically from the static frames, i.e., TIME_BASE, NUM_PMU, and FRAMESIZE. Similarly, to deserialize a single stream configuration, it is necessary to partially deserialize the leading static fields and then determine the size of the channel configuration words dynamically. In Figure 28, dynamic fields like CHNAM are depicted as elements of an array of arbitrary length.

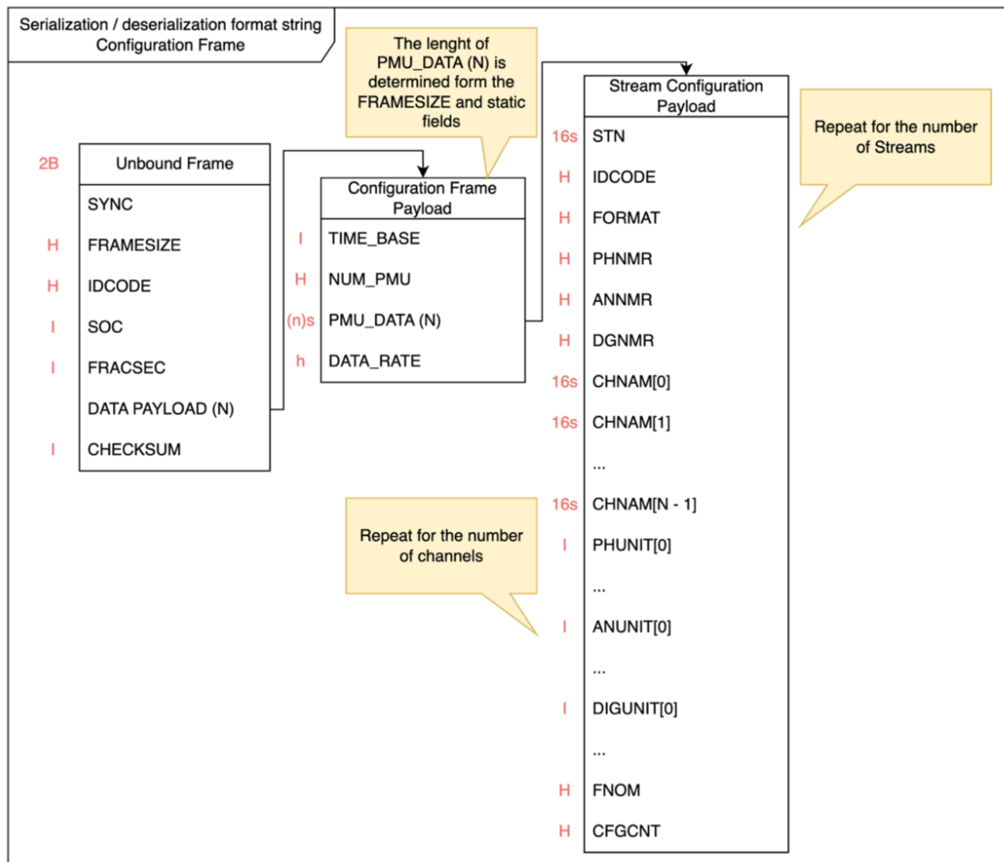


Figure 28 Configuration Frame Serialization and Deserialization Format Strings

Unlike other frame types, the only self-describing field available in data frames is the FRAMESIZE, which is helpful in extracting the payload. However, to interpret the payload or serialize a data frame, the configuration frame is needed. The format string for the data frame payloads is determined by the configuration object depicted in Figure 28. Then, it is stored in the __pack_fmt so it can be reused efficiently by multiple data frames. This process is depicted in Figure 29.

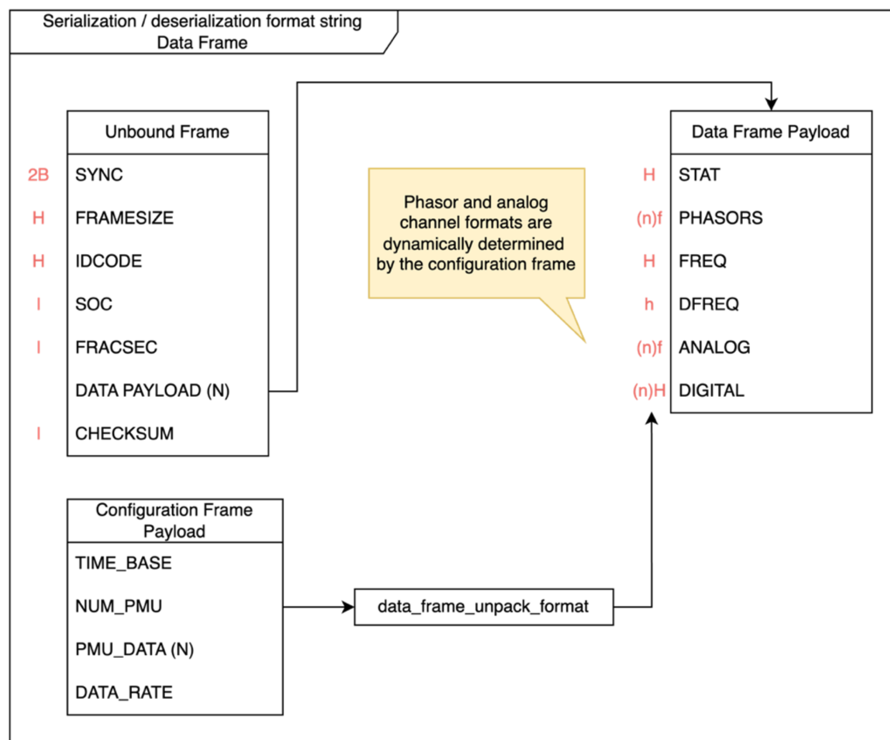


Figure 29 Data Frame Serialization and Deserialization Format Strings

4.2 Capabilities and Usage

4.2.1 Command Line Interface Tool and Usage

A command line interface tool was created to set up and use the substation agent. The command line interface tool has the following functions and can be extended to perform others depending

on the need. The main three functions of the command line tool are to create setup files to connect to a PMU or PDC, to connect to a PMU and PDC and print the received messages for debugging, to create an emulated synchrophasor data stream for testing purposes, and to connect to a PMU or PDC to

The substation agent stores the configuration to connect to a PMU or PDC in a YAML file generated from a *PMUConnection* file generated by the PMU Connection Tester [52]. YAML is a widely used, human readable format for storing structured data. A fragment of this YAML file containing configuration is shown in Figure 30, key_mapping contains an array of channels, one for each measurement in the data frame, pmu_config is an associative array containing the connection configuration.

```
key_mapping:
- channel_name: VA
  channel_type: VOLTAGE_REAL
  pmu_index: 0
  station_name: Station A
  stream_id: 63743
- channel_name: VA
  channel_type: VOLTAGE_IMAGINARY
  pmu_index: 0
  station_name: Station A
  stream_id: 971157
  …
pmu_config:
  frame_rate: 30
  id_code: 511
  interface: 127.0.0.1
  ip: 127.0.0.1
  port: 1995
```

Figure 30 YAML configuration fragment

The configuration file contains information to connect to the synchrophasor data stream and metadata added during the setup; this is shown in Table 8, row 1. The substation agent also creates random integers to identify each channel with a channel ID. To connect to a PMU or PDC, the from-agent-config command is used; this command takes two parameters: the first one is the

YAML config file, and the second one is the agent type to which the data is going to be forwarded; currently, only kinesis and printing are available. The printing agent prints each frame it receives, and the Kinesis agent transforms data frames into Protobuf messages and forwards them to a Kinesis stream. The connection command is shown in Table 8, row 2.

forward the data to a remote system such as AWS Kinesis.

	Command
1	poetry run python -m substation_agent create-pmu-config ./local.PmuConnection ./config1.yaml
2	poetry run python -m substation_agent from-agent-config config1.yaml --agent-type=kinesis

Table 8 Command Line Interface Tool Commands to Connect to a PMU

4.2.2 Managing configuration changes in PDC

In an operating substation, it is often the case that new PMUs are installed or new channels are added to the existing PMUs which causes a new configuration frame to be emitted by the PDC. Some provisions were made in the library to handle configuration frame changes. The first and most important is that the base abstract class of the library was implemented with abstract methods to be overwritten by each implementation to do what is deemed to be appropriate for a given context. This can be seen in Figure 31.

The default implementation used for the kinesis type agent updates the frame deserializer and assigns a new random ID to the new channels as shown in Figure 32 Existing channels are kept with the original ID.

```python
class SubstationAgent:
    def __init__(
            self, host: str, port: int, id_code: int, recv_timeout=0.5,
            local_addr: Optional[str] = None, fix_fracsec: bool = False,):
        self.max_missed_frames = 5
        self.max_restarts = 5
        self.host = host
        self.port = port
        self.id_code = id_code
        self.recv_timeout = recv_timeout
        self._local_addr = local_addr
        self.fix_fracsec = fix_fracsec

    def create_client(self, host, port, id_code, recv_timeout=0.5):⋯

    async def initialize_connection(self):⋯

    async def on_config_frame_1(self, message: ConfigurationFrame):⋯

    async def on_config_frame_2(self, message: ConfigurationFrame):⋯

    async def message_handler(self, message: Frame):⋯

    async def receive_messages(self):⋯

    async def run(self):⋯

    async def disconnect(self):⋯

    @classmethod
    def run_from_args(⋯
```

Figure 31 Abstract Substation Agent Event Handlers

```python
    async def forward_message(self, message: Frame):
        if message.frame_type == FrameType.CFG_FRAME_2:
            config_pb = configuration_frame_to_protobuf_messages(
                cast(ConfigurationFrame, message), stream_ids=self.key_mapping,
                three_phase_groups=self.three_phase_groups)
            self.channel_metadata = config_pb
            self._forwarding_client_serializer.set_config_frame(
                cast(ConfigurationFrame, message)
            )
            asyncio.create_task(self.send_config(config_pb))

        elif message.frame_type == FrameType.DATA_FRAME:
            data_frame = cast(DataFrame, message)
            ts_ms = int(data_frame.timestamp * 1000)
            self.received_timestamps.append(ts_ms)
            df_messages = data_frame_to_protobuf_message(
                data_frame,
                self.key_mapping
            )
            for b in df_messages:
                self.in_batch_timestamps.append(b.timestamp)
            self.batch.extend(df_messages)
            # print("Len batches: ", len(self.batch))
```

Figure 32 Configuration frame update

4.3 Testing and Validation

4.3.1 Unit Testing

Unit tests were used to implement the serialization and deserialization of IEEE Std C37.118™-2005 frames, which allowed reproducible tests to be conducted during development. In addition, random tests were conducted to create frames that were serialized to an array of bytes and then deserialized back. The rationale behind performing multiple random tests is that using the data frame models makes it straightforward to create random instances of configuration and data frames. Random frames are then serialized, deserialized, and compared field by field to ensure no functionality is missing.

The program described in 4.1 is organized in two projects, a library containing the IEEE Std C37.118™-2005 data models, serialization, deserialization and abstract client and a concrete implementation of the substation agent that uses the IEEE Std C37.118™-2005 library. Figure 33 shows the coverage report of the tests conducted on the IEEE Std C37.118™-2005 library. A coverage report quantifies the execution paths that a group of testing functions has tested. The coverage is the proportion of executed execution paths to the available execution paths in a software source. Lines highlighted in gray show the files that correspond to serialization, deserialization and data modelling. The combination of random testing, unit testing and coverage assessment helps to ensure that at least frames serialized by the library can be correctly deserialized again without data loss.

```
Name                                                    Stmts   Miss  Cover
---------------------------------------------------------------------------
c37118/__init__.py                                          0      0   100%
c37118/deserializers/__init__.py                            0      0   100%
c37118/deserializers/bytearray.py                           0      0   100%
c37118/deserializers/bytearray/__init__.py                 48      7    85%
c37118/deserializers/bytearray/command_frame.py            11      1    91%
c37118/deserializers/bytearray/configuration_frame.py      44      1    98%
c37118/deserializers/bytearray/data_frame.py               60      5    92%
c37118/deserializers/bytearray/header_frame.py              9      1    89%
c37118/deserializers/bytearray/unbounded_frame.py          14      1    93%
c37118/models/__init__.py                                   9      0   100%
c37118/models/command_frame.py                             18      0   100%
c37118/models/configuration_frame.py                      117      0   100%
c37118/models/data_frame.py                                55     16    71%
c37118/models/enums.py                                    246      0   100%
c37118/models/frame.py                                     43      0   100%
c37118/models/header_frame.py                              17      0   100%
c37118/models/unbounded_frame.py                            5      0   100%
c37118/pb.py                                              125    125     0%
c37118/schemas/__init__.py                                  0      0   100%
c37118/schemas/synchrophasor_pb2.py                        39     39     0%
c37118/serializers/__init__.py                              0      0   100%
c37118/serializers/bytearray/__init__.py                   30      0   100%
c37118/serializers/bytearray/command_frame.py              12      0   100%
c37118/serializers/bytearray/common.py                      9      0   100%
c37118/serializers/bytearray/configuration_frame.py        44      0   100%
c37118/serializers/bytearray/data_frame.py                 62      0   100%
c37118/serializers/bytearray/header_frame.py               11      0   100%
c37118/serializers/json/__init__.py                         8      8     0%
c37118/serializers/protobuf.py                             19     19     0%
c37118/synchrophasor_client.py                            116    116     0%
c37118/utils.py                                            13      0   100%
---------------------------------------------------------------------------
TOTAL                                                    1184    339    71%
```

Figure 33 Unit Tests Coverage

4.3.2 Validation and Performance Testing

Multiple tests were conducted to validate the operation of the synchrophasor agent. For this end, physical PMU of various vendors, RTDS simulated PMU and pyPMU [53-54] simulated PMU were used.

An initial test was done using the printing agent to evaluate the overall performance of the IEEE Std C37.118™-2005 client. This was of particular importance to test the deserialization capabilities and to test the connection to physical devices. Figure 34 shows the example output of the printing substation agent connected to a virtual PMU realized using pyPMU [53-54]. The values in the response were compared to the fixed values set in the virtual PMU to verify the proper operation.

Furthermore, the output from the printing agent type can be compared with other data sources such as a captured network stream of packets to verify it against physical devices.

```
poetry run python -m substation_agent from-agent-config config1.yaml --agent-type=printing

DataFrame(version=<ProtocolVersion.IEEE_C37_118_2005: 1>, frame_type=<FrameType.DATA_FRAME: 0>, frame_size=52, id_code=511, second_of_century=1728017114,
frac_sec=966657, time_quality=TimeQuality(leap_second_flag=<LeapSecondFlags: 0>, time_quality_indicator=<TimeQualityIndicator.NORMAL_OPERATION: 0>), checksum=18341,
measurements=[SynchrophasorData(stat=0, phasors=OrderedDict([('VA', (133987.37645+0j)), ('VB', (-66998.26586-116052.20252j)), ('VC', (-66998.26586+116043.04725j)),
('I1', (499.87392000000006+0j))]), freq=62.5, dfreq=0.0, analog=OrderedDict([('ANALOG1', 100.0), ('ANALOG2', 1000.0), ('ANALOG3', 10000.0)]),
digital=OrderedDict([('BREAKER 1 STATUS', False), ('BREAKER 2 STATUS', True), ('BREAKER 3 STATUS', False), ('BREAKER 4 STATUS', False), ('BREAKER 5 STATUS', True),
('BREAKER 6 STATUS', False), ('BREAKER 7 STATUS', False), ('BREAKER 8 STATUS', False), ('BREAKER 9 STATUS', False), ('BREAKER A STATUS', False), ('BREAKER B STATUS',
True), ('BREAKER C STATUS', True), ('BREAKER D STATUS', True), ('BREAKER E STATUS', True), ('BREAKER F STATUS', False), ('BREAKER G STATUS', False)]))], time_base=1000000)
1728017114967.0 66.26105308532715

DataFrame(version=<ProtocolVersion.IEEE_C37_118_2005: 1>, frame_type=<FrameType.DATA_FRAME: 0>, frame_size=52, id_code=511, second_of_century=1728017115, frac_sec=0,
time_quality=TimeQuality(leap_second_flag=<LeapSecondFlags: 0>, time_quality_indicator=<TimeQualityIndicator.NORMAL_OPERATION: 0>), checksum=46921,
measurements=[SynchrophasorData(stat=0, phasors=OrderedDict([('VA', (133987.37645+0j)), ('VB', (-66998.26586-116052.20252j)), ('VC', (-66998.26586+116043.04725j)),
('I1', (499.87392000000006+0j))]), freq=62.5, dfreq=0.0, analog=OrderedDict([('ANALOG1', 100.0), ('ANALOG2', 1000.0), ('ANALOG3', 10000.0)]),
digital=OrderedDict([('BREAKER 1 STATUS', False), ('BREAKER 2 STATUS', True), ('BREAKER 3 STATUS', False), ('BREAKER 4 STATUS', False), ('BREAKER 5 STATUS', True),
('BREAKER 6 STATUS', False), ('BREAKER 7 STATUS', False), ('BREAKER 8 STATUS', False), ('BREAKER 9 STATUS', False), ('BREAKER A STATUS', False), ('BREAKER B STATUS',
True), ('BREAKER C STATUS', True), ('BREAKER D STATUS', True), ('BREAKER E STATUS', True), ('BREAKER F STATUS', False), ('BREAKER G STATUS', False)]))], time_base=1000000)
1728017115000.0 70.45197486877441

DataFrame(version=<ProtocolVersion.IEEE_C37_118_2005: 1>, frame_type=<FrameType.DATA_FRAME: 0>, frame_size=52, id_code=511, second_of_century=1728017115,
frac_sec=66666, time_quality=TimeQuality(leap_second_flag=<LeapSecondFlags: 0>, time_quality_indicator=<TimeQualityIndicator.NORMAL_OPERATION: 0>), checksum=10884,
measurements=[SynchrophasorData(stat=0, phasors=OrderedDict([('VA', (133987.37645+0j)), ('VB', (-66998.26586-116052.20252j)), ('VC', (-66998.26586+116043.04725j)),
('I1', (499.87392000000006+0j))]), freq=62.5, dfreq=0.0, analog=OrderedDict([('ANALOG1', 100.0), ('ANALOG2', 1000.0), ('ANALOG3', 10000.0)]),
digital=OrderedDict([('BREAKER 1 STATUS', False), ('BREAKER 2 STATUS', True), ('BREAKER 3 STATUS', False), ('BREAKER 4 STATUS', False), ('BREAKER 5 STATUS', True),
('BREAKER 6 STATUS', False), ('BREAKER 7 STATUS', False), ('BREAKER 8 STATUS', False), ('BREAKER 9 STATUS', False), ('BREAKER A STATUS', False), ('BREAKER B STATUS',
True), ('BREAKER C STATUS', True), ('BREAKER D STATUS', True), ('BREAKER E STATUS', True), ('BREAKER F STATUS', False), ('BREAKER G STATUS', False)]))], time_base=1000000)
1728017115067.0 42.611122131347656
```

Figure 34 Substation Agent Testing and Validation Output

In addition, four tests at different data rates were conducted to assess the time performance of the IEEE Std C37.118™-2005 client. The data rates used in for the tests were 10, 30 and 60 fps and the 30 fps test was conducted twice to verify that the same results were obtained. Each test had a duration of 30 to 60 minutes depending on the data rate to gather enough data to see the behaviour of the agent in time. During the experiment the timestamps were recorded to compare the time at which the record was measured and the time at which it was received by the agent. Table 9 summarizes the results. These suggest a correct operation because the number of received frames corresponds to the expected number of frames at the given data rate used for the experiment.

Measure	**Values**			
Test number	1	2	3	4
Data rate [frames/s]	60	30	30	10
Time difference between the last and the first frame timestamps [minutes]	31.66638	33.3327	31.4438	66.665

CPU measured time difference between ingestion of the last and the first frame [minutes]	31.6667	33.3327	31.4438	66.6649
Number of records processed [frames]	114000	60000	56600	40000

Table 9 Substation Agent Validation Measurements

As the function of the client is to forward frames from individual PMU to the cloud, one of the most significant issues that it could experience would be the queuing of frames due to an inability to keep up with the rate at which the frames are produced. To assess this, the measured time difference between consecutive frames is calculated; the difference is normalized to the expected period (100 ms, 33.33 ms, and 16.67 ms) to compare between reporting rates of 20, 30 and 60 frames/s. The arrival period at which frames are produced is constant and known. Moreover, the difference between the GPS timestamps in the frames is constant. The difference between the measured period and the expected period is used to assess the behaviour of the software. Equation (2), where $t_{arrival}$ represents the arrival time measured by the monotonic clock of the CPU at a given instance, shows the measured period and (3), where the ϵ represents the error and r the data rate e.g. 30 frames/second, the error of the measured period with respect to the expected period. Equation (4) represents the percent error of the measured arrival period.

$$\Delta T = t_{arrival,i} - t_{arrival,i-1} \tag{2}$$

$$\epsilon = \Delta \mathrm{T} - \frac{1}{r} \tag{3}$$

$$\delta = \frac{\left(\Delta \mathrm{T} - \frac{1}{r}\right)}{\frac{1}{r}} \tag{4}$$

Figure 35 shows the probability density of the percent of error of the measured arrival period. The probability density plots show that the deviation of the measured time difference between frames is less than 10% with respect to the expected arrival period. However, the width of the distribution grows with the data rate; this shows that, at higher rates, it is more challenging for the agent and the communication channel to keep up with the producer.

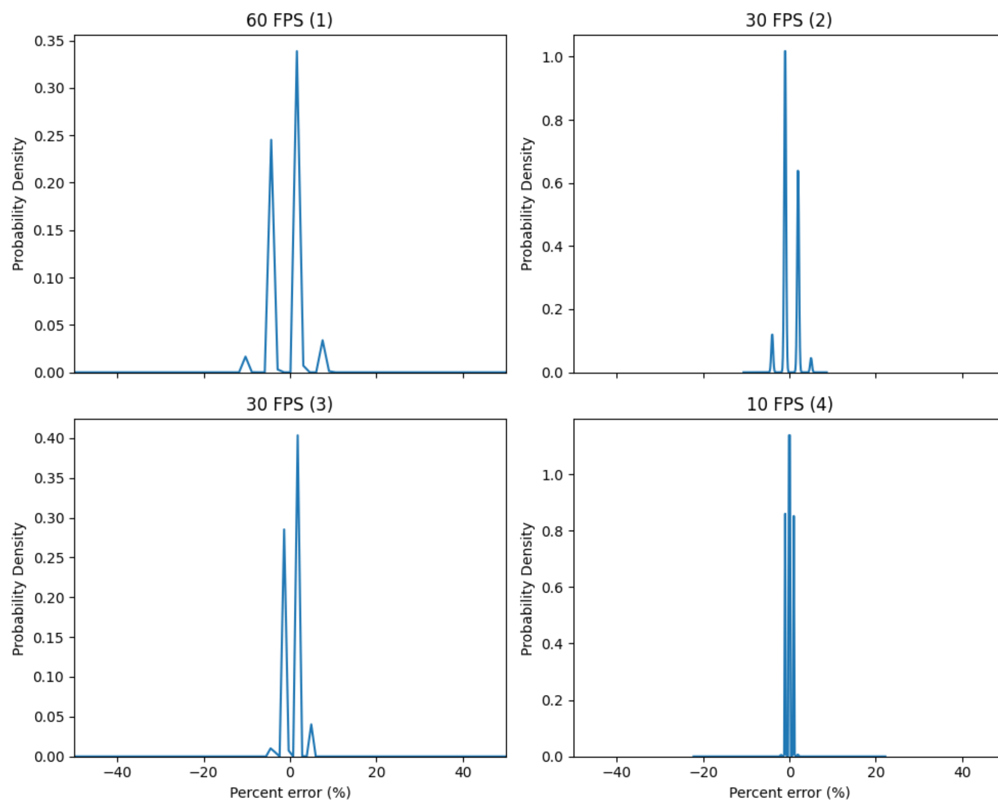


Figure 35 Percentage of Error in the Time Between Frames and the Expected Period

Depending on the data rate, the time difference between consecutive frames is expected to be 100 ms, 33.33 ms, or 16.6667 ms. The measured period error is zero on average, to get a better insight into how much it drifts, cumulative error is calculated by summing the error over a long period of time. Equation (5) represents the cumulative error of the measured period.

$$\Delta E = \sum_{i=0}^{n-1} \Delta\epsilon_i \tag{5}$$

Figure 36 shows the cumulative sum of measured time differences between successive frames. If no queuing is occurring, meaning that the consumer can keep up with the producer data rate, the graphs are expected to grow unbounded. A slight decrease in the magnitude of the accumulated time difference is observed; around 5 or 6 ms every 30 minutes. This would suggest that the rate at which frames are received increases with time which is not likely. Instead, a more plausible explanation is that this can be attributed to the drifting of the computer clock. A typical CPU clock has an accuracy of around 15 ppm which would account for a drift of 27 ms every 30 min [55]. It is likely that this perceived drift would be eliminated by synchronizing the computer where the agent runs with a satellite clock using a suitable method, for example using precision time protocol.

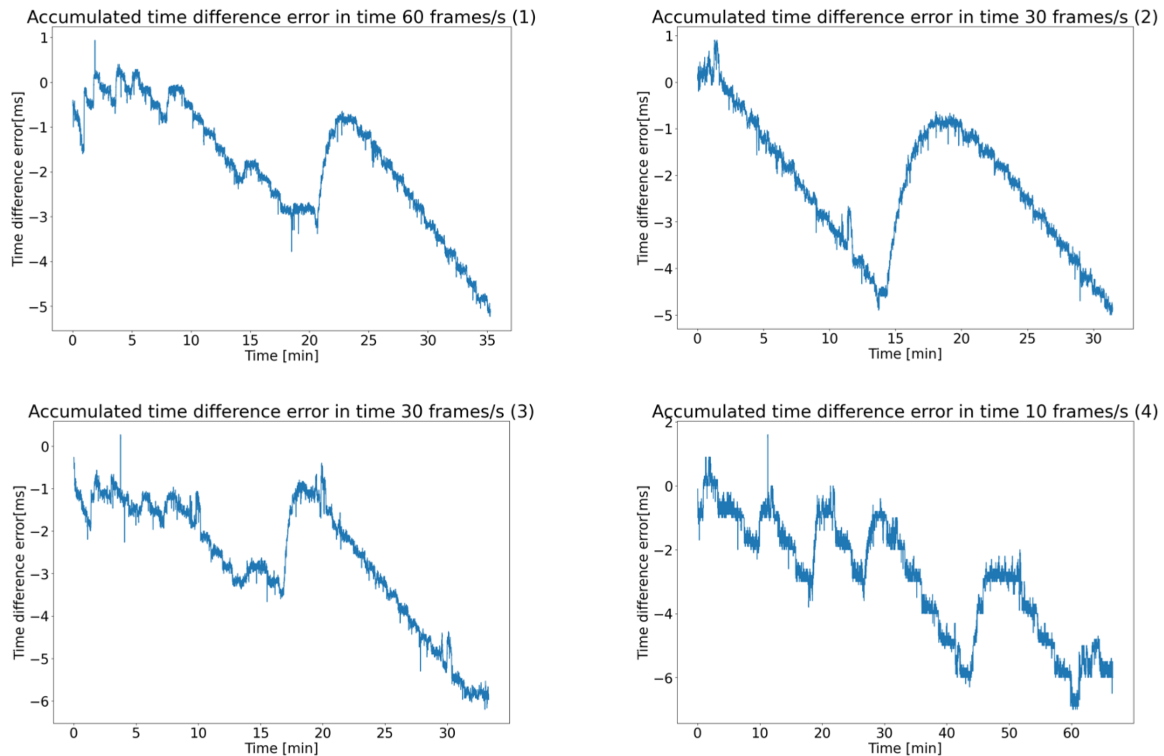


Figure 36 Accumulated Average Time Difference

4.4 Summary

The design of the substation agent, the software aimed to extract data from the PMU/PDC and forward it to the cloud, is presented in detail. The substation agent is developed as a modular library that contains the data models, serialization and deserialization capabilities needed to interact with IEEE Std C37.118™-2005 devices and an implementation of the IEEE Std C37.118™-2005 client in a command line utility that is the main piece of software aimed to extract and forward the synchrophasor data to a cloud message broker. Finally, a description of how the agent was validated is provided. The tests show the agent is functioning as intended and that there is little to no latency added.

Chapter 5 Cloud Infrastructure

This section describes the cloud infrastructure used for ingesting and processing synchrophasor data and the technique used for its implementation. The cloud infrastructure comprises a data ingestion system, a distributed stream processing platform, a database for storing synchrophasor metadata, computing resources to access the metadata database, and the underlying networking components.

In cloud-based applications, particularly during the development stages, it is helpful to have the means for reproducing the cloud infrastructure consistently through multiple iterations and on demand. This helps to ensure the consistency of the environment where the application runs and helps to reduce costs as the resources are only created when needed. This is achieved by the use of a technique known as infrastructure as code or IaC. IaC consists of using a cloud infrastructure description language to set up the necessary resources for the operation of the software. This project uses AWS CloudFormation™ to describe the cloud resources.

5.1 Challenges integrating synchrophasor protocols with the cloud

The main challenge of integrating synchrophasor protocols such as GEP, STTP or any version for C37.118 remains to be the clock synchronization as there is no open literature on how to synchronize cloud servers with an external time source such as a GPS clock in the way required by the protocols. Moreover, while the Precision Time Protocol (PTP) can be used to synchronize servers inside the cloud, there are no standard ways of synchronizing network servers with a satellite clock source via PTP over long distances. The second challenge is that STTP does not define how it should work in a distributed environments.

5.2 Cloud Infrastructure Resources

Table 10 presents the primary cloud resources used; lower-level resources, such as networking elements, are omitted for brevity. Two groups of resources can be distinguished: the ones used to host and run the application and the ones used to automate the deployment of code updates. The latter is needed because deploying software to the cloud involves a series of repetitive steps that are prone to error and that are often automated to achieve a repeatable build process.

Resource	**Role**
AWS Kinesis Streams™	Data stream
	Metadata stream
	Output stream
	Latency measurement stream
Amazon Managed Service for Flink™	Stream processing platform
AWS Relational Database Service (RDS)™	Postgres metadata database
AWS Elastic Container Service (ECS)™	Cloud Containers for the interface with the database
AWS Application Load Balancer™	Interface between services and internet
AWS Elastic Container Registry (ECR)™	Elastic container registry (ECR) for storing the built application image
AWS Simple Storage Service(S3)™	S3 Bucket for storing the build Java Flink application
AWS CodeCommit™	Tools and resources for building and deploying the application
AWS CodeBuild™	
AWS CodePipeline™	

Table 10 Provisioned Cloud Resources

To implement this stream processing architecture, Apache Flink was considered instead of other stream processing applications because it implements an event-time notion, provides true stream processing capabilities and was expressive enough to create synchrophasor applications. As described earlier, in the context of stream processing, the time at which the source emits an event or measurement is known as event time. In contrast, the time at which an event is observed is known as processing time. Unlike previous stream processing engines, Apache Flink distinguishes between event-time and processing time and therefore highly advantages for synchrophasor data processing [31] [37].

5.3 Cloud Application Design

Figure 37 shows the architecture diagram of the cloud application. Users access the metadata database through the load balancer (1) in the public network; this routes the traffic to an Amazon ECS™ deployment connected to a Postgres™ database (2) with a centralized metadata registry. The processing pipeline is hosted in the same private network as the Amazon ECS™ and service; this helps to secure access to the service, allowing it to run privately. Data from the substation agent (8) is ingested through Kinesis Streams™ (3), a stream for data and another for metadata. A processing pipeline (4) built with Apache Flink using Amazon Managed Service for Flink™ consumes the data and metadata from the Kinesis Streams™. After processing, the data pipeline (4) can write the output to an S3™ bucket (6) for long-term storage or back into Kinesis Streams™ where the results can be made available for other applications.

Lastly, a continuous integration and continuous deployment (CI/CD) pipeline was built to update the metadata registry and the processing pipeline whenever changes are pushed to the software repository. Git is a widely used version control system for software projects, a git repository is

used to store the program in the cloud using AWS CodeCommit. An event that is triggered by pushing changes to the repository triggers the CI/CD mechanism initiating the redeployment of the software. While a CI/CD pipeline is not strictly necessary for the application to work, it optimizes and improves the development process.

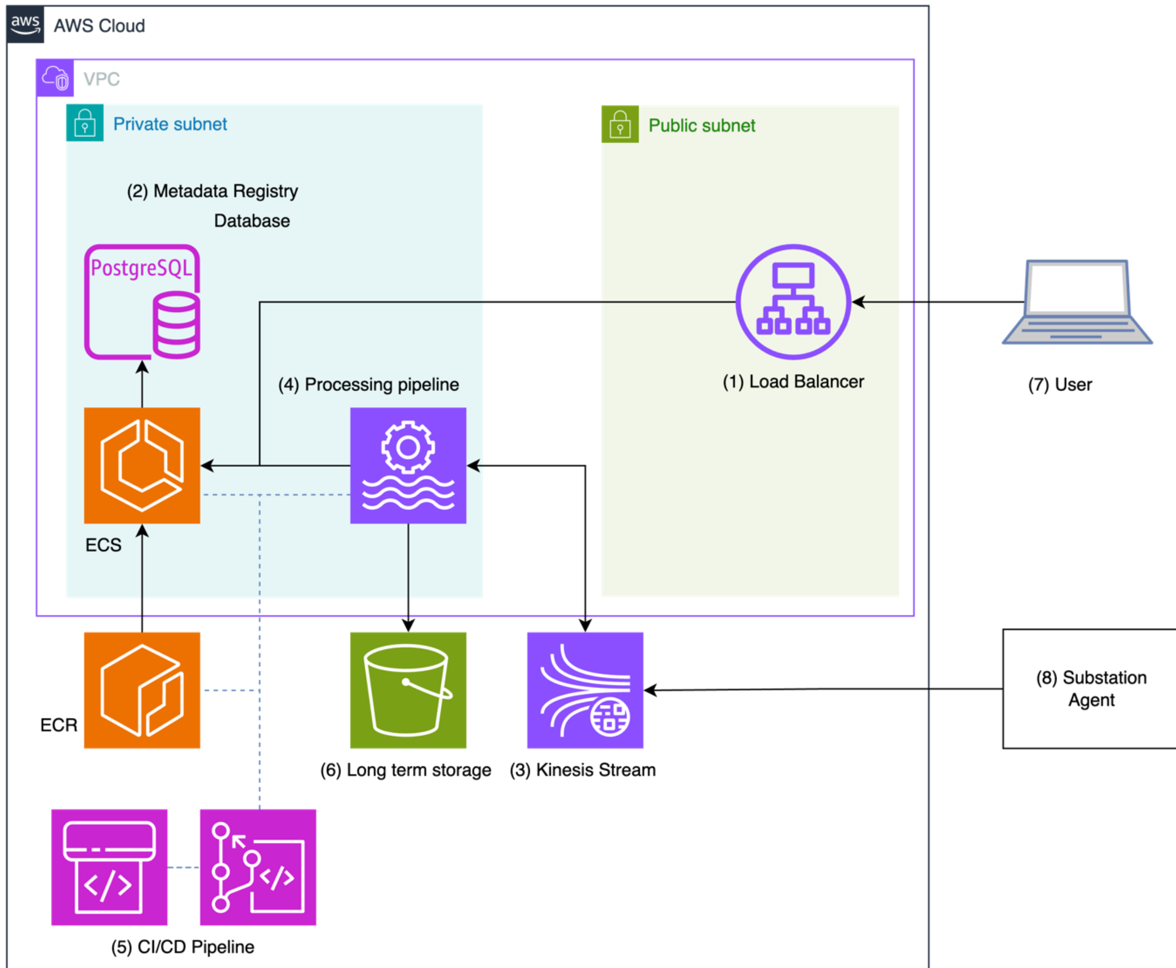


Figure 37 Cloud Application Architecture

5.4 Database Schema

The database stores the metadata and operation definitions for the processing pipeline. Currently, only operation definitions are sent to the processing pipelines as the channel metadata is not strictly necessary during the processing of the measurements.

Figure 38 shows the database schema. Every table is shown with its structure and attributes. A relationship between two tables is represented by a line and a Crow's foot, this notation means that zero or more rows of the table on the Crow's foot side are related to a single row on the table on the other side via a foreign key relationship by referencing the row id on every related row on the table on the crow's foot side. The processing pipeline sends the channel metadata to the database, and the web service that receives it creates or updates Phasor and ThreePhasorGroup models if the information is available.

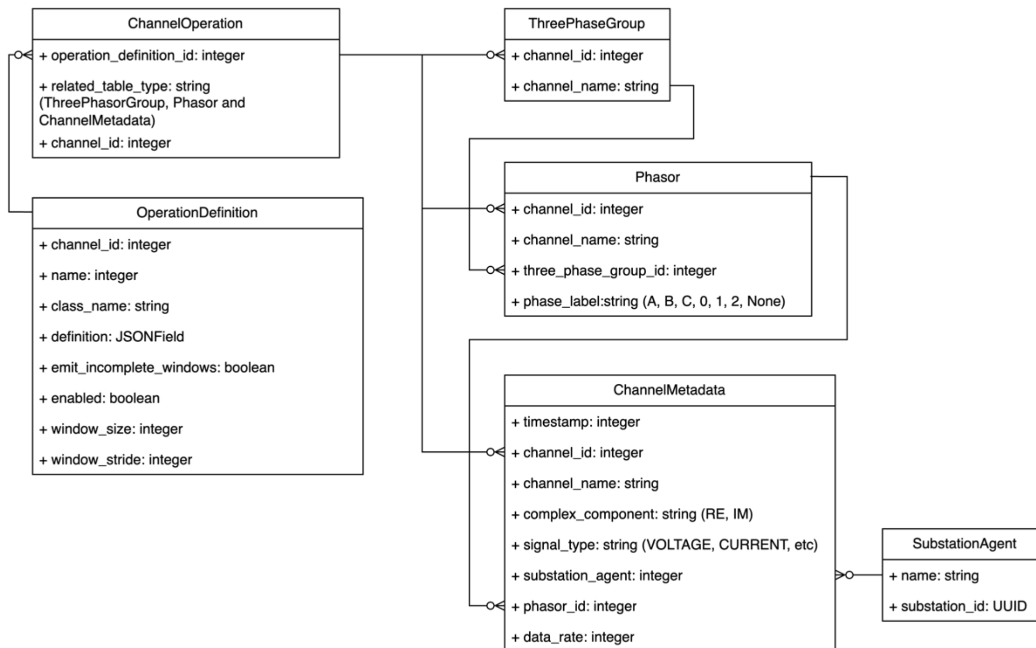


Figure 38 Database Schema

When the information is not available, the user can set it manually through the web interface. Channels are associated with a substation agent using the SubstationAgent model, where additional metadata such as geographical coordinates can be potentially added in the future. The operation definition (OperationDefinition) is defined manually and contains a collection of channels to which an operation is applied using the processing pipeline. The ChannelOperation model relates an operation definition to channels whose metadata is available.

5.5 Summary

This section presents a recount of the cloud infrastructure resources that were used in the implementation of the synchrophasor data system. The relation between different components and their objectives are presented. Furthermore, the internal structure of the metadata and operations registry is presented. Lastly, the cloud infrastructure is laid out around the Apache Flink Cluster which enables the interoperation with the other cloud resources. A detailed description of the processing software in the Apache Flink Cluster is introduced in Chapter 6.

Chapter 6 Data Pipeline Design

This chapter presents the design of the reconfigurable data pipeline for near-real-time synchrophasor applications. The objective is to study how to build synchrophasor applications from data that is stored in a partitioned system. To achieve this, a bottom-up approach is used to build a system where features are progressively added as problems arise in an effort to present a comprehensive exploration of the techniques involved.

The data pipeline is in charge of retrieving measurements that are stored in a partitioned system, which means that measurements are physically located in multiple servers. Stream processing frameworks are used in this context, as they implement tools and algorithms to perform this task correctly.

A synchrophasor application is a signal-processing operator defined on a fixed set of inputs of a defined length. The first task of the pipeline is to efficiently arrange the input data that is passed to the processing operator. The second major task is to instantiate processing operators to be applied to the input data. The following are the design objectives:

- The synchrophasor pipeline consumes data from a distributed message queue where records are received in real-time as the agent produces them.
- The pipeline must be able to host multiple operations and apply them to different sets of records.
- The pipeline consumes operation definition records from a slowly changing source and applies this configuration through the pipeline

- Operation chaining must be considered in the design to minimize the number of network transactions that occur so as to reduce the end-to-end latency.

This chapter begins with an exploration of stream processing methodologies and builds up the necessary concepts to design the processing pipeline. Following, the data models and pipeline design are presented and described in detail. Lastly, the latency and performance measurements of the data pipeline are presented.

6.1 Exploration and Design

The first approach to the problem consists of a stream processing job for a single operation. This serves as a design baseline. The general structure of the application involves taking a set of channels that may be initially in different partitions and using a time window to gather them and process them together.

Figure 39 Shows how the records are transformed in the initial approach. A combination of a timestamp and a channel ID represents a record, each column shows a snapshot of a number of records in the stream after a transformation is applied, and arrows represent partitions or parallel data flows. The first step shows a collection of records, then only channels 2 and 3 are filtered, the next step creates groups of data over many timestamps using a sliding window, and lastly, the processing operation is applied to the group of records. A processing function that counts the number of inputs and prints the data received was created to demonstrate the process.

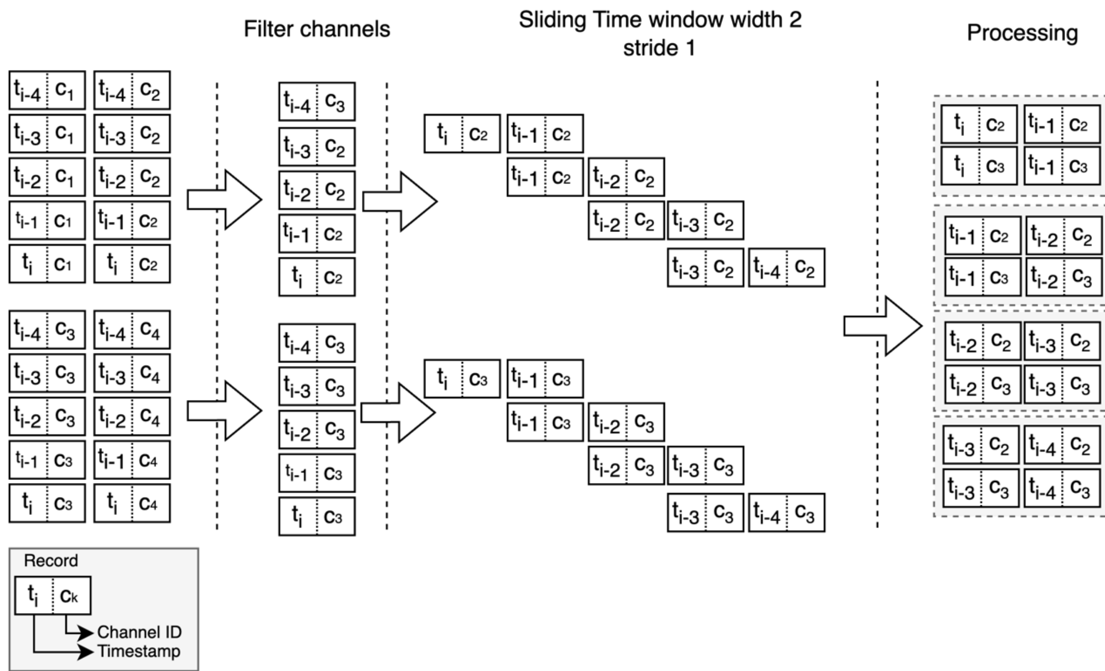


Figure 39 Simple Processing Job

The code snippet and execution plan of this simple processing job are shown in Figure 40. This creates two sets of chained operations and a single point at which records are exchanged in the network. This happens from the filter to the windowing operator. While this simple approach is the most flexible as it potentially allows any type of processing, it has the disadvantage that is can only represent a single operation on the data and it cannot be reconfigured.

```
dataStream
  .filter(x -> x.getChannelId() == 2L || x.getChannelId() == 3L)
  .windowAll(
    SlidingEventTimeWindows.of(
      Duration.ofMillis(64),
      Duration.ofMillis(34))
    )
  .process(…);
```

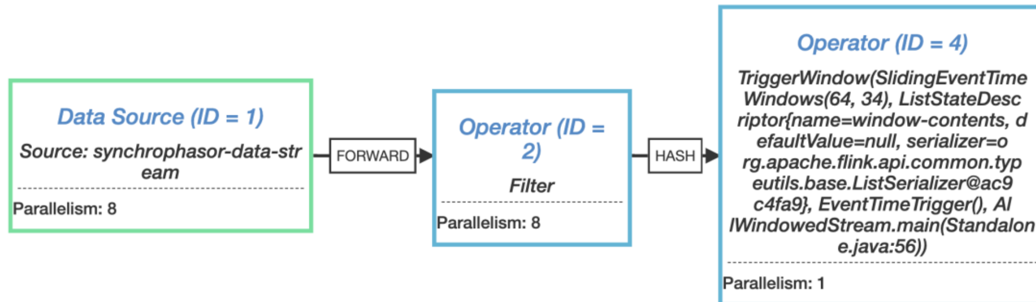


Figure 40 Simple Processing Job Snippet and Execution Plan

The next step consists of generalizing the filter operation to allow multiple operations. Figure 41 shows the modified pipeline. This includes a connection to a new stream that carries operation definitions and a rekey step that tags records with their corresponding event and routes them to a partition to be processed in parallel. The windowing and processing steps remain unchanged.

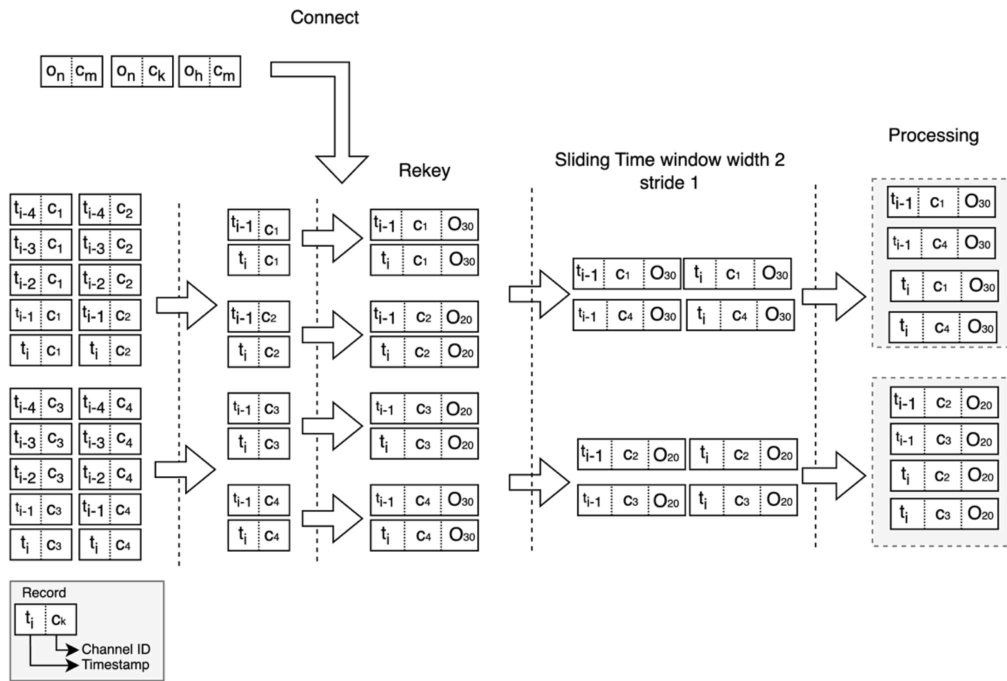


Figure 41 Simple Processing Job with Operation Stream

Figure 42 shows the snippet for the modified processing job and the execution plan. The connection to the broadcast stream modifies the nature of the connection from a forward type of connection to a hash type when the filter is replaced, this adds two network shuffles to the process.

Window parameters and lengths are application dependent. For this reason, the processing application must support keyed time windows. Keyed time windows, windows whose parameters depend on a key are not available in Apache Flink off-the-shelf. To achieve this functionality, window parameters are added to the event in the enrichment process when the operation ID is added. A modified version of the event-time sliding window is implemented, and the window parameters, such as length and stride, are determined from the event. This implementation does not impact the execution plan.

```
dataStream
  .keyBy(Event::getChannelId)
  .connect(broadcastedChannelOperationMapping)
  .process(new ReKeyByOperationId())
  .keyBy(Event::getOperationId)
  .window(
    SlidingEventTimeWindows.of(
      Duration.ofMillis(64),
      Duration.ofMillis(34)
    )
  )
  .process(…);
```

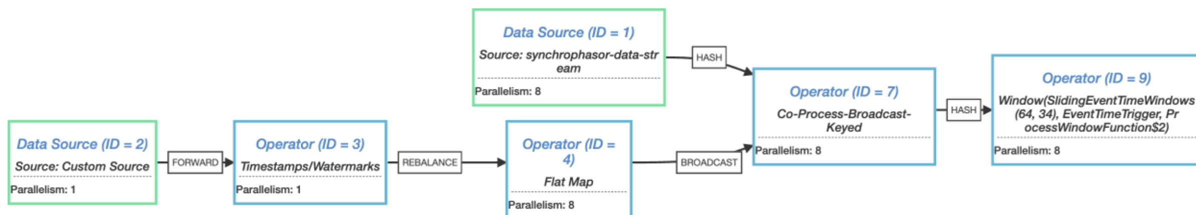


Figure 42 Simple Processing Job with Operation Stream Snippet and Execution Plan

6.2 Data Model

The data arrives in Protobuf-encoded messages from the Amazon Kinesis Stream™, which are small batches assembled and sent by the substation agent. These are converted into an efficient data structure natively supported by Kinesis called a Tuple9. A Tuple9, as its name suggests, is an immutable collection of 9 elements of predefined types. Apache Flink implements efficient mechanisms for tuples that are not available for other forms of event serialization. A higher-level interface is defined by extending the Tuple9 class to represent incoming measurements uniquely.

Figure 43 presents the Event data model. An event is a single measurement received from the substation agent, e.g., voltage magnitude, voltage angle, frequency, etc. Instead of defining variables as it is commonly done in a class definition, this implementation is a wrapper around

Tuple9. This allows it to leverage Apache Flink's efficient built-in serialization/deserialization capabilities for tuples while providing a high-level interface. Benchmarking of different Apache Flink serialization mechanisms has shown that the native serialization of tuples leads to a higher throughput [56].

An operation is related to incoming events through the field inputChannelIds; this relation is depicted in Figure 43. In this way, the data pipeline tags and forwards events to the partition that applies the operation. In each partition, a sliding window is used to group data from all the channels in an operation. Finally, the operation IDs are also used to tag output streams.

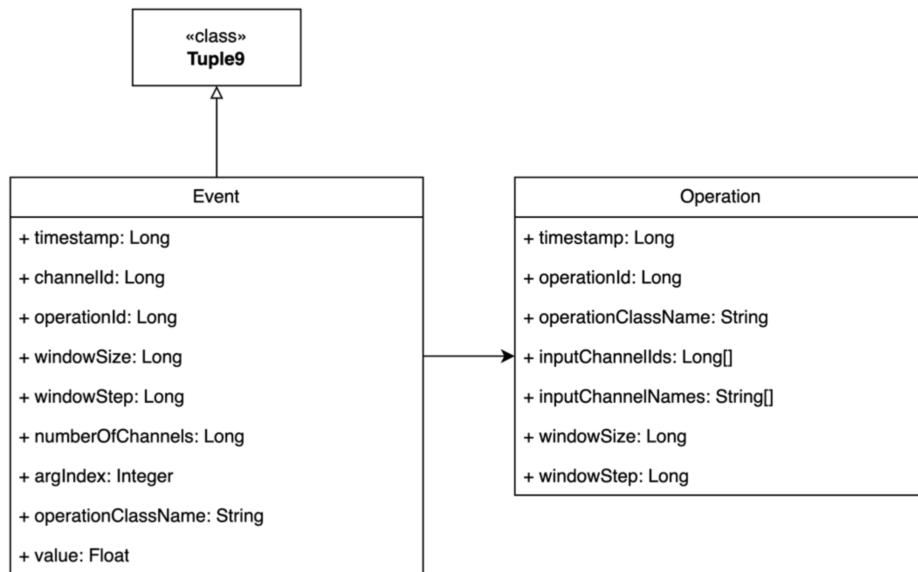


Figure 43 Pipeline Data Model

Outputs produced from an operation are serialized as a Channel, which can be aggregated using a ChannelArray to create batches. The Channel and ChannelArray models are depicted in Figure 44. A channel can contain three types of values: Real, Complex and Event. Real and Complex values, as their names suggest, are designed to represent numerical and phasor domain data. Events, on

the other hand, are designed to tag a stream with a string at particular timestamps. These models are implemented using Protobuf to make them available to consumers outside the data pipeline.

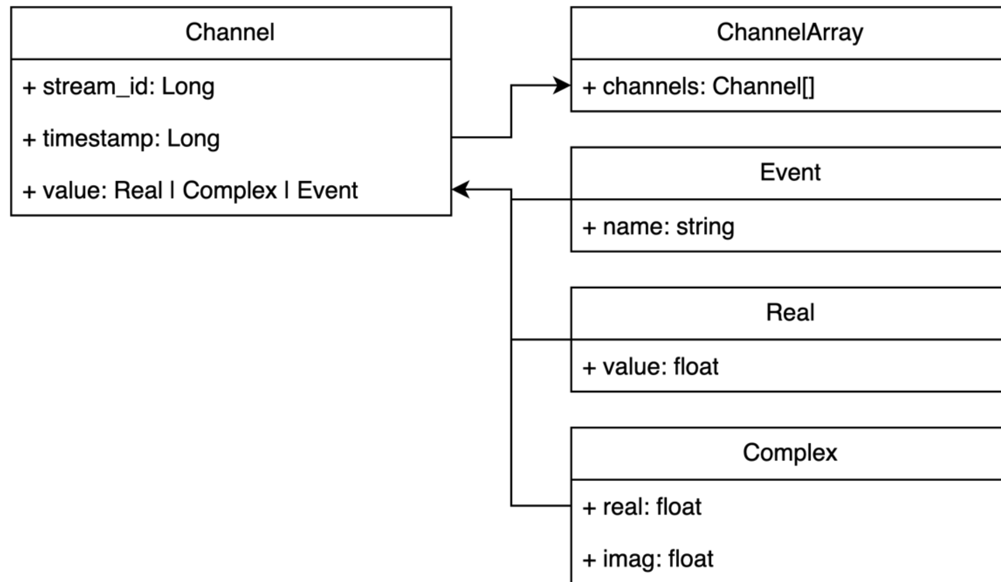


Figure 44 Output Data Model

6.3 Synchrophasor Data Pipeline

The objective of the data pipeline is to take input streams from multiple PMUs from the distributed streaming platform Amazon Kinesis™ in this case. It routes them to nodes where processing operations are executed. This work seeks to meet this objective by proposing a generalized pipeline architecture that can perform arbitrary operations on synchrophasor data. A generalized pipeline architecture can support predefined and optimized operations without the need to change the streaming job.

The shape and nature of synchrophasor streams strongly influence the pipeline architecture. A synchrophasor stream is a high-throughput, partitioned data stream of time-stamped numeric

values emitted at uniform intervals. Several substation agents connected to PMUs emit data simultaneously; this data is serialized, grouped in batches, and published in one or more partitions of a stream on the streaming platform.

At the substation agent level, batches are created by grouping measurements of consecutive timesteps; these are assigned to a stream partition by using the channel ID. This strategy was found to be the best for balancing the consumer in the processing pipeline, while other strategies, such as assigning a single partition to a single agent or defining a partition key based on time, produced unbalanced or empty partitions. Conversely, assigning partitions based on the channel ID, i.e. to each measurement, resulted in a trade-off between partition balancing and latency as the real and imaginary parts of phasors were sent to different partitions.

Batching is needed to send high-throughput data streams over TCP. Therefore, the first step in the pipeline involves deserialization and splitting the batches into individual measurements. Short latencies between the cluster nodes make it feasible to transmit individual measurements over TCP between the pipeline stages. At this point, individual measurements are keyed and assigned internally to different partitions.

A data flow diagram representing the synchrophasor processing pipeline is depicted in Figure 45. The upper path takes the operations from the operation source, which can point to an API of the metadata registry {1.1}. Watermarks are disabled {1.2} from the operation stream to avoid problems with the intermittent nature of the source. The following steps map the operations into tuples of operation ID and channel ID {1.3}. Lastly, the stream is broadcasted to be used by other operators {1.4}. Broadcasting is a mechanism to distribute state to other operators; in this instance, its purpose is to set up the configuration of the operator {2.6} on each partition.

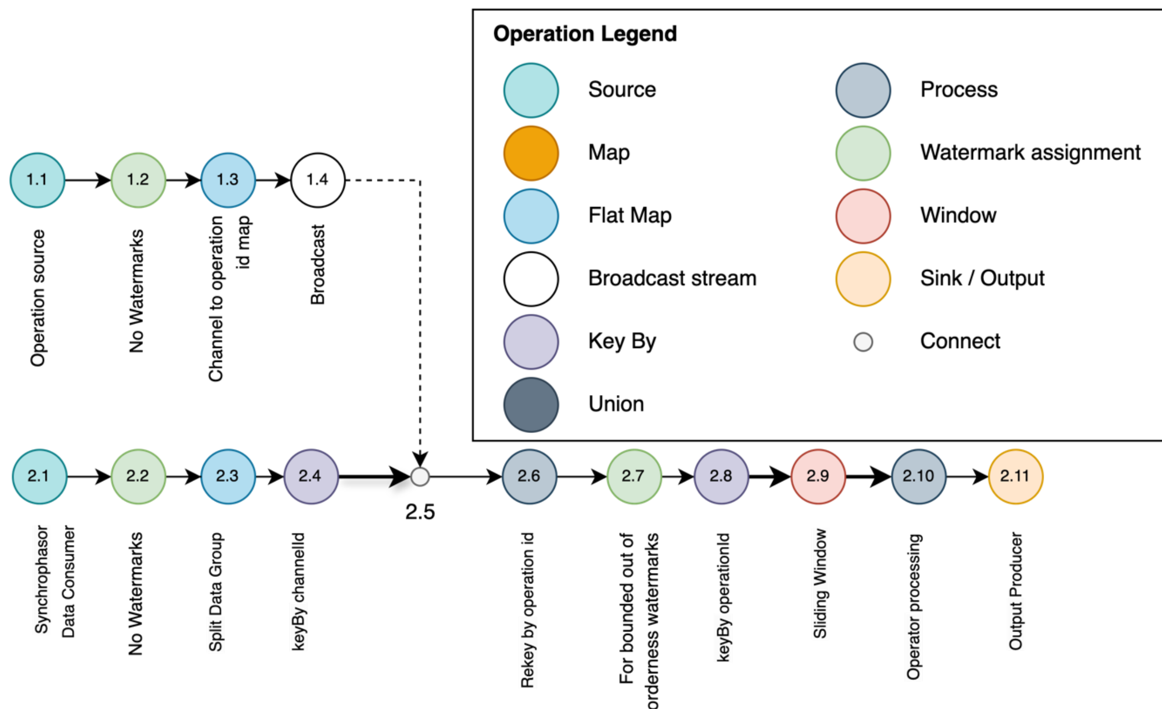


Figure 45 Synchrophasor processing pipeline data flow diagram

The data path from {2.1} to {2.11} is used for processing the synchrophasor data depending on the configuration set by the operations. The transformations that ingested data undergoes are depicted in Figure 46. Synchrophasor data enters the data stream in small batches; these are fetched from multiple partitions of the Amazon Kinesis™ stream {2.1}. No watermarks are set for the input stream at this stage {2.2}. Input batches are split and keyed by the channel ID {2.3}. At this stage {2.5}, the stream is connected to the broadcasted stream coming from {1.4}. The connect operator {2.5} crates two inputs in the process operator that set the operation ID to the incoming records {2.6}; this operation is referred to as rekey. The rekey operation {2.6} produces copies of incoming records tagged with the appropriate operation ID. Watermarks are added {2.7} to enable event time windowing over the stream. The rekeyed output stream is keyed by operation ID {2.8} to allow the stream to be partitioned using the new key; this allows the stream processing framework

to send all the records with the same key to the same partition. A custom sliding window {2.9} groups consecutive records based on two windowing parameters previously set in the rekey operation {2.6}; this aims to create windows of different lengths for each key. Lastly, the windowed stream sends data groups to the Synchrophasor Processing Operator {2.10}, where a custom processing function is applied to the group, and the output is forwarded to the output sink {2.11}.

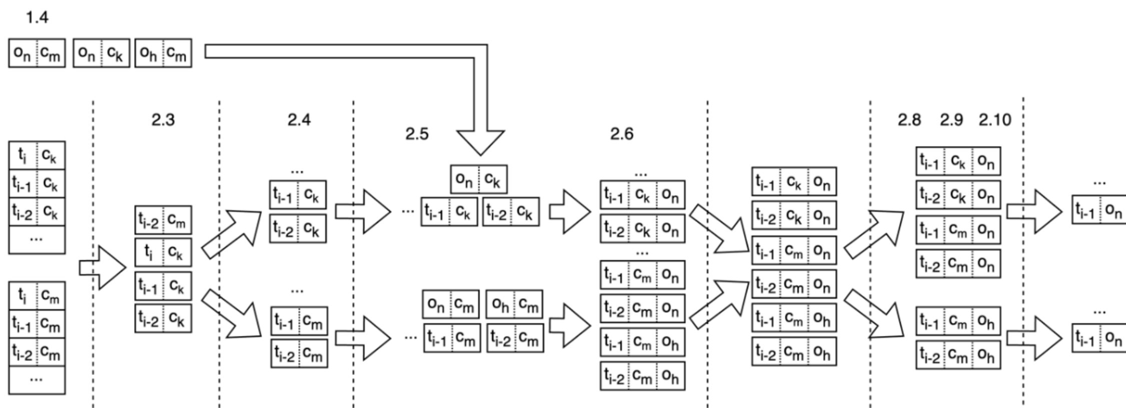


Figure 46 Data transformations in the pipeline

It is important to highlight how the system would behave when data are receiving at multiple data rates. The first step at which the event-time timestamps interact with the pipeline is in {2.7} when wattermarking is set; different data rates have no influence before this point as event-time is ignored until this point. After {2.7} the pipeline interacts with the measurement event-time in three ways: watermarking, windowing and processing operations. It is unlikely that different data rates would have a strong effect on watermarking as the only forseeable effect would be for watermakrs to be updated more often. The windowing operator is not affected either by having multiple data rates as it groups all data inside an upper and lower bound. The processing operators receives the raw groups from the windowing operator, this means that it is up to the user to define how multiple

data rates are handled inside the processing operator. When multiple data rates are present, the user must be aware that every time the operation is triggered, each channel will have a different number of measurements depending that particular channel data rate. No provissions for downsampling or interpolating missing values were made as these decisions are often application dependent. However, it is possible to extend the pipeline to include some standard strategies.

6.3.1 Processing Operator

The processing operator is the last step of the pipeline. It is in charge of applying predefined computations to a group of windowed data from multiple channels. Upon the arrival of the first measurement enriched with the operation data, the processing operator instantiates a class that extends the AbstractOperator class and implements the processing method. Each channel has a predefined position in the data structure passed to this method, this way it can be identified in the concrete implementation of operations. An example of the implementation of an operation is shown in Figure 47. This is a very simple example to illustrate how this approach makes the development of synchrophasor applications simpler by removing low-level stream processing constructs such as windowing and routing of measurements to the correct partitions. The operation shown in Figure 47 begins with the retrieval of the channel IDs from the metadata contained in the operation definition. Then, these IDs are used to retrieve the data from the window. Lastly, an operation is carried out using the data retrieved from the window object and emitted as a new event that can be published to a shared message broker for external applications to consume. This example implements a phase difference between two busses in the power system, this is a simple but interesting example as the measurements used come from different geographic locations, making it challenging for the whole system to get the measurements aliened for processing in a timely manner. The processing operator is built on top of a stateful stream processing operator that

keeps the operation definition as its state. What this means is that, once initialized it will keep the operation definition for subsequent measurements.

```java
public class PhaseDifferenceOperator extends AbstractOperation {
    @Override
    public void process(Map<Long, WindowBuffer<? extends QuantityEvent>> windows) {

        long VA_RE_A = operationDefinition.getBaseChannelIds().get(0);
        long VA_IM_A = operationDefinition.getBaseChannelIds().get(1);

        long VA_RE_B = operationDefinition.getBaseChannelIds().get(2);
        long VA_IM_B = operationDefinition.getBaseChannelIds().get(3);

        long timestamp = windows.get(VA_RE_A).getTimestamp();

        Double va_re_a = ((RealQuantity) windows.get(VA_RE_A).stream().findFirst().get()).value;
        Double va_im_a = ((RealQuantity) windows.get(VA_IM_A).stream().findFirst().get()).value;

        Double va_re_b = ((RealQuantity) windows.get(VA_RE_B).stream().findFirst().get()).value;
        Double va_im_b = ((RealQuantity) windows.get(VA_IM_B).stream().findFirst().get()).value;

        Double angle = Math.atan2(va_im_a, va_re_a) - Math.atan2(va_im_b, va_re_b);

        RealQuantity angleOutput = new RealQuantity(getOperationId(), timestamp, angle);
        emitOutputValue(angleOutput);
    }
}
```

Figure 47 Example Implementation of the Phase Difference Between Synchrophasors

6.4 Latency Measurement

The simulation setup presented in Chapter 3, specifically in Figure 12, is used to measure the performance of the whole data system. However, Figure 48 highlights in detail the sources of latency that impact the measurements in the trip from the source to the processing operator.

An initial important step is to correctly identify these sources to understand what can and cannot be optimized. For example, the time it takes for a measurement to go from 1 to 2 is very small and can hardly be reduced by changing the implementation of the software. Therefore, the system must be implemented in a way that the latency from 1 to 2 does not artificially increase it to unacceptable levels; this was shown in Chapter 4. The latency from 2 to 3 is highly dependent on the network, batch size and protocol. The latency from 3 to 4 should be small and can be influenced by the

number of partitions used. Lastly, 4 to 5 does depend on the data pipeline implementation and is the one that is measured in this section.

The latency between 4 to 5 is measured indirectly by measuring the time it takes for a measurement to arrive to the pipeline (1 to 4) and the time it takes it to arrive at the processing operator (1 to 5).

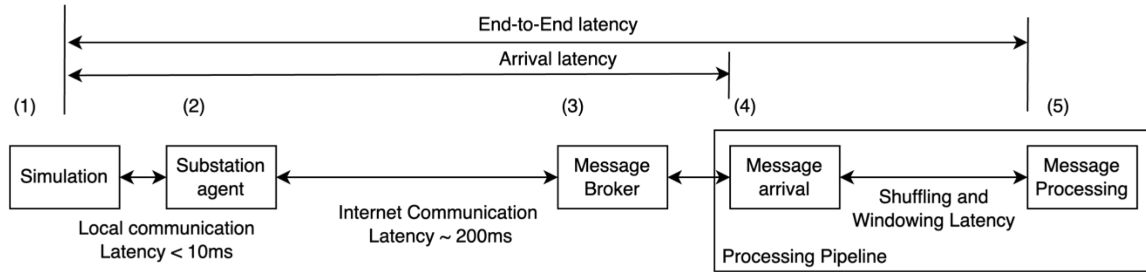


Figure 48 System Latency

Two secondary data streams as side outputs were created to capture the event-time and processing-time timestamps of the measurements once they arrive at operators {2.6} and {2.10}; these streams are combined via a union operator and sent to an Amazon Kinesis Stream. Latency data is processed offline to understand the performance of the data pipeline at different data rates. This setup is shown in Figure 49.

Four experiments were performed at 10, 20, 30, and 60 frames per second to understand how and if the data rate affects latency. The effects of the data rate on latency can limit the system's scalability. For each experiment, 5 PMUs send data concurrently to the cloud for 5 minutes, where the pipeline groups and prepares the data to perform 12 operations. Two types of operations are considered: operations between the data of a single PMU and operations that involve data from multiple PMUs. This is to ensure that latencies do not depend heavily on the choice of the data source.

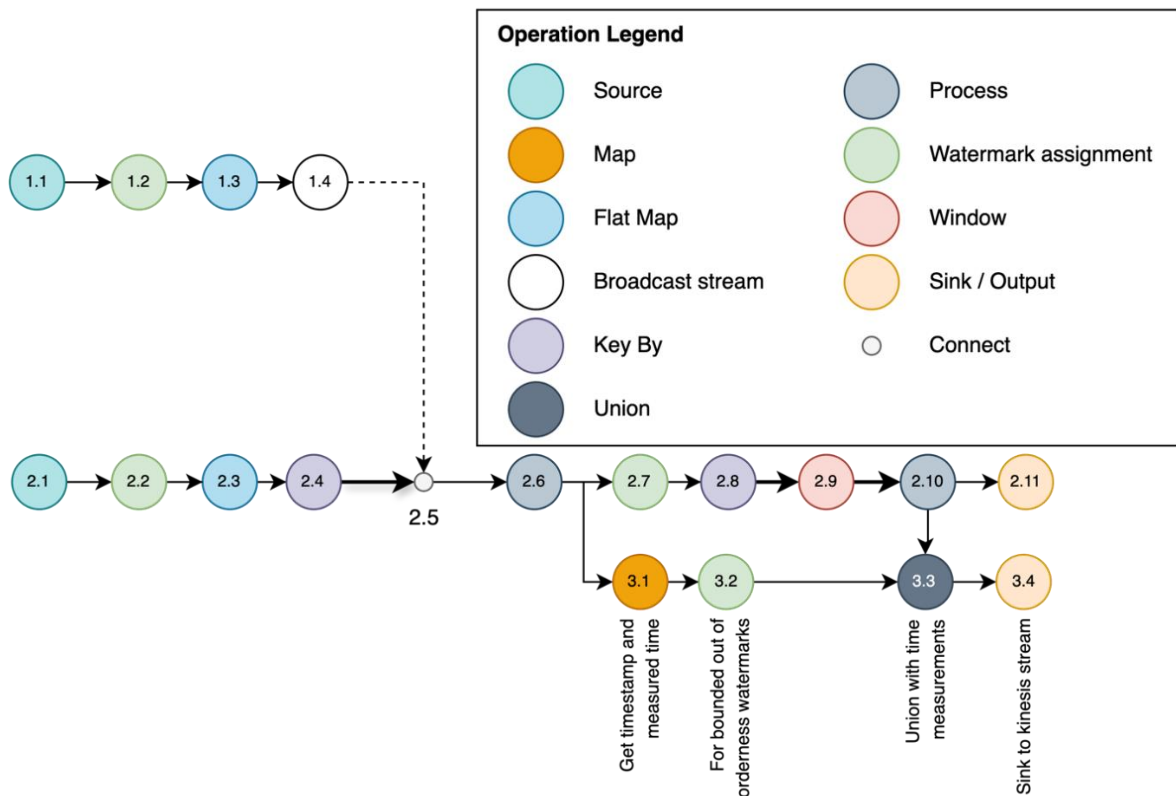


Figure 49 Latency measurement path

The first latency measurement is taken at the output of {2.6}, where data is enriched and available for processing. The second latency measurement is taken at the input of the process operator at the end of the pipeline {2.10} when data is grouped and ready for the operation to be applied.

Figure 50 presents the arrival latency measurements. At 10, 20, and 30 frames per second, the latency stays around 250ms and increases to around 400ms on average for 60 frames per second.

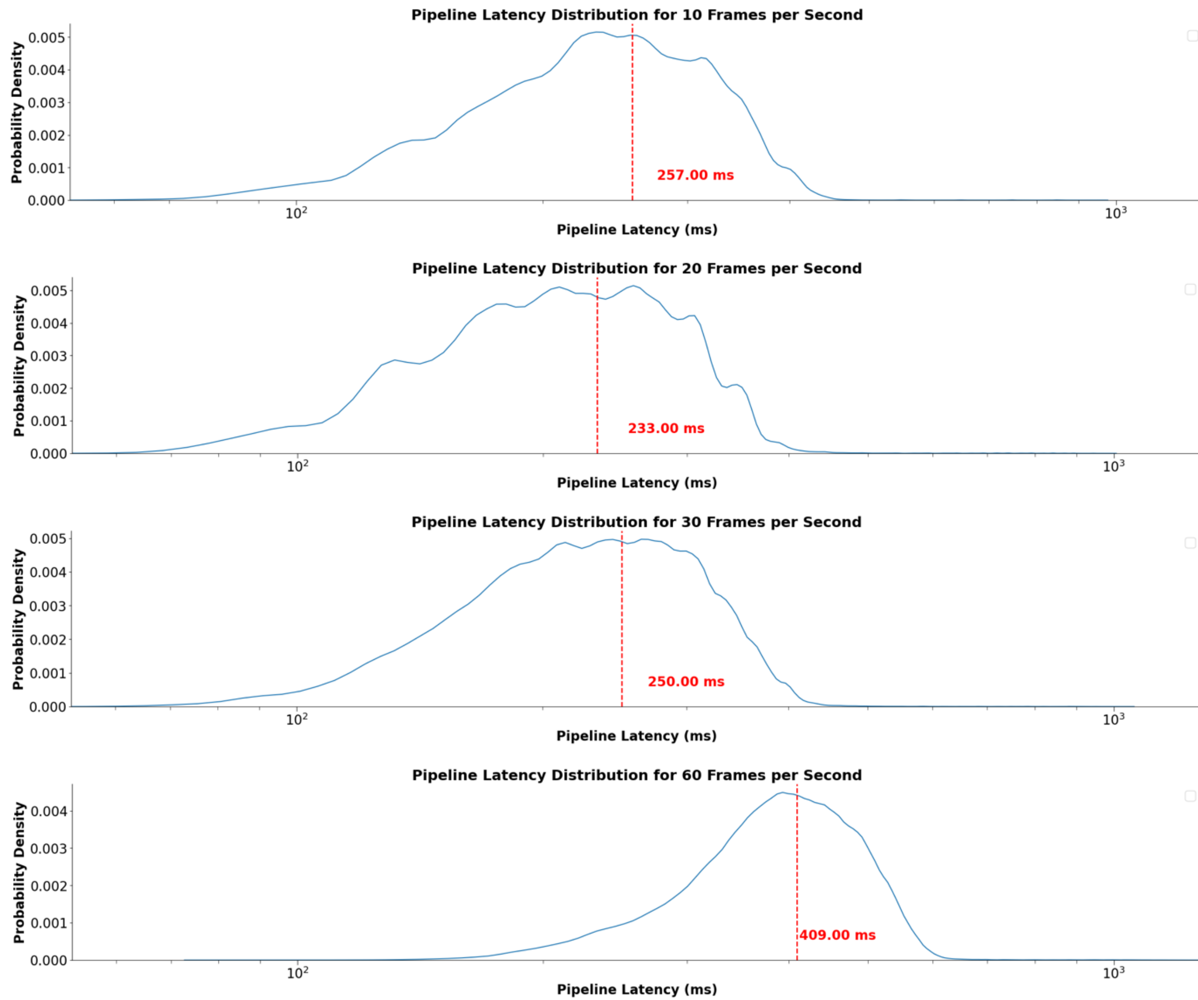


Figure 50 Arrival Latency Results

Figure 51 shows the time it takes for data to be grouped, windowed and ready for processing. The average time is around 1.35s, and it is noticed that the variance tends to increase with the data rate. The increased variance seems related to the fact that some channels lagged compared to others rather than being a direct consequence of the increased data rate; this can be seen in the shape of the distributions, which become multi-modal. A detailed analysis of arrival latencies is required to understand the source of this behaviour.

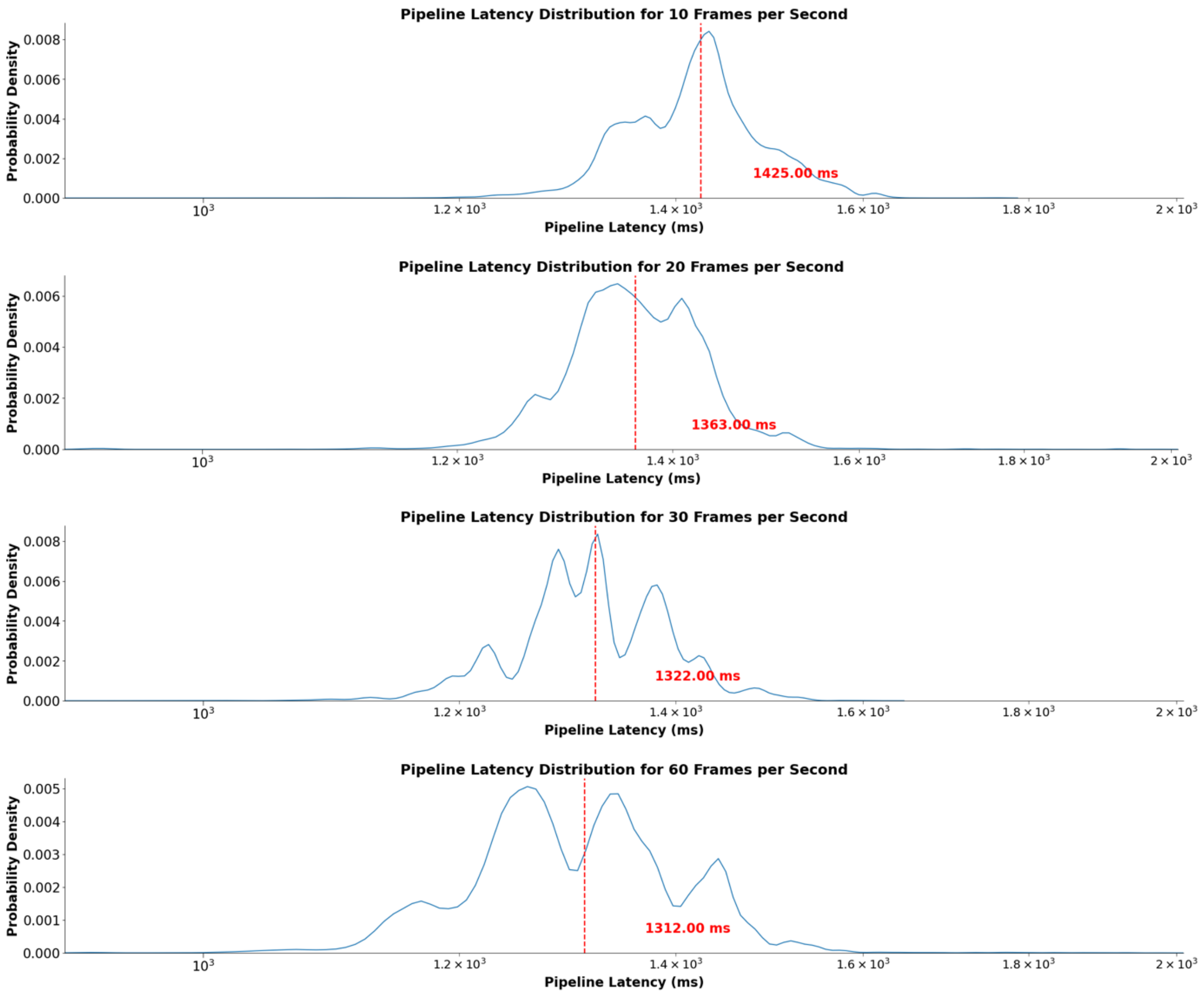
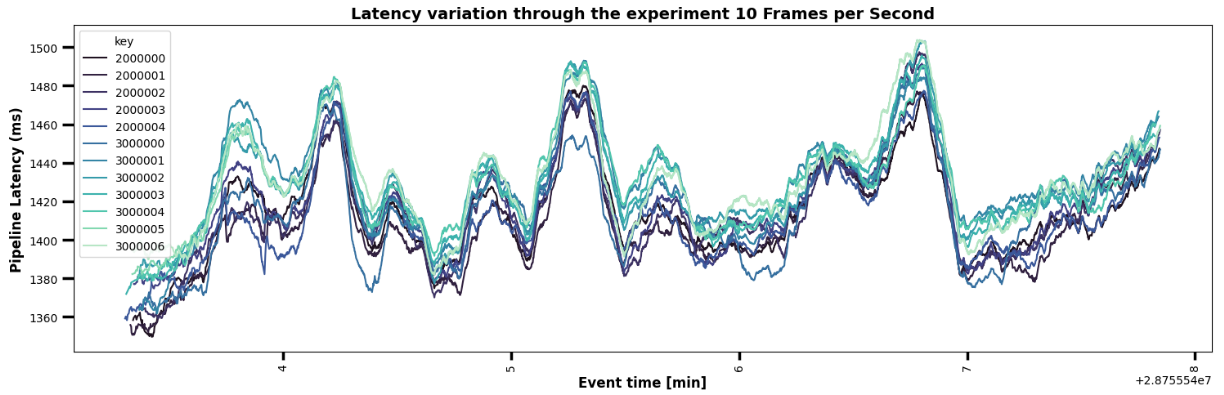
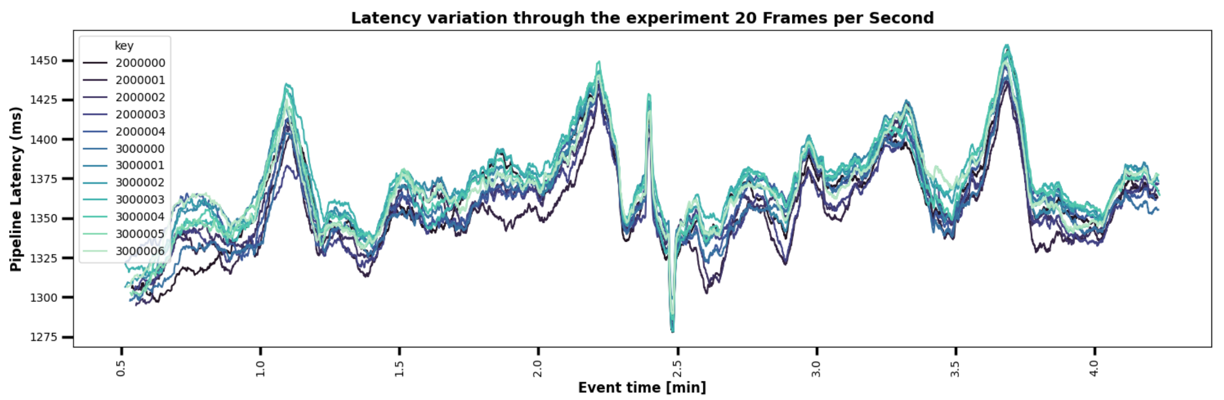


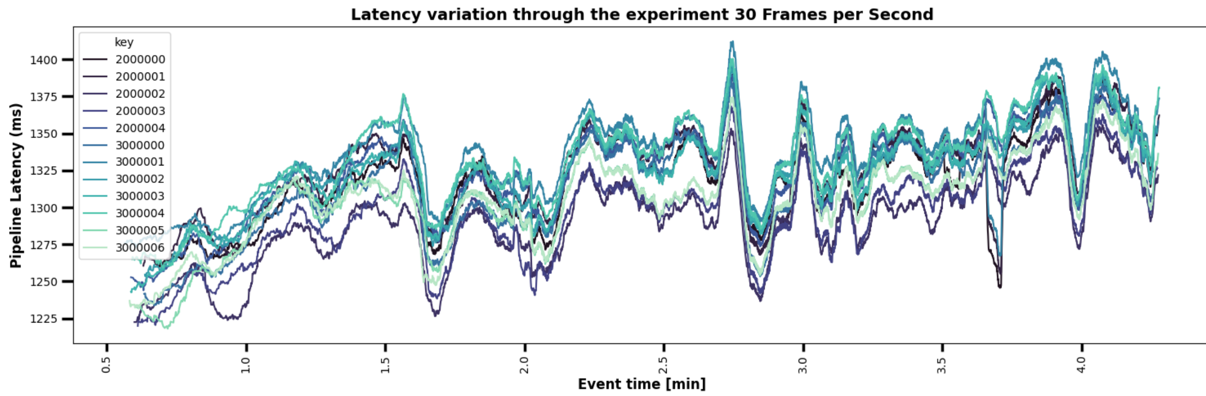
Figure 51 Data Ready for Processing Latency

Figure 52 shows the latency through the experiments. The first thing that is evident is that slow variations of the latency may influence the shape of the distributions in Figure 51. However, there is a distinctive latency behaviour in the 60 frames per second experiment. Other data rates exhibit a uniform behaviour. Lastly, Figure 53 shows the deviation of latencies from the mean of all operations. Here, the 60 frames per second data stream behaves consistently with previous observations, suggesting that some operations are lagging while the distribution for 10, 20 and 30 frames per second remains narrow and similar to each other. During these experiments, the backpressure in the Apache Flink dashboard was monitored and the partitions were verified to be

balanced which showed that the pipeline was able to respond to different data rates with the appropriate throughput.







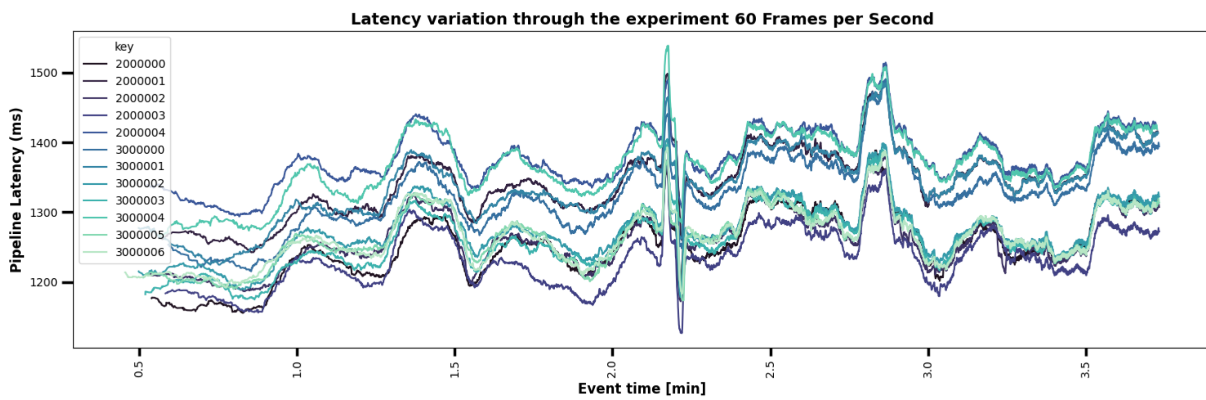


Figure 52 Latency variation in time

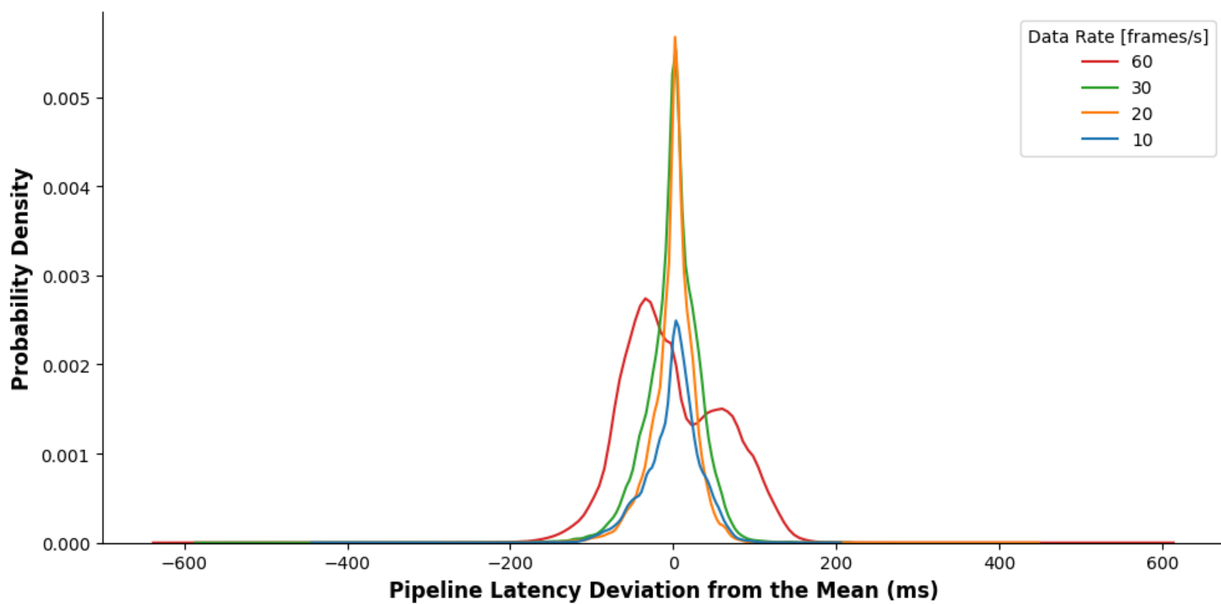


Figure 53 Latency Deviation From the Mean

6.5 Scalability, Fault Tolerance and Testing

Utilizing a cloud environment takes care of the most important aspect of scalability when compared to an on-premises system which is the ability to provide cloud resources on-demand for a price. Beyond this, the use of Apache Flink helps to make the system scalable because it provides out-of-the-box mechanism for distributing the processing application in multiple nodes.

Fault tolerance is another aspect in which Apache Flink would contribute to building better synchrophasor applications. In this context, fault tolerance would mean the ability of the processing application to withstand the loss of a computer node with or without downtime and to restart processing from the point at which it failed. This application takes advantage of Apache Flink checkpointing capabilities to keep the state of the processing function so it can restart and recover its state.

Finally, the pipeline was tested by using a real time simulator with five emulated PMUs. Testing at larger scales to understand how it would behave with larger numbers of PMUs and operations is not trivial and would require a specialized system for doing so. It was found that one of the aspects that are the most demanding for the streaming application is the need to handle data that is generated at regular intervals and this becomes difficult to test at a large scale without a source that can produce data in the same way PMUs do.

6.6 Summary

This chapter introduced some stream processing concepts in the context of synchrophasor data processing and used these to design and build a configurable data pipeline. The processing pipeline leverages stateful stream processing and the notion of event time, two key features of Apache Flink. Stateful stream processing plays a role into keeping the state of the operations applied to the

stream of measurements. The notion of time is utilized to process synchrophasor data based on the time at which the measurement was produced rather than the time at which it was received.

Performance measurements of the pipeline receiving real time data from the experimental setup described in Chapter 3 are presented. These measurements lead to conclude that using the presented methodology and architecture it is possible to receive the measurements with an average latency of around 300 ms regardless of the data rate for data rates equal or lower to 60 frames per second. The latency was shown not to be heavily influenced by the choice of data rate. Lastly, the time from the source to the processing operator was found to be around 1300ms on average which suggests that the process of retrieving data from multiple partitions routing and windowing can take up to 1s.

Chapter 7 Conclusions and Future Work

7.1 Conclusions

This thesis presents a detailed description of the implementation process of a cloud-based near-real-time synchrophasor pipeline, the necessary cloud resources, and an IEEE Std C37.118™-2005 client library. The term near-real-time was carefully chosen because in the context of power systems, real-time applications are defined to have deterministic and low latencies. The proposed approach assumes a shared transmission medium and makes no provisions for adding quality of service (QoS) rules that could guarantee a deterministic latency on the receiving end nor last-mile dedicated links.

A literature review of the evolution of synchrophasor technology and the availability of big-data platforms for synchrophasors was conducted. Little information is available on the data engineering aspects of synchrophasor technology. Moreover, the growing number of sensors and applications and the appearance of some big-data platforms make a strong case for the evolution of data concentrators from single-node devices to distributed systems.

A significant research gap was identified as there are not many references to the implementation of synchrophasor data processing pipelines in the available literature. Moreover, it was shown that the nature of synchrophasor data streams and synchrophasor data applications do not follow the typical structure of stream processing applications and need additional development. To address this, a novel architecture for processing synchrophasor data using a stream processing framework called Apache Flink was proposed. This architecture could potentially be implemented in other frameworks, granted that it has features similar to Apache Flink. A methodology for testing the

pipeline was proposed and used to assess its performance. Lastly, a real-time simulator was used to validate and test the entire system.

To overcome the limitations that traditional synchrophasor hierarchical networks can impose on data applications, a system was designed to extract synchrophasor measurements through a secure channel to a cloud application. For this end, a software called the substation agent was implemented and thoroughly described. The function of the substation agent is to extract IEEE Std C37.118™-2005 data directly from the PMU or PDC in the substation transform it and forward it to the cloud where it is processed.

Cloud infrastructure resources were provisioned to host the synchrophasor processing pipeline and the distributed message broker that receives measurements from the substation agent. The synchrophasor processing pipeline was created using Apache Flink and hosted in a cloud-based service. In addition, a small web application was created to be used as a metadata repository to keep information on channels and operations.

Based on the experience gained using Apache Flink, a state-of-the-art stream processing framework, some conclusions can be made about the challenges of using modern stream processing techniques in near-rea-time synchrophasor processing. Stateful stream processing, a notion of event time and exactly one semantics, comes in very handy when processing synchrophasor data.

Stateful stream processing allows easy creation and management of the state in a processing pipeline where synchrophasor data needs to be enriched with metadata and external inputs such as the operations described in this thesis. In addition, state management tools make it easy to create a resilient pipeline where the resiliency and state management layers are decoupled from the processing logic. Moreover, the event-time notion is embedded in the synchrophasor protocol

structure and philosophy; for this reason, the support for event-time operations makes it suitable for processing synchrophasor data.

Conversely, there are some aspects of modern stream processing techniques where there is room for improvement when it comes to synchrophasor data. Stream processing techniques seem to be optimized for cases when there is a predefined number of streams that are processed together to produce a result. In contrast, synchrophasor data is often a single keyed stream where each key identifies a channel and different type of variable; the number of keys in this stream is large as each PMU can potentially produce tens of other channels. One way of handling this would be to split the incoming stream into the streams that are needed for a particular application, combine them, apply a time window and perform an arbitrary processing task. This may result in many different implementations where much of the windowing logic needs to be implemented for each application. A systematic and efficient way of dealing with this scenario is still not readily available. Another issue that arises when attempting to process synchrophasor data from a distributed message broker is that a single measurement can be needed on multiple different processing applications; this inevitably creates a complex scenario where data exchange between multiple compute nodes is necessary.

As part of the development of the substation agent, a command-line interface was developed to communicate with existing PMUs and forward data from the substations to a cloud-based system. Likewise, a cloud-based ingest and processing system was designed and implemented.

Lastly, these are some of the ways in which the proposed solution improves the systems identified in the literature review. First, the use of a distributed system for data ingestion increases reliability and horizontal scalability as the cloud infrastructure includes built-in mechanisms to respond to changes and failures. Second, the introduction of a stream processing framework to synchrophasor

analysis presents an opportunity to take advantage of mature patterns and best practices that can ease the creation of synchrophasor applications by managing some of the complexities that arise from interacting with large scale data streams. Finally, the introduction of edge devices at the substation level such as the substation agent brings opportunities for extracting data from substations with less management overhead than the traditional approaches.

7.2 Future work

The system proposed in this research is only an initial step towards the creation of general-purpose tools for processing synchrophasor data at a scale. Likewise, the findings and conclusions are limited to the scope of the project, and more research is needed to form a better understanding of the subject. This section proposes a list of future research directions, possible improvements and trends in the literature.

- The proposed data pipeline has multiple network shuffling steps within the cloud. It should be possible to reduce this to one network shuffle; however, doing so may need definition of new stream processing operators.
- The data ingestion strategy proposed in this study can be optimized by leveraging the STTP protocol rather than HTTP, which is not optimized for this type of data transmission. Moreover, there are no distributed implementations of STTP servers, which represents a good research opportunity. This integration would require replacing the distributed message broker by an STTP gateway.
- Time synchronization for cloud computers preferably with GPS clock is highly desirable, if not required, for synchrophasor application integration into the cloud. Currently, how to achieve is not very clear and more research is needed in this direction.

- In this research, the system's performance was assessed using a real-time simulator and emulated PMU. Testing with an arbitrarily large number of PMUs may become necessary in the future. However, a framework for testing large-scale PMU systems needs to be developed which poses many technical challenges.
- It was found that stream processing primitives available in Apache Flink, many of which are also present in other frameworks, felt short to provide useful interfaces for synchrophasor data in some cases. This may arise from the fact that there is a difference in the nature of typical stream processing applications where there are many streams with or without keys that are combined to produce a result, whereas in synchrophasor applications the most common scenario is a single stream with many partition keys (or channels) that need to be grouped to perform operations on itself. While it is still possible to use stream processing techniques for synchrophasor applications and these techniques provide a lot of value; some of the challenges suggest that there are research opportunities developing more specialized techniques for synchrophasor and time series data processing.
- The proposed approach can bring some opportunities to address distribution system level applications such as collecting and processing power quality data.
- While the proposed approach is in principle compatible with some inference applications either by implementing them directly in the processing operator or by preprocessing the data in a processing operator and making the features available as a stream or in a distributed message broker for machine learning applications to use. Machine learning opens the door to other applications that could be addressed by extending the processing pipeline. One example would be retraining an existing algorithm with newer data.

References

[1] G. A. Ortiz, D. G. Colome, and J. J. Quispe Puma, "State estimation of power system based on SCADA and PMU measurements," in 2016 IEEE ANDESCON, Arequipa: IEEE, Oct. 2016, pp. 1–4. doi: 10.1109/ANDESCON.2016.7836221.

[2] A. G. Phadke, "Synchronized phasor measurements in power systems," IEEE Comput. Appl. Power, vol. 6, no. 2, pp. 10–15, Apr. 1993, doi: 10.1109/67.207465.

[3] M. C. Garcia, D. Dotta, L. Pereira, M. C. D. Almeida, O. L. D. Santos, and L. C. P. Da Silva, "Design and Development of D-PMU Module for Smart Meters," in 2020 IEEE Power & Energy Society General Meeting (PESGM), Montreal, QC, Canada: IEEE, Aug. 2020, pp. 1–5. doi: 10.1109/PESGM41954.2020.9281601.

[4] "IEEE Standard for Synchrophasor Data Transfer for Power Systems (IEEE Std C37.118.2TM-2011)".

[5] J. Taft, "NASPInet 2.0 Architecture Guidance Version 1.19," *NASPI*, Jul. 2019. https://www.naspi.org/sites/default/files/reference_documents/NASPInet%202%20v1.19_PNNL%20.pdf (accessed Jun, 2024).

[6] S. Murphy, K. Jones, T. Laughner, M. Bariya, and A. Von Meier, "Accelerating Artificial Intelligence on the Grid," in 2020 Clemson University Power Systems Conference (PSC), Clemson, SC, USA: IEEE, Mar. 2020, pp. 1–7. doi: 10.1109/PSC50246.2020.9131317.

[7] M. Baba et al., "A review of the importance of synchrophasor technology, smart grid, and applications," Bulletin of the Polish Academy of Sciences Technical Sciences, pp. 143826–143826, Nov. 2022, doi: 10.24425/bpasts.2022.143826.

[8] M. Khan, M. Li, P. Ashton, G. Taylor, and J. Liu, "Big data analytics on PMU measurements," in 2014 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), Xiamen, China: IEEE, Aug. 2014, pp. 715–719. doi: 10.1109/FSKD.2014.6980923.

[9] S. Kantra, H. A. Abdelsalam, and E. B. Makram, "Application of PMU to detect high impedance fault using statistical analysis," in 2016 IEEE Power and Energy Society General Meeting (PESGM), Boston, MA, USA: IEEE, Jul. 2016, pp. 1–5. doi: 10.1109/PESGM.2016.7741454.

[10] A. Kummerow, S. Nicolai, and P. Bretschneider, "Spatial and temporal PMU data compression for efficient data archiving in modern control centres," in 2018 IEEE International Energy Conference (ENERGYCON), Limassol: IEEE, Jun. 2018, pp. 1–6. doi: 10.1109/ENERGYCON.2018.8398809.

[11] M. P. Andersen and D. E. Culler, "BTrDB: Optimizing Storage System Design for Timeseries Processing".

[12] C. Wang et al., "Apache IoTDB: A Time Series Database for IoT Applications," Proc. ACM Manag. Data, vol. 1, no. 2, pp. 1–27, Jun. 2023, doi: 10.1145/3589775.

[13] I. Kosen et al., “UPS: Unified PMU-Data Storage System to Enhance T+D PMU Data Usability,” IEEE Trans. Smart Grid, vol. 11, no. 1, pp. 739–748, Jan. 2020, doi: 10.1109/TSG.2019.2916570.

[14] M. P. Andersen, S. Kumar, C. Brooks, A. Von Meier, and D. E. Culler, “DISTIL: Design and implementation of a scalable synchrophasor data processing system,” in 2015 IEEE International Conference on Smart Grid Communications (SmartGridComm), Miami, FL, USA: IEEE, Nov. 2015, pp. 271–277. doi: 10.1109/SmartGridComm.2015.7436312.

[15] A. G. Phadke, “Synchronized phasor measurements-a historical overview,” in IEEE/PES Transmission and Distribution Conference and Exhibition, Yokohama, Japan: IEEE, 2002, pp. 476–479. doi: 10.1109/TDC.2002.1178427.

[16] A. G. Phadke, M. Ibrahim, and T. Hlibka, “Fundamental basis for distance relaying with symmetrical components,” IEEE Trans. on Power Apparatus and Syst., vol. 96, no. 2, pp. 635–646, Mar. 1977, doi: 10.1109/T-PAS.1977.32375.

[17] G. D. Friedlander, “The Northeast power failure‑a blanket of darkness,” IEEE Spectr., vol. 3, no. 2, pp. 54–73, Feb. 1966, doi: 10.1109/MSPEC.1966.5216894.

[18] A. Phadke, J. Thorp, and M. Adamiak, “A New Measurement Technique for Tracking Voltage Phasors, Local System Frequency, and Rate of Change of Frequency,” IEEE Trans. on Power Apparatus and Syst., vol. PAS-102, no. 5, pp. 1025–1038, May 1983, doi: 10.1109/TPAS.1983.318043.

[19] “IEEE Standard for Synchrophasers for Power Systems (IEEE Std 1344-1995 R2001),” IEEE. doi: 10.1109/IEEESTD.1995.93278.

[20] “IEEE Std C37.118-2005 IEEE Standard for Synchrophasors for Power Systems”.

[21] IEEE Standard for Synchrophasor Measurements for Power Systems (IEEE Std C37.118.1TM-2011). doi: 10.1109/IEEESTD.2011.6111219.

[22] “Gateway Exchange Protocol Overview.” Grid Protection Alliance, 2013.

[23] J. R. Carroll and F. R. Robertson, “A Comparison of Phasor Communication Protocols,” PNNL--28499, 1504742, Feb. 2019. doi: 10.2172/1504742.

[24] IEEE Standard for Streaming Telemetry Transport Protocol (STTP). doi: 10.1109/IEEESTD.2024.10620421.

[25] R. Carroll, “Appendix E - TSSC Algorithm.” Accessed: May 02, 2024. [Online]. Available: https://github.com/sttp/Specification/blob/master/Sections/TSSCAlgorithm.md

[26] V. Kotu, Data science: concepts and practice, Second edition. Cambridge, MA: Elsevier/Morgan Kaufmann Publishers, 2019.

[27] E. Brewer, “CAP twelve years later: How the ‘rules’ have changed,” Computer, vol. 45, no. 2, pp. 23–29, Feb. 2012, doi: 10.1109/MC.2012.37.

[28] J. Reis and M. L. Housley, Fundamentals of data engineering: plan and build robust data systems, First edition. Sebastopol, CA: O’Reilly Media, 2022.

[29] I. Lee, “Big data: Dimensions, evolution, impacts, and challenges,” Big data.

[30] M. Klettke and U. Störl, "Four Generations in Data Engineering for Data Science: The Past, Presence and Future of a Field of Science," Datenbank Spektrum, vol. 22, no. 1, pp. 59–66, Mar. 2022, doi: 10.1007/s13222-021-00399-3.

[31] T. Akidau, "Streaming Systems".

[32] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing NY USA: ACM, Oct. 2003, pp. 29–43. doi: 10.1145/945445.945450.

[33] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Commun. ACM, vol. 51, no. 1, pp. 107–113, Jan. 2008, doi: 10.1145/1327452.1327492.

[34] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA: IEEE, May 2010, pp. 1–10. doi: 10.1109/MSST.2010.5496972.

[35] M. Zaharia et al., "Apache Spark: a unified engine for big data processing," Commun. ACM, vol. 59, no. 11, pp. 56–65, Oct. 2016, doi: 10.1145/2934664.

[36] J. Kreps, "Questioning the Lambda Architecture." [Online]. Available: https://www.oreilly.com/radar/questioning-the-lambda-architecture/

[37] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache FlinkTM: Stream and Batch Processing in a Single Engine".

[38] Schwarzkopf, Malte, "The Remarkable Utility of Dataflow Computing." [Online]. Available: https://www.sigops.org/2020/the-remarkable-utility-of-dataflow-computing/

[39] S. Shahrivari, "Beyond Batch Processing: Towards Real-Time and Streaming Big Data," Computers, vol. 3, no. 4, pp. 117–129, Oct. 2014, doi: 10.3390/computers3040117.

[40] T. Akidau et al., "Watermarks in stream processing systems: semantics and comparative analysis of Apache Flink and Google cloud dataflow," Proc. VLDB Endow., vol. 14, no. 12, pp. 3135–3147, Jul. 2021, doi: 10.14778/3476311.3476389.

[41] "Windows." Apache Flink Documentation. Accessed: Jun. 10, 2024. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/windows/

[42] "Generating Watermarks." Apache Flink Documentation. Accessed: Jun. 10, 2024. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/event-time/generating_watermarks/

[43] Z. Zhong et al., "Power System Frequency Monitoring Network (FNET) Implementation," IEEE Trans. Power Syst., vol. 20, no. 4, pp. 1914–1921, Nov. 2005, doi: 10.1109/TPWRS.2005.857386.

[44] S.-J. S. Tsai et al., "Study of global frequency dynamic behavior of large power systems," in IEEE PES Power Systems Conference and Exposition, 2004., New York City, NY, USA: IEEE, 2004, pp. 569–576. doi: 10.1109/PSCE.2004.1397518.

[45] Y. Zhang et al., "Wide-Area Frequency Monitoring Network (FNET) Architecture and Applications," IEEE Trans. Smart Grid, vol. 1, no. 2, pp. 159–167, Sep. 2010, doi: 10.1109/TSG.2010.2050345.

[46] D. Zhou et al., “Distributed Data Analytics Platform for Wide-Area Synchrophasor Measurement Systems,” IEEE Trans. Smart Grid, vol. 7, no. 5, pp. 2397–2405, Sep. 2016, doi: 10.1109/TSG.2016.2528895.

[47] T. H. Cormen, Ed., Introduction to algorithms, 3rd ed. Cambridge, Mass: MIT Press, 2009.

[48] V. S. Kumar, T. Wang, K. S. Aggour, P. Wang, P. J. Hart, and W. Yan, “Big Data Analysis of Massive PMU Datasets: A Data Platform Perspective,” in 2021 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), Washington, DC, USA: IEEE, Feb. 2021, pp. 1–5. doi: 10.1109/ISGT49243.2021.9372203.

[49] Protocol Buffers (Protobuf). Google LLC. Accessed: Jul. 10, 2024. [Online]. Available: https://protobuf.dev/

[50] “Protocol Buffers Documentation: Encoding.” Google LLC. Accessed: Aug. 01, 2024. [Online]. Available: https://protobuf.dev/programming-guides/encoding/

[51] M. Gogolla, “Unified Modeling Language,” in Encyclopedia of Database Systems, L. Liu and M. T. Özsu, Eds., Boston, MA: Springer US, 2009, pp. 3232–3239. doi: 10.1007/978-0-387-39940-9_440.

[52] Python 3 struct. [Online]. Available: https://docs.python.org/3/library/struct.html

[53] PMU Connection Tester. Grid Protection Alliance (GPA). Accessed: Jun. 10, 2023. [Online]. Available: https://github.com/GridProtectionAlliance/PMUConnectionTester

[54] S. Sandi, B. Krstajic, and T. Popovic, “pyPMU — Open source python package for synchrophasor data transfer,” in 2016 24th Telecommunications Forum (TELFOR), Belgrade, Serbia: IEEE, Nov. 2016, pp. 1–4. doi: 10.1109/TELFOR.2016.7818916.

[55] S. Šandi and T. Popovi, “PYTHON IMPLEMENTATION OF IEEE C37.118 COMMUNICATION PROTOCOL,” vol. 21, no. 1.

[56] U. Windl, D. Worley, D. Dalton, and M. Martinec, “The NTP FAQ and HOWTO.” https://www.ntp.org/ntpfaq/. Accessed: Aug. 23, 2024. [Online]. Available: https://www.ntp.org/ntpfaq/

[57] N. Kruber, “Flink Serialization Tuning Vol. 1: Choosing your Serializer — if you can.” Apr. 15, 2020. [Online]. Available: https://flink.apache.org/2020/04/15/flink-serialization-tuning-vol.-1-choosing-your-serializer-if-you-can/