OBTAINING HIGH PERFORMANCE PHASOR MEASUREMENTS IN A GEOGRAPHICALLY DISTRIBUTED STATUS DISSEMINATION NETWORK

By

RYAN JOHNSTON

A thesis submited in partial fulfilment of
the requirements for the degree of

MASTER OF SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

AUGUST 2005

To the Faculty of Washington State University:

The members of the committee appointed to examine the thesis of RYAN JOHNSTON find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGEMENT

Foremost, I would like to thank my committee members for providing me with their valuable time. In particular, I would like to thank my advisor and committee chair, Dr. Carl Hauser, for all of the work he has done to help me formulate and execute my research.

I would also like to thank Ruby Young for all of her assistance in helping me through all of the paper work and submission deadlines involved in the procurement of my degree.

I would like to thank the school of Electrical Engineering and Computer Science for providing me with the opportunity to pursue a master's degree.

This research was partially supported by NSF grant ITR CCR-0326006, for which I am very grateful.

OBTAINING HIGH PERFORMANCE PHASOR MEASUREMENTS IN A GEOGRAPHICALLY DISTRIBUTED STATUS DISSEMINATION NETWORK

Abstract

by Ryan Johnston, M.S.
Washington State University
August 2005

Chair: Carl Hauser

The emergence of phasor measurement units (PMUs) coupled with GPS time devices makes it feasible to directly compare timestamps collected at different locations without taking special measures to account for clock drift. GridStat is a flexible publish-subscribe status dissemination middleware framework with capabilities that include rate-filtered multicast for each subscription. In this thesis we explore GridStat's ability to hide the complexities of distributed systems from application developers while efficiently providing time-synchronous groups of PMU data from physically disparate locations.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER ONE

INTRODUCTION

A power grid is a complex power distribution infrastructure composed of thousands of miles of transmission lines connecting power generators to power consumers. Keeping such a complex system stable and resilient to failure is not a trivial task. Small fluctuations in one part of the grid can cascade into system wide failures if left unchecked. It is therefore imperative to be aware of what is happening within the grid so that proper reactionary measures can be taken to insure stability and uninterrupted service.

The growing availability of synchronous phasor measurement units (PMUs) and their deployment on power grids is making available a wealth of new real-time data about the state of the grid. PMU data contains much more information than is typically measured in existing SCADA systems. PMU measurements are synchronized and timestamped using a global clock (GPS). A PMU's measurements, made 20 to 60 times per second, allow derivation of the bus current, voltage, phase and frequency where it is attached.

Existing mechanisms for using these data are mainly targeted at collecting them in archival central repositories for use in after-the-fact analysis of system response to normal and abnormal events. Because of their centralized nature, these collection mechanisms have limited ability to support the use of these data for real-time applications. For example, if these data are to be used to make control decisions that involve making operational changes at various points throughout the grid, then delay is

incurred to first transfer the sampled data to the centralized repository, and then to retransmit the control message to the destination in the grid where it is to be implemented.

Nevertheless, the ability to receive status data from a variety of remote locations opens new, largely unexplored opportunities for improving the power grid's control systems to provide better reliability and efficiency. Ongoing research, [18, 19], suggests that PMU data will be valuable in fast control applications, but widespread use of such mechanisms will require a more flexible communication infrastructure than exists today. Current fast controls are either local or require dedicated point-to-point communication links. Such a communication infrastructure makes it expensive to accommodate changing needs.

GridStat is a middleware framework for dissemination of operational power grid status data (both digital and analog). Middleware is a layer of software above the operating system which provides a common programming abstraction across a distributed computing system. Its high-level building blocks shield the programmer from many different kinds of heterogeneity that are inherent in a distributed system, including that of CPU, operating system, and programming language.

A fundamental design motivation of GridStat was to provide flexible, managed, low-latency sharing of status data among entities that participate in operation of an interconnected power grid. GridStat's management capabilities are intended to ease the deployment of applications by encapsulating solutions to problems such as quality of service (QoS), reliable delivery, and security in the communication infrastructure rather than leaving their solution to applications. GridStat was designed to provide high-

bandwidth communication for power grid data, delivering the right data to the right computer at the right frequency with the ability to easily adapt to changing needs.

In this thesis we examine GridStat's suitability as the communication medium for PMU data distribution. We show GridStat's ability to meet the latency and bandwidth requirements of various PMU data applications. While GridStat is used heavily in this research, this research did not involve the actual development of GridStat, but rather used GridStat as a blackbox middleware system.

Additionally we present an application development framework created as part of this research that removes some of the complexity involved in deploying GridStat applications by acting as a value added abstraction layer between GridStat, and GridStat application developers. This framework was developed to further meet GridStat's avowed goal of removing the complexity of distributed systems programming by complementing it with a framework that eases the creation of applications in general.

CHAPTER TWO

BACKGROUND

2.1 Power System Monitoring

2.1.1 Introduction

Advances in power systems monitoring, coupled with the availability of high speed data networks, make it possible to monitor the power grid in real-time. This monitoring makes it possible to not only determine the state of the power grid at a given moment, but perhaps even use the monitored data as a basis for making automated control decisions. Automating control decisions makes it possible for the power grid to be more adaptive, and resilient to disturbances.

2.1.2 Phasors

In an AC power system, voltage and current can be represented as a sine wave oscillating at the system frequency. A phasor is a vector representation of such a wave, and is composed of a magnitude, and phase angle. Alternatively a phasor may be represented in the complex plane, with a real and imaginary component.

2.1.3 PMU data

PMUs collect phasor information, and are typically placed in a substation to monitor that substation's three phase voltages and current in lines, transformers, and loads that terminate there [7]. These quantities are collected along with frequency, and

each sample is timestamped with a time value that is accurate to within one microsecond of UTC.

The accurate timestamping is achieved by using GPS clock synchronization at each of the data collection sites. This time keeping has very important implications for our research. In particular, the dependability of the timestamps is unusual in distributed systems, as discussed later in this chapter.

The length of an IEEE 1344 data packet depends on the number of phasor and digital quantities reported by a particular PMU. A basic PMU packet is 14 bytes long consisting of

- 4-byte second-of-century (SOC)
- 2-byte sample count (SMPCNT)
- 2-byte status (STAT)
- 2-byte frequency (FREQ)
- 2-byte rate-of-change-of-frequency (DFREQ)
- 2-byte CRC16

A separate configuration file specifies how many 4-byte phasors are inserted in each data frame between the STAT and FREQ words and how many 2-byte digital status words are inserted between the DFREQ and CRC16 words. For example, a PMU measuring 3-phase voltage phasors and no digital status would send 26-byte frames; one measuring 3-phase voltage, and current, and 32 bits of digital status would send 42-byte frames.

The configuration file also controls the frequency at which frames are sent. Twenty or 30 frames per second is common today. Researchers looking into use of PMU

data for wide-area control suggest that moving to 60 frames per second would be useful in their application, [24].

Even at only 20 frames per second a PMU device produces far more data than does older-technology power system monitoring equipment designed for the 2- or 4-second SCADA polling cycle. At 60 frames per second a PMU measuring 3-phase voltage and current and 32 digital samples (42-byte frames) produces just over 20Kbits/second: huge by the standards of legacy SCADA but miniscule by the standards of even consumer-grade broadband networking service.

2.1.4 PMU Data Applications

PMU data can be used in numerous applications, including event analysis, system monitoring and real-time controls.

Event Analysis

Phasor measurements have extremely low noise and precise timing, which allows their collective data to be used to monitor system wide events and disturbances. For instance, a system composed of PMUs would be sensitive enough to track the progression of minor system changes throughout the grid. Such precision makes it possible to evaluate and compare real power grid performance to predicted system models. This comparison allows critical analysis of predicted system models, so they may be refined to more accurately reflect real world grid dynamics.

System Monitoring

If PMU data can be collected from throughout the grid, it becomes possible to analyze the current state of the system. What is important for this application is the ability to receive PMU data frames that were sampled at the same moment in time (or as close as possible). PMUs coupled with GPS time devices are able to create timestamps that are within one microsecond of UTC, so they have the ability to know the current time, the only complication then is to ensure they collect their samples at the proper moments, which some PMUs already ensure. Once the samples are collected, they must be gathered at a centralized location for comparison.

Real-time Control

If the state of the grid can be ascertained in a reasonable amount of time it is possible to make control decisions based on the collected data. The aforementioned reasonable amount of time is of course specific to the real-time control application, as some controls have stronger latency requirements than others.

2.2 Distributed Systems

2.2.1 Overview

In Computer Science, the term distributed system is rather loosely defined, and its specific definition has been the source of debate. [22], defines a distributed system as a collection of independent computers that appears to its users as a single coherent system. For the purposes of this discussion, that definition will be sufficient.

Because distributed systems are comprised of many computer systems, it is often difficult to determine the order of events that occur throughout such a system. In general, computer clocks are not accurate enough to be relied on to determine when an event occurs on one machine in relation to another. Many algorithms exist to establish a global ordering of events with varying degrees of precision [3, 12]. However, all such algorithms require additional messages to be passed through the network. PMU data is unique in that it contains a universally synchronized timestamp, making it possible to directly compare the timestamps of remotely collected PMU data frames without the need or any form of distributed ordering algorithm.

2.2.2 Middleware

Middleware, [1], comprises a class of software technologies that simplify the development of distributed systems in heterogeneous software, hardware, and network communication environments. Middleware exists as a software abstraction layer between an operating system and an application. The abstractions that middleware provides to an application developer remove a measure of the complexity involved in the heterogeneity found in many distributed systems. This is accomplished by providing some or all of the following functionality:

- **Spatial Decoupling** removes the need for an application developer to be aware of the network layer address of a machine it wishes to communicate with.
- **Temporal Decoupling** allows messages to be delivered between participants even if they are not both online at the same time.

- **Remote Procedure Calls** allow an application programmer to execute code on a remote machine in that same manner they would execute code locally.
- **Distributed Objects** allow object oriented programming paradigms to be applied to distributed systems programming, by allowing application programmers to both access objects throughout a network as though they existed on a local machine.

All of the previously mentioned features of middleware simplify the process of developing distributed applications. While it is certainly possible to implement all of the functionality provided by middleware within every distributed application, middleware helps focus application development on actual application logic rather than communication concerns. GridStat, described later in this chapter, provides these features to application developers concerned with creating applications for critical infrastructures, like the power grid.

2.2.3 Publish-Subscribe Middleware

Publish-subscribe middleware is a specialized type of middleware designed for situations when the participants in a distributed system closely fill the roles of either a publisher or subscriber. In this context, a publisher refers to an application that periodically provides information to a network. A subscriber, on the other hand, is an application that requests the delivery of published items. In general, there will be more subscribers than publishers. Therefore, such networks benefit greatly from multicasting

and other technologies for bandwidth conservation. Examples of publish-subscribe systems include video distribution networks, stock tickers and weather updates.

2.3 GridStat architecture and mechanisms

GridStat, [6], is a flexible communication framework for delivering periodic status updates between equipment operated by entities such as grid operators, energy suppliers, and marketers. Its architecture defines the relationships between GridStat entities. The architecture is designed to provide the flexible technical underpinnings needed to meet diverse status communication needs of the different kinds of business entities. At a more detailed level, GridStat's mechanisms encapsulate reusable communication paradigms such as multicast, filtering, QoS management, and redundant path routing. These are quite naturally part of the communication infrastructure and would be very difficult or impossible to provide without such built-in support.

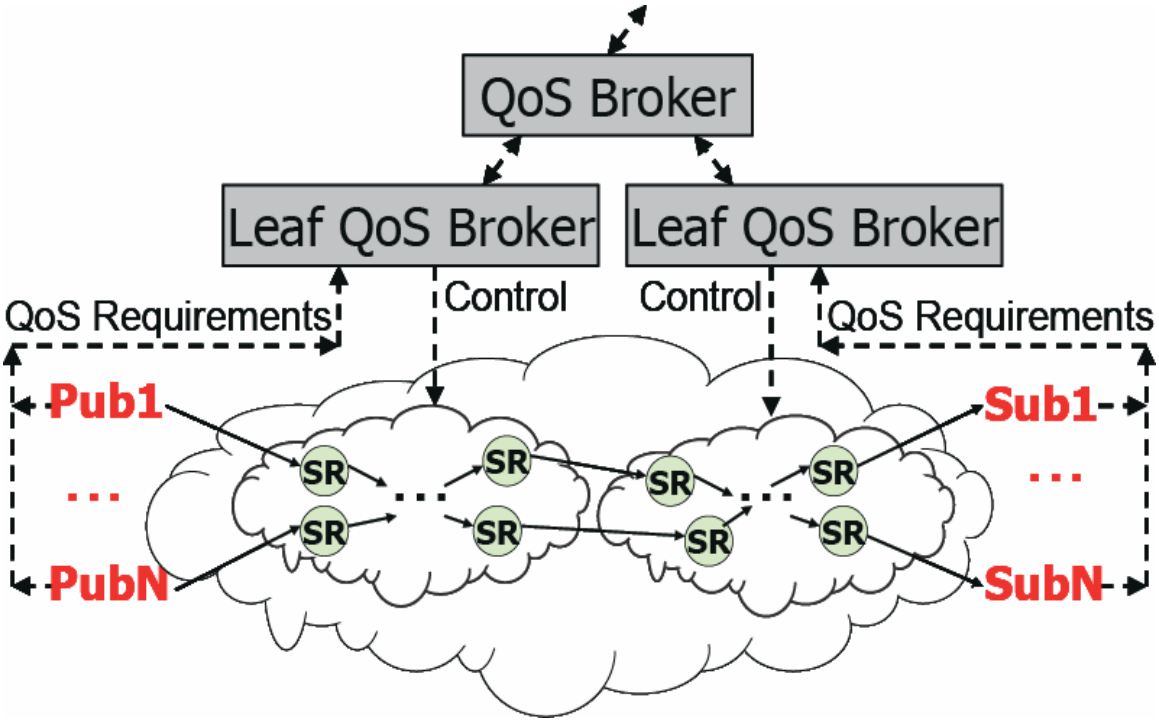

Figure 2.1: GridStat Architecture

2.3.1 Architecture

The GridStat architecture encompasses four primary types of active entities as shown in Figure 2.1. Publishers produce periodic streams of status values. Status routers accept status values from publishers and other status routers and forward them to other status routers and subscribers. Subscribers run applications that use (or store) the data. Publishers, status routers, and subscribers make up the GridStat data plane. The data plane's job is to move data packets quickly from publishers to subscribers. The GridStat architecture supports independent ownership and control of data plane components and anticipates that groups of components under common control will work together more closely than will components that are controlled by different business entities.

The fourth type of active GridStat entity is the QoS broker. QoS brokers are arranged in hierarchies that reflect geographic and business hierarchies. The QoS broker hierarchies constitute the management plane of GridStat. The lowest-level QoS brokers (leaf QoS brokers) each directly manage resources within a set of commonly-controlled data plane components. QoS brokers are responsible for establishing the route(s) through the status routers used by each subscription so as to ensure that each subscription's QoS requirements are met.

GridStat simultaneously provides a hierarchical management model and a flat delivery model. QoS brokers are not involved in packet-by-packet delivery: they merely direct the set-up of the status routers and monitor their performance, providing adaptation ability if status routers or links fail.

2.3.2 Status router mechanisms

Status routers include several mechanisms of use for PMU-based applications of which efficient multicast, synchronous filtering, packet combining and preconfigured modes are of the most interest for PMU applications. These mechanisms and their implementations are fully described in [6]. Here we provide only a short synopsis.

Efficient multicast

A major goal of GridStat is to allow sharing of status data. Multicast allows sharing without requiring multiple copies of each data item to traverse communication links as would be required if only unicast were provided. This makes better use of communication resources and reduces the latency seen by most subscribers.

Rate filtering

It is clear that not every subscriber needs every update produced by the publishers to which it subscribes. Rate filtering allows each subscriber to indicate, as part of its subscription, the rate at which it needs updates. The status routers implement rate filtering in conjunction with multicast and only forward packets that are desired by downstream subscriptions. Rate filtering is performed on the basis of timestamps contained in the packets—not the local clock of the status routers.

One of the biggest advantages PMU data has over conventional SCADA data is that it is synchronously timestamped. By collecting measurements from various points taken at the same time an application can derive properties of the grid's state that are not available from data taken at different times. On the other hand, it is clear that some applications for PMU data will not require all of the samples taken during each second. Synchronous filtering allows an application to subscribe to every n^{th} element of each of several PMU data streams and receive samples bearing the closest timestamps available to GridStat. (The question of whether or not the timestamps are the same depends on the PMU implementation.)

Packet combining

Status routers combine status packets into network layer 3 packets when it is possible to do so without incurring additional latency. This reduces layer 3 bandwidth overhead. More importantly it reduces the execution time overhead of both the sending and receiving status routers by reducing context switches.

2.3.3 Status packetization

The GridStat prototype uses a simple packetization scheme with each packet consisting of a basic 24-byte header (which includes 8 bytes of user data) and a variable number of 24-byte data extensions. A GridStat packet contains

- 8-byte timestamp (milliseconds since Jan. 1, 1970)
- 4-byte publication ID
- 1-byte number of optional fields
- 3-bytes padding
- 8-bytes user data
- 0 to 4 24-byte optional user data fields

GridStat packets are encapsulated in Internet Protocol User Datagram Protocol (IP/UDP) packets in the prototype which uses Internet technology as its link layer.

CHAPTER THREE

RELATED WORK

3.1 Power Systems Research

3.1.1 Wide Area Measurement System

The Wide Area Measurement System (WAMS) on the U.S. Western Interconnection uses PMUs to gather data from several points in the interconnection to allow off-line analysis of the grid's performance, [18]. The data gathered has been instrumental in understanding outages on the system such as those occurring in July and August, 1996, [8]. WAMS has inspired a number of commercial offerings such as those of e.g. [13, 15]. Several applications can be deployed given global knowledge of the grid [13].

Corridor Voltage Stability Monitoring

This type of monitoring involves examining measurements from both ends of a transmission corridor. Using the measurements, coupled with knowledge about any load or generation in the corridor, analysis can be performed to determine the theoretical maximum load that may be placed on the corridor, as well as the margin to voltage instability. Such monitoring could be used to make control decisions leading to load shedding to avoid voltage collapse when the load on the corridor becomes excessive.

Oscillatory Stability Monitoring

Oscillatory stability monitoring is achieved through an algorithm that examines measured date from one or more locations. The algorithm is capable of making predictions about the frequency and dampening of the dominant electro-mechanical oscillatory modes during normal operation.

FACTS Set-Point Optimization using Feedback Control

A FACTS device is a network controller used to control bus voltages, line currents or phase angles [14]. Unlike other network controllers in this class that utilize mechanical switches that require several minutes to operate, FACTS devices use power-electronics which allows response times in the millisecond range. These devices were primarily designed to improve transfer capability of transmission corridors, and to keep transmission corridors running near to capacity. Because of their quick response time, they may also potentially be used to implement control decisions of wide-area control application. It is approximated that by using wide-area control coupled with FACTS devices, 10% higher total transfer capacity can be achieved compared to using FACTS devices alone.

Topology Detection and State Calculation

Using PMU data collected from throughout the grid 10 to 20 times per second, this technique can detect the present topology of the network. In addition, phasor state-estimation can be completed in predictable time, because it is a non-iterative process.

Loadability Calculation

This is a more advanced form of voltage stability assessment which is derived from the information derived from the topology detection and state calculation application. Using this technique, the maximum transfer capacity of the currently operating power system can be determined.

3.1.2 BPA Phasor Data Concentrator

In 1988, the Bonneville Power Administration (BPA) first deployed phasor measurement units (PMUs) in the USA Western System (WSCC). Subsequently in 1995, the first commercial PMUs were installed in the WSCC as part of an EPRI project. These PMUs were active in collecting data in the 1996 blackout [16]. This was the first time PMUs were able to capture data about a significant outage. We will be examining the Phasor Data Concentrator (PDC) used by the BPA to collect this data.

PDC Definition

A PDC collects data from multiple sources including both PMUs and other PDCs for the purpose of combining these data feeds into a single measurement set. Once collected the data is valuable in multiple applications including both power system monitoring and control. Generally, PDCs provide mechanisms for coping with data loss, errors and synchronization. A PDC can be used for both data recording and for distributing data to other applications.

BPA Implementation and Applications

The BPA PDC implementation uses standard VME hardware, which is intended to provide ready availability and high reliability.

The BPA PDC acts as the primary system monitor. It is responsible for examining data received from PMUs for transmission errors and losses. It was designed to detect spurious alarm messages from PMUs and discard them. In addition to these primary tasks, the PDC monitor provides its examined data to external applications.

The BPA PDC implementation also provides a data display application runs on a PC, and is designed to display archived PMU data from a PDC in a user readable format. In addition, this data display program provides facilities for data export to popular data analysis software packages like Matlab.

Phasor real-time display and recording is another application provided by the BPA PDC. This application provides a continuous strip-chart type display of real-time phasor data. At the time of the BPA experiments, the real-time control portion of the PDC was still being tested, and had not been used in practice. The BPA had developed a method for using feed-forward voltage control to prevent voltage collapse. This is accomplished by inserting capacitors and SVC units to prevent the collapse.

3.1.3 Eastern Interconnection Phasor Project

Currently, the Eastern Interconnection Phasor Project is deploying PMUs on the Eastern U.S. grid. EIPP is developing a communication infrastructure to collect and archive the measurements at the TVA Super PDC, [2]. In addition to archiving, the TVA

site makes a feed of all the collected data available to participating utilities within a few seconds after it is collected.

3.1.4 FNET Project

The FNET project is exploring uses for a large number frequency monitors throughout the North American grid, [15]. Frequency monitors are similar to, but simpler than PMUs. FNET uses the Internet, without management, to collect data for analysis.

3.1.5 The Wide-Area Stability and Voltage Control System

The Wide-Area stability and voltage Control System (WACS) project of the Bonneville Power Administration, Ciber, Inc. and Washington State University uses PMUs as sensors in a real-time system for stability and voltage control, [23]. Calculations from that project provide a time budget for creation, delivery and processing of PMU measurements in the Western Interconnection. The budget is based on the need to respond to a disturbance within 750-1500ms. Of this time budget approximately 67 ms is attributed to communication latency and jitter using the project's current point-to-point communication infrastructure. WACS uses a dedicated communication network to collect input from its sensors. Other projects using networked PMUs for control include [21] and [11].

3.2 Centralized Systems

The TVA Super PDC mentioned in the section 3.1.4 is an example of a centralized system for PMU data distribution, but most of the current solutions employ a

similar architecture. We feel that this is not the optimal solution for a PMU data distribution network. In the Super PDC, phasor data is collected from all of the various sources throughout the grid, and grouped by synchronized timestamps. After this grouping, the data is retransmitted out to any locations that wish to make use of the real-time data.

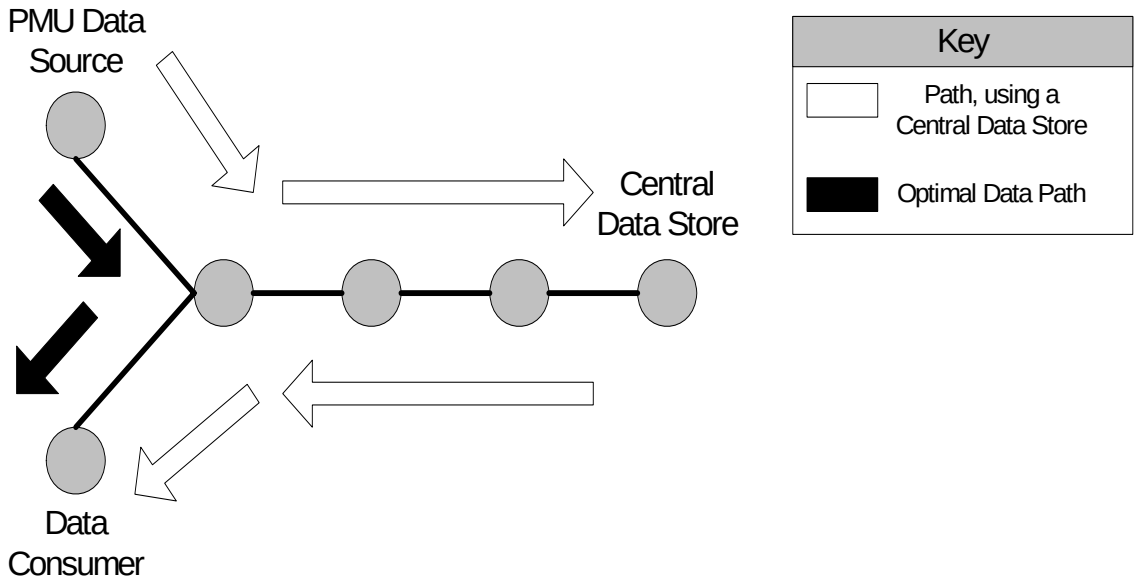

Figure 3.1: Centralized Systems

Such a system has several shortcomings. Clearly a centralized solution leads to questions about reliability due to the use of a single point of failure. Perhaps more significantly is the increased latency experienced by data consumers as illustrated in figure 3.1. The white path incurs significantly more network hops to before reaching its destination. On the other hand, the black path shows the optimal solution.

The value added by a centralized PDC is in synchronizing the data for delivery to consuming applications. Any alternative solution would have to be able to provide this functionality with out collecting all of the data centrally.

3.3 Middleware Research

3.3.1 Data Distribution Service

The Data Distribution Service (DDS) for Real-time Systems Specification by the Object Management Group addresses communication of signals, streams and states, [19]. Existing implementations of the specification are targeted to the LAN environment and do not address QoS management issues associated with the wide area, [20, 24].

3.3.2 PASS System

The PASS system was designed for monitoring the status of a communication network, [26]. It does address QoS in the wide area but does not provide independent QoS specification for each subscription.

CHAPTER FOUR

USE OF GRIDSTAT MECHANISMS FOR PMU DATA DISSEMINATION

While PMU data is valuable in many applications, the number of actual packets required per second varies depending on the application. GridStat provides the framework to allow a PMU to publish its data as fast as it can provide it, but only those packets that are required by subscribing applications are actually delivered. For instance, if a PMU is publishing at a rate of 60 times per second, and a subscribing application requests 1 packet every 2 seconds, only a single PMU data frame will traverse the GridStat network every other second. This middleware level packet filtering assures application developers that network traffic is only that which is needed to fulfill their particular application's requirements.

Packet filtering is useful for minimizing the use of network resources, but many applications require data from multiple PMUs to be effective. Such applications require that the packets delivered share a common timestamp, regardless of what path they take through the network. As mentioned in section 3.2.2, GridStat's synchronous filtering fulfills this requirement while preserving the network traffic reduction of packet filtering.

To support the collection of PMU data from multiple locations, GridStat provides an interface for specifying a set of publications for subscription. In this way, the entire set may be accessed from a subscribing application as a single unit rather than as a series of individual subscriptions.

4.1 Using GridStat

One of the primary sections of research in this thesis was the investigation of deploying a real-world application, PMU data dissemination, using GridStat. GridStat was always designed to function as a communication medium for the power grid, but this is the first time that an attempt has been made to use real power grid data in GridStat. Pervious research involved moving arbitrary data, and focused more on showing the that certain QoS requirements of the power grid could be met.

The research done in this thesis did not involve direct work on the GridStat architecture, but rather on the development of application code utilizing GridStat. The development of this application code led to a better understanding of the issues involved in utilizing GridStat for the transmission of real-world data. These lessons learned led to several small changes in GridStat, but that was not the focus of our research.

4.2 Key Issues

Two key issues arise in encapsulating PMU frames in GridStat packets: timestamp mapping and size overhead.

4.2.1 Timestamp Mapping

For timestamp-based filtering to be useful in PMU applications each packet timestamp must match the PMU timestamp of the encapsulated PMU frame. Packet timestamps refer directly to a global epoch. Timestamps in PMU frames (second of century || sample number within second), are referenced to a globally synchronized clock. However, because the number of samples per second is part of the PMU configuration,

mapping a PMU timestamp to a GridStat timestamp requires knowledge of the PMU configuration—something that is not part of the PMU data packet. It is therefore necessary for the PMU publisher to interpret the PMU timestamp and construct the GridStat timestamp.

4.2.2 Size Overhead

A base GridStat packet contains a 16 byte header, and 8 bytes for user data. In addition, GridStat allows up to 4 option fields for additional user data of 24 bytes each. Because the GridStat packets expand by 24 bytes at a time the question of overhead due to internal fragmentation must be considered. For example, a PMU reporting 3-phase voltage and current along with 32 digital statuses (42-byte PMU frame) the corresponding GridStat packet is 72 bytes of which 14 are padding and 16 are packet header. A graph of the percentage of waste in a GridStat packet due to this overhead is examined in figure 4.1.

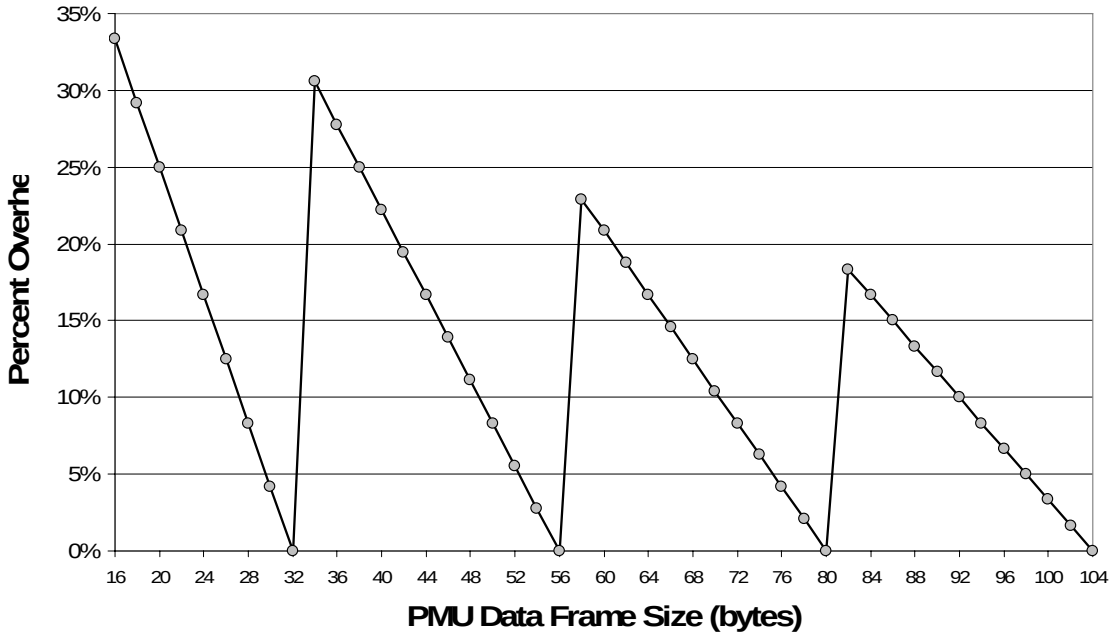

Figure 4.1: GridStat Packet Overhead

4.3 Experiments

To demonstrate the suitability of GridStat to meet the data transmission requirements of PMUs, we developed two sample applications. A virtual PMU data publisher emulates a publisher attached to a real PMU, as it produces data of the same size and at the same rate. A virtual phasor data concentrator (PDC) subscribes to all the emulated PMU data. A PDC is one of the most data intensive applications for PMU data, as it collects data from all of the PMUs in a grid. We surmised that if we could support the demands of a PDC, we should be able to support the demands of any other PMU data applications.

4.3.1 PDC requirements

We set out to show empirically that GridStat can fulfill the data communication needs of a PDC. We used the recommendations of the EIPP real time task team, [17], as a basis for our evaluation of GridStat. Those requirements specify that a PDC should be able to receive each PMU's data 30 times a second. Further, to be useful for real-time monitoring, data must be collected from at least twelve different sources. For real time adaptive control applications the communications infrastructure is expected to add approximately 10ms to the overall time delay of PMU data delivery. To meet short-term needs, therefore, GridStat should be able to provide a PDC data from at least 12 PMU publishers at a rate of 30 per second while contributing less than 10ms of delay. In the longer term there will be many more PMUs on the grid so tests were done with 100 PMU data streams.

4.3.2 Experimental setup

To illustrate that GridStat can meet the stated requirements we conducted four experiments. All of the experiments ran 100 separate publishers, publishing emulated PMU data frames either at 30 or 60 times per second. The PMU data frames were composed of six phasor fields and 2 digital channel fields which made the PMU frame size 42 bytes. With the additional GridStat header information and padding, the packet size was 72 bytes. In all four experiments, a single virtual PDC subscribed to the entire set of 100 published data items at a rate either of 30 or 60 times per second. All 100 publishers as well as the PDC were run on a single computer. This allowed use of the local clock to calculate end-to-end delay without concern for clock synchronization. All

of the computers used in these experiments were current home-PC-class machines running Linux, and connected by a switched 100Mbps Ethernet.

Single Path Delay Jitter

This pair of experiments used three status routers: two edge status routers connected by a single internal status router as shown in Figure 4.2. (The difference between edge and internal status routers is that edge status routers provide interfaces for publishers and subscribers.) Each of the routers ran on a dedicated computer. All 100 publishers were connected to one edge status router and the PDC was connected to the other.

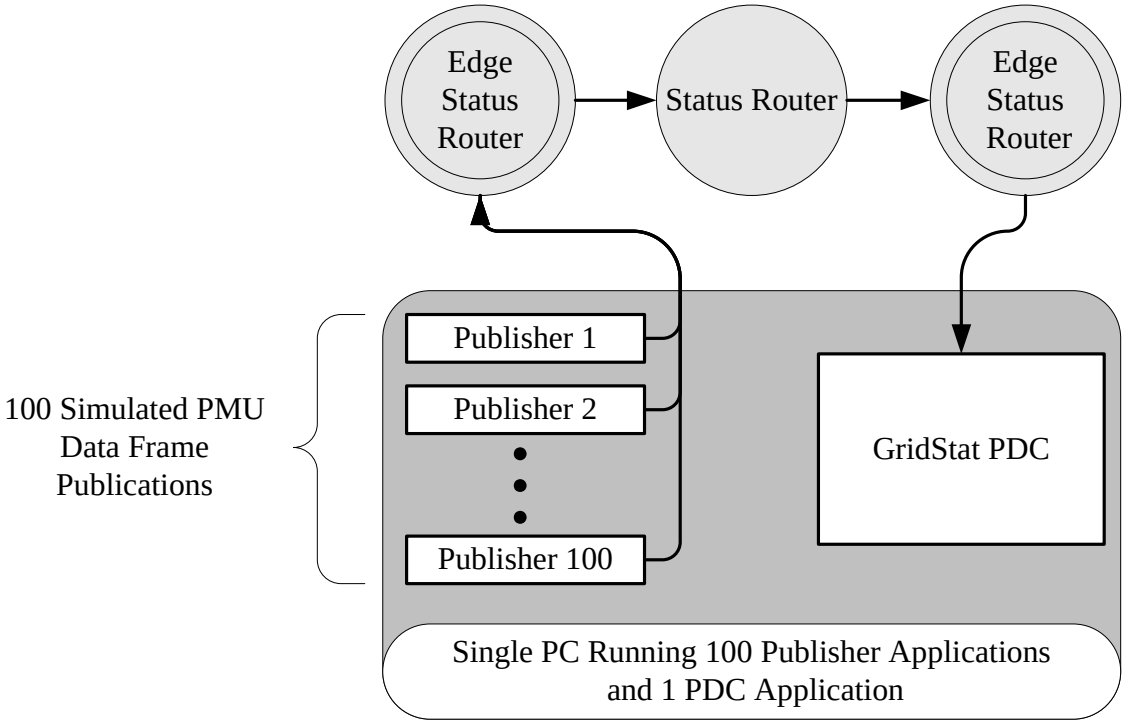

Figure 4.2: Single Path Configuration

Redundant Two-Path Delay Jitter

These two experiments were identical to the previous except for the addition of a second internal status router that independently linked the two edge status routers as shown in Figure 4.3.

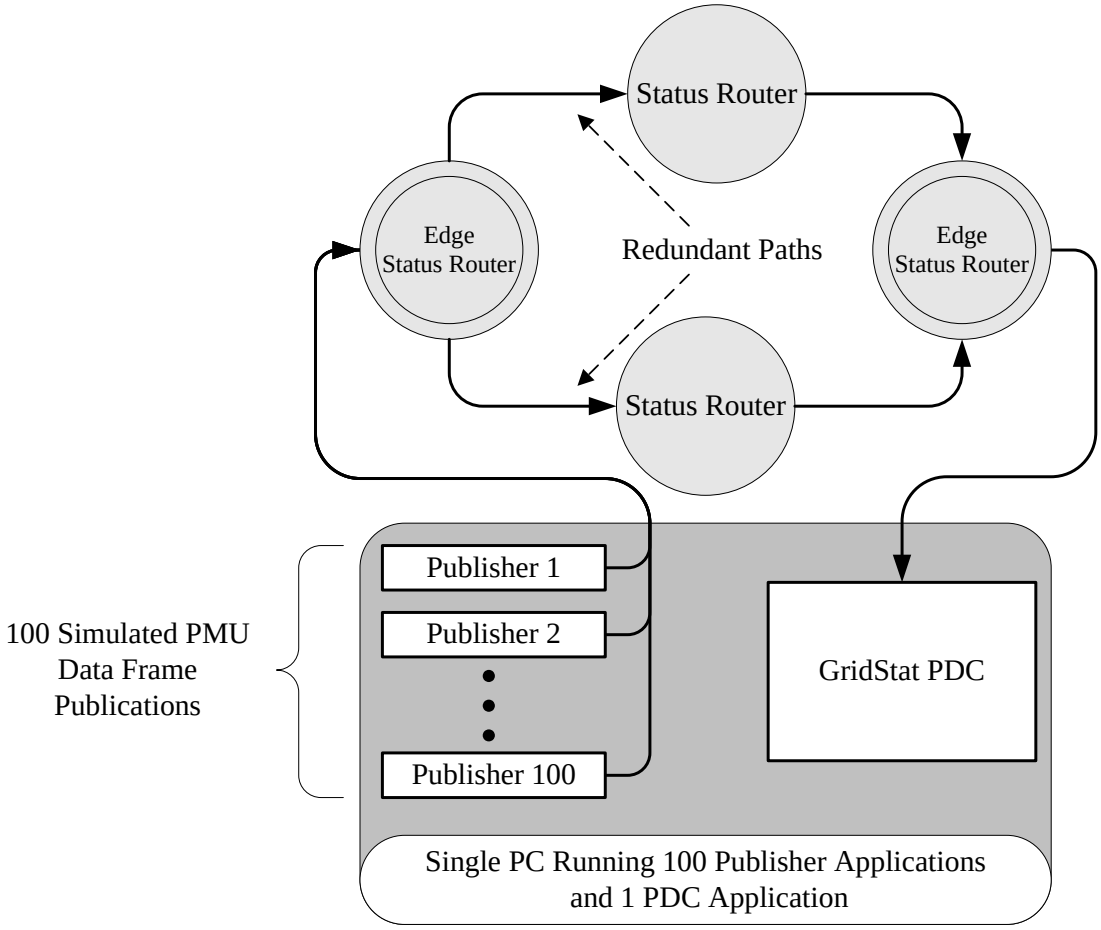

Figure 4.3: Redundant Path Configuration

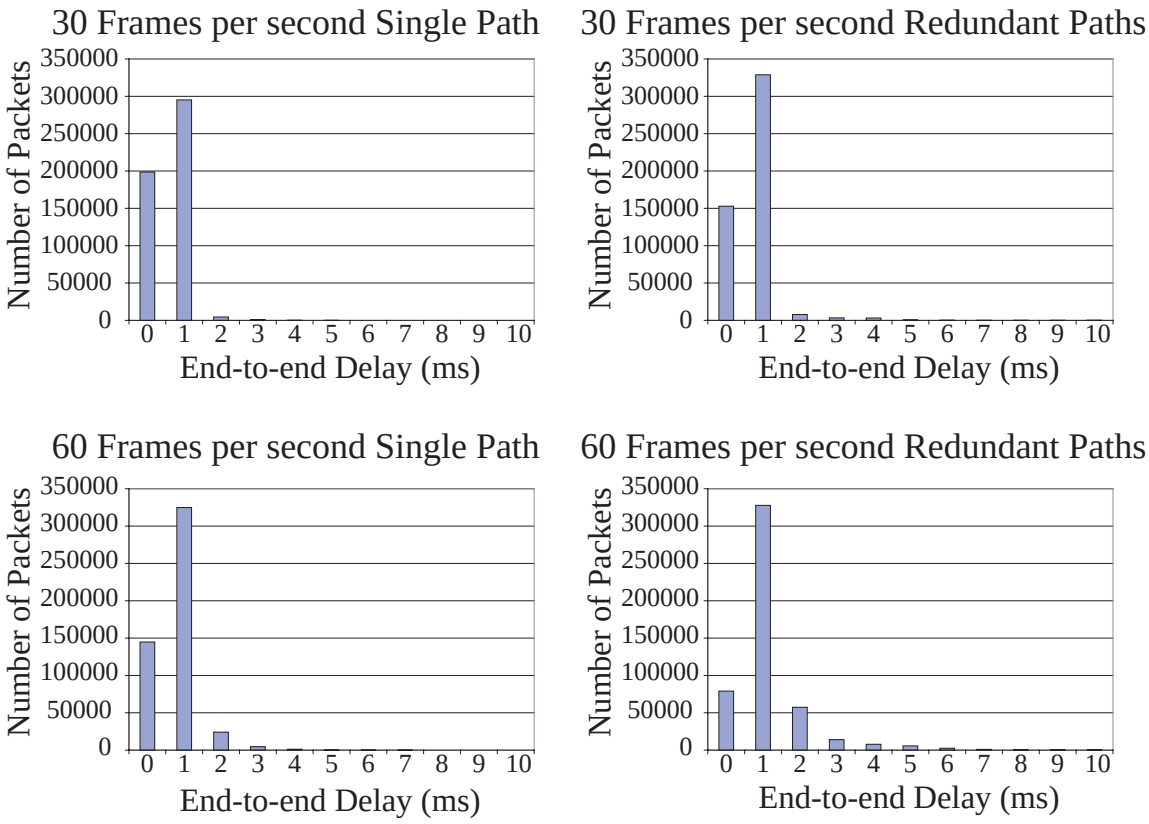

Figure 4.4: End-to-end Delay

4.3.3 Experimental results

Figure 4.4 and Table 4.1 present the results of the four experiments. In each experiment 500,000 PMU data frame transmissions were evaluated for end-to-end delay. In all of the experiments all of the frames that were sent were delivered.

Experiment	End-to-end over 10ms	Percentage
30/s Single Path	173	0.034%
30/s Redundant Paths	2527	0.505%
60/s Single Path	108	0.022%
60/s Redundant Paths	3808	0.762%

Table 4.1: Delays Over 10ms

In all four experiments the vast majority of PMU data frames were delivered in less than 2ms (bars 0 and 1 on the graphs), and over 99% were delivered in less than

10ms. End-to-end delays above 10ms, summarized in Table 5.1, indicate delay jitter attributable to use of a general purpose operating system, and Java with garbage collection. Increased delay and jitter when using redundant paths is attributable to the additional work performed at the edge routers.

We do not feel that the delays we experienced can be attributed to using a shared switched Ethernet rather than true point-to-point links. In the case of sixty data frames per second, the data produced by the publishers is only 3.456 Mbps, which is well below the 100 Mbps limit of the Ethernet.

These experiments generally support the feasibility of using a multi-casting based infrastructure for PMU status dissemination. The phasor data requirements document from EIPP, [17], does not specifically address acceptable amounts and frequencies of jitter. We believe that both the jitter and latency observed in these experiments could be significantly reduced in an implementation that did not use Java. Of course this requires confirmation.

Further experiments are also needed to establish the number of routers that can be in a path while still meeting the delay and jitter requirements. Other GridStat experiments have shown a delay penalty of less than 1ms per additional router, [6]. Propagation time between widely-separated power grid entities must also be added.

4.4 Conclusion

Using GridStat for PMU data dissemination was not unmanageable, but it was not designed from the ground up with the PMU data format specifically in mind. This led to a few application development challenges that could have been avoided if GridStat was

tuned to the transmission of PMU data. If GridStat is to be used primarily for PMU data, a few changes to GridStat seem reasonable. For instance, the GridStat data and option fields could be tuned to better reflect the sizes PMU data frames. It would also be useful to provide a software component for GridStat publisher application developers that recognizes common configuration files, like the IEEE 1344 PMU configuration format, and handles the timestamp mapping automatically.

Overall, as a middleware system, GridStat provides an abstraction layer that removes many of the complexities of distributed systems from the application developer. In addition, as will be mentioned in chapter five, GridStat provides an application development framework to help streamline the process of creating applications using GridStat. This allows applications to be developed quickly and easily, even by individuals who may not have a background in distributed systems.

CHAPTER FIVE

APPLICATION DEVELOPMENT

5.1 Overview

This section describes the three programming projects that were involved in the completion of this thesis. Two applications were developed, a *Phasor Measurement Unit Emulator*, and a *Phasor Data Concentrator Emulator*. These applications were deployed to evaluate GridStat's ability to handle data flows identical to that which would be experienced in a field deployment. The third project developed was an application framework that simplifies the development of GridStat applications. Both of the applications developed utilized this framework.

5.2 Phasor Measurement Unit Emulator

The PMU emulator emulates the output of a real PMU. It is a GridStat publisher that publishes a user defined type encapsulating a standard PMU data frame as defined in [10].

Relevant features of an emulated PMU include:

- Run-Time configuration of PMU data frame size. This is useful because the standard allows for a variable number of phasors and digital status data fields.
- Run-Time configuration of publication rate as defined in number of milliseconds between packets.

5.3 Phasor Data Concentrator Emulator

The PDC emulator is intended to show how GridStat can be used as a communication medium connecting PMUs to PDCs. The only thing that makes the PDC emulator an emulator is that it merely collects data from PMUs, but it does nothing with that data. A real PDC would provide its collected data to other applications and/or archive the data for offline analysis.

5.4 GridStat Java Application Framework

5.4.1 Motivation

GridStat was designed to ease the development of distributed applications for critical infrastructures like the power grid. In that regard, GridStat goes a long way towards providing advanced distributed systems features, while insulating application developers from all of the details in their implementation. However, we felt that perhaps GridStat could benefit from an additional software library specifically designed to ease the application development process. Just as GridStat removes some of the complexities of distributed application development, the GridStat Java Application Framework hides many of the mundane and trivial details of building Java applications by providing a class library that is tuned to GridStat application development.

During the development of GridStat, it has been necessary to deploy several prototype applications to test and evaluate GridStat. After the development of several of these prototypes, it became clear that all of these prototypes shared a great deal of common application code that was neither interesting nor dynamic. In particular a great

deal of code was involved in making applications that could interface easily with external scripts, as well as providing a useful graphical user interface. In the interest of making the development of GridStat applications easier and faster, a class library of common application code was created.

5.4.2 Overview

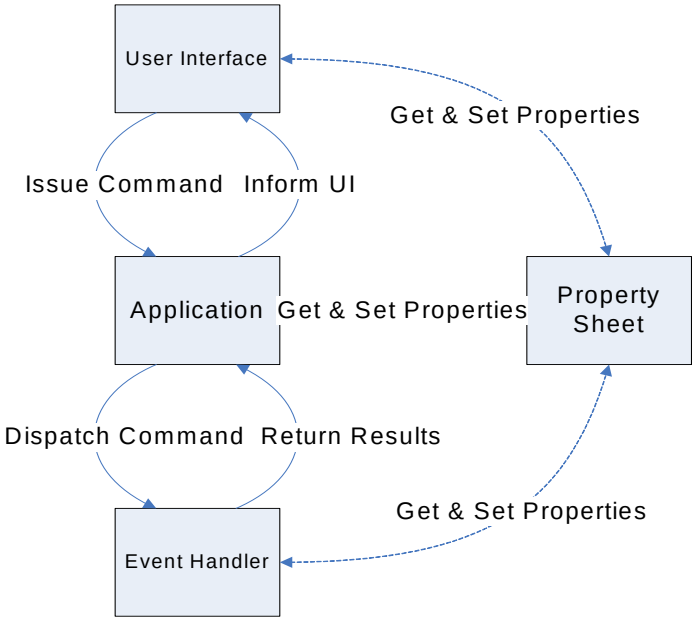

Figure 5.1: Application Diagram

The GridStat Java Application Framework is a set of base classes that ease the development of a GridStat application. The framework is divided into four major modules, as shown in Figure 5.1. The *User Interface* component segregates user interface code from application logic code. The *Application* component is the metaphorical glue that holds the other pieces of the application together. The *Event Handler* portion of a program is responsible for passing command requests to an actual GridStat component. The *Property Sheet* is a globally accessible hierarchical repository of key value pairs that

is intended to contain an application's data. An application developer need only extend the base classes to add that functionality which is required for their particular application.

Message passing between the user interface, the application, and the event handler is done with Java ActionEvents. The application acts as a Java ActionListener, listening to events generated by the user interface. The application then examines the ActionEvent, and chooses a course of action, which most commonly involves dispatching the ActionEvent to the event handler component. If the dispatch to the event handler succeeds, the application passes the ActionEvent to the user interface to inform the user that their requested action was successfully performed. If the dispatch fails, the application still informs the user interface, but in this case it calls a different method to provide the user with the error information. Because the messages are simply command strings, more complex data exchanges are performed by reading and writing to the property sheet, which is accessible by all of the components.

5.4.3 User Interface

The user interface (UI) component exists to segregate user interface code from application logic code. The architecture is such that it does not matter what kind of user interface exists. Base classes have been created to handle the cases of both a command line interface, and a graphical user interface. Other styles of user interaction are allowed as long as they can implement the specified java interface.

Of note is the fact that a given application can have multiple user interfaces, but only one may be registered with the application component to receive notification of successfully executed commands, and error messages. This registered user interface may

be thought of as the primary user interface. Other user interfaces merely pass their commands to the application component with no expectation of a response. In many existing GridStat applications this facility is used to provide a supplementary command line interface, while using a graphical user interface for the primary user interface. The limitation of only one primary UI is arbitrary, and the base application component could be extended to support an unlimited number of primary UIs.

Command Line Interface

The base class implementation of the command line interface can stand alone as the sole user interface for any GridStat application. This base implementation merely reads lines from the standard input, and passes them to the application. Properly formed commands will be handled, and improperly formed ones will fail with an appropriate error message. This base class can easily be extended to provide more complex functionality.

Graphical User Interface

The base graphical user interface is an extended Java Swing JPanel. The intention is that this component contains all of the code necessary to set up the GUI, and register its GUI components with the property sheet component. Upon receipt of successful command messages, the GUI may update its appearance to reflect the success of the command. It is important to note that if the GUI is setup as the primary user interface, it will receive all successful command messages, even if they did not originate from the

GUI. In this way, the GUI can be updated to reflect commands executed from secondary user interfaces.

5.4.4 Application

The base application class contains all of the necessary code to route messages from the various components, and only needs to be extended to allow the application developer to specify which type of components they are going to be using. In addition, the base application has the ability to pass initialization parameters from the command line to the property sheet.

The base application class also contains a method for registering objects for cleanup on exit. All such objects are instances of a class that implements a particular Java interface that contains a method containing all of the cleanup code for that class. When the base application receives an exit command, it will ensure that the cleanup method of all of its registered cleanup objects is called.

5.4.5 Property Sheet

The property sheet is a globally accessible interface to a repository of data intended to represent an application's state. Generally it is used to store the data values that will be used to initialize GridStat components activated by the event handler component. It is self-contained and unlike the other components, it is not designed to be extended. It is hierarchical in nature, and is intended to be composed of hash tables for sparsely distributed data, and vectors for densely packed data. For simple applications, it is unlikely that the property sheet will need to store anything more than name-value pairs.

However, when more complex data sets are needed (as is often the case in applications with multiple publications and/or subscriptions), the property sheet becomes a tree like data structure.

As an example, consider a simple GridStat status router. A status router needs two parameters. The first being the name of its leaf QoS broker, and the second being its own name. For such a simple example, the name value pairs shown in Table 5.1 would suffice. In this example, the event handler doesn't need the variables storing the configuration information. Instead, the event handler expects that it can find the variables by looking up their key within the property sheet. This adds a level of indirection that allows the developer of the event handler to be unconcerned about how the variables arrive in the property sheet, and only be concerned with the property sheet directly.

Initilization String	
"QoSBrokerName=CloudA.SubCloudB"	
"Name=StatusRouter12"	
Key	**Value**
QoSBrokerName	"CloudA.SubCloudB" (String)
Name	"StatusRouter12" (String)

Table 5.1: Leaf QoS Broker Example

To consider a case where a hierarchy is helpful it is useful to look at a subscriber application that is designed to subscribe to multiple events. The subscriber would have a set of parameters to specify its name, its leaf QoS broker, and its edge status router. However, in addition each subscription would have a set of parameters specifying its group name, its publisher, its status name, its data rate, its maximum allowed latency, and the number of redundant paths required.

Initilization String	
“QoSBrokerName=CloudA.SubCloudB”	
"Name=Subscriber3"	
"EdgeStatusRouterName=EdgeStatusRouter1"	
"Subscriptions={{GroupName=G1,Publisher=Pub1,StatusName=S1,DataRate=20,Latency=300,Redundancy=0},{GroupName=G1,Publisher=Pub2,StatusName=S2,DataRate=20,Latency=300,Redundancy=0}}"	
Key	**Value**
QoSBrokerName	“CloudA.SubCloudB” (String)
Name	“Subscriber3” (String)
EdgeStatusRouterName	“EdgeStatusRouter1” (String)
Subscriptions[0].GroupName	“G1” (String)
Subscriptions[0].Publisher	“Pub1” (String)
Subscriptions[0].StatusName	“S1” (String)
Subscriptions[0].DataRate	20 (Integer)
Subscriptions[0].Latency	300 (Integer)
Subscriptions[0].Redundancy	0 (Integer)
Subscriptions[1].GroupName	“G1” (String)
Subscriptions[1].Publisher	“Pub2” (String)
Subscriptions[1].StatusName	“S2” (String)
Subscriptions[1].DataRate	20 (Integer)
Subscriptions[1].Latency	300 (Integer)
Subscriptions[1].Redundancy	0 (Integer)

Table 5.2: Subscriber Example

Notice that in Table 5.2, the “Subscriptions” initialization string contains curly braces around comma delineated lists to denote that the “Subscriptions” key points to a value that has sub-values. The property sheet component contains the code to parse initialization strings into actual keys and values. An alternative initialization method would be to specify each subscription separately (“Subscription[0]={GroupName=G1,Publisher=Pub1,StatusName=S1,DataRate=20,Latency=300,Redundancy=0}”), or every sting key/value separately (“Subscription[1].Latency=300”). The parser is designed to accept all of the previously mentioned input styles.

Variable Adapters

The property sheet is a repository of data items that extend a base class referred to as a variable adapter. A variable adapter wraps a real data object, and provides methods for setting and getting the value contained in the object. This is particularly useful for data contained in GUI components. For example, an adapter can wrap a set of radio buttons, so that when the wrapper is queried for its current value, it returns the currently selected radio value. In that same example, if the adapter is set, it can contain all of the relevant code for updating the GUI to reflect the change.

Variable adapters are intended to be of a particular variable adapter type, which currently includes: String, Integer, Float, Boolean, Vector and Map. The Vector and Map variable adapters are container types which contain other variable adapters. They are required to provide additional interfaces for retrieving the variable adapters they contain.

During the property sheet initialization, command line parameters are converted into basic adapters. This property sheet initialization happens before the UI component is initialized. If the UI is implemented as a GUI, it can register its components as variable adapters with the property sheet during the UI component's initialization. If an adapter with the same key already exists (because it was initialized from the command line), the new adapter is updated to contain the value contained in the old adapter. In this way, GUI components can be initialized at run-time to a particular default value. This simplifies the process of starting a GridStat application.

The framework comes with a simple variable adapter implementation of each of the basic types that is simply a wrapper around a Java variable of the same type. In

addition to these simple variable adapters, there is also a set of variable adapters for some basic Swing components, and the values they would encapsulate. There are adapters for Swing text boxes that contain Integers, Floats, and Strings. There is an adapter for Swing checkboxes that represent Booleans, and an adapter for sets of Swing radio buttons that represents a String.

5.4.6 Features of a Base Application

One of the key advantages of developing applications using the GridStat Java Application Framework is that every application automatically possesses some useful features without writing any additional code. Those features are summarized in the following subsections.

Command Line Parameter Parsing

Key/value pairs passed to the application at run-time are collected and stored in the globally accessible property sheet. This feature makes it easy to create generic applications that can be configured for execution by writing start up scripts that pass relevant configuration information on start up. We take advantage of these features heavily during test runs where many instances of a particular application need to be started, and each instance has unique configuration settings.

GUI Generated Commands are Strings

Because GUI generated commands are strings, any command that could be executed by pressing a button on a graphical user interface can be executed directly using a command line interface. Because the framework allows a command line interface to run

concurrently with a graphical one, this presents some interesting abilities. During test runs we took advantage of this facility to start GUI applications with some preset commands. For example, we could write a script to pipe a few commands to the standard input of our application at startup that would cause it to initialize and start executing. Once the preset commands are read from the standard input, the application can be controlled manually through its GUI.

Base Command Set Recognition

The basic application recognizes and properly handles the basic commands listed in table 6.3.

Command	Default Action
Set <name>=<value>	Parse the name and value pair just as though it was passed from the command line, and store the result in the global property sheet. If the name already exists, set the variable adapter already in the property sheet to the new value.
Unset <name>	Remove the specified variable adapter from the global property sheet.
Env	Short for environement, this command displays the current contents of the global property sheet. This command is intended for users of command line interfaces.
Exit	Notify all registered components of system shutdown, and close application.

Table 5.3: Base Commands

All of the base commands are case insensitive, but that is not necessarily true of all commands. Because all advanced commands are passed to custom event handlers, it is the responsibility of each custom event handler to determine if it wishes to check for case.

5.4.7 Effectiveness of Framework

After having developed several applications that utilize the GridStat Java Application Framework and several that did not, we have obtained a feeling for its effectiveness in streamlining the production of GridStat applications. The shared application code base offered by the framework certainly has made it easier to develop the types of GridStat applications we envisioned. If an application does not require a GUI the framework removes nearly all code work from the application developer except the application specific code required to execute the desired task of the particular GridStat application. In the case that a GUI is required, the task of creating the GUI is reduced to simply laying out Java Swing components, as the framework is designed to take care of making fields in the GUI accessible to the application logic.

To convey an idea of how the application framework simplifies GridStat application development, we examine our implementation of the two emulator projects created for this thesis in the following subsections.

Implementing the PMU Emulator

While developing the PMU emulator, we were able to avoid a lot of the standard boiler plate code involved in creating a Java application of any complexity. Instead we were able to focus our efforts on writing the code to contain and publish a PMU data frame. While we decided to provide a GUI for this application, it was simple to put together, and contained no application logic in the GUI code.

Implementing the PDC Emulator

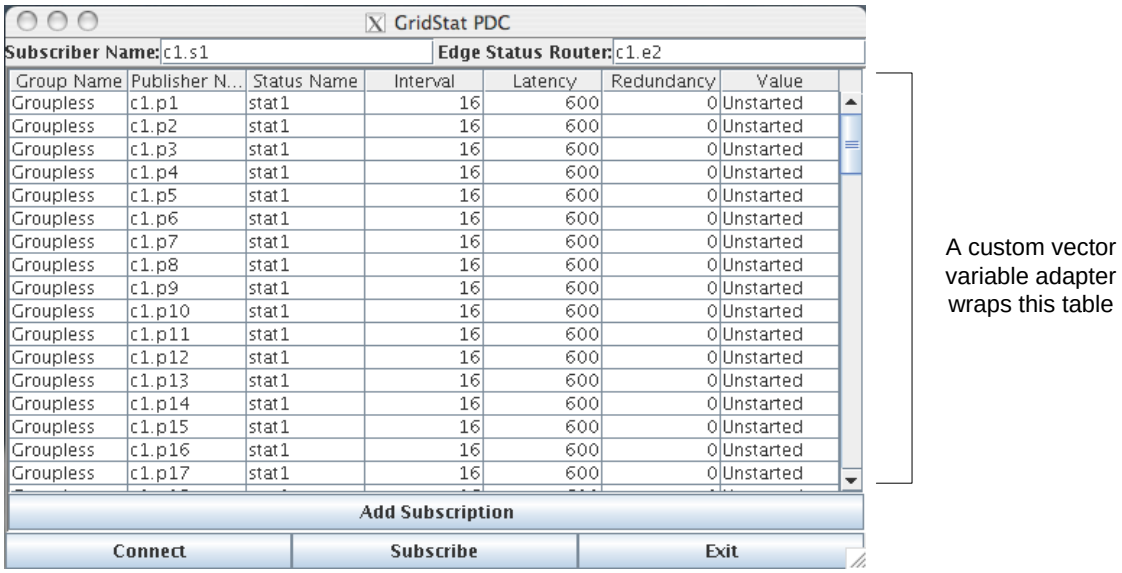

Figure 5.2: PDC Emulator Screen Capture

The PDC emulator was a much more involved application than the PMU emulator, because it was designed to be able to subscribe to any number of PMU publication feeds. As such, each of those publication feeds would have configuration information associated with it, and so we developed a GUI that contained a scrollable table component for collecting the configuration information about all of the various subscriptions as seen in Figure 5.2. Because this application contained a GUI component that does not have a predefined variable adapter in the framework, we had to create our own variable adapter. We created a variable adapter that treated the table as a Vector, and implemented the interface required for a variable adapter of type Vector. In addition, because the table was two dimensional, we made each of the variable adapters in the original Vector a Vector itself.

The two classes implementing the interfaces for a Vector variable adapter were all it took to make our PDC's GUI completely compatible with the global property sheet. Our event handler had complete access to the data entered into the table, and the table could be populated from the command line. Again, the framework made it possible to spend most of our coding work on the actual application logic, instead of mundane application details.

5.4.8 Conclusion

The GridStat Java Application Development framework was intended to speed and ease the development of GridStat applications. Our experience with using the framework for the development of the two applications developed for our experiments leads us to believe that the framework is effective. Our development work was able to focus on code specific to solving the problem, rather than things like user interaction, and scripting support. We believe that the full value of the framework will not be known until it has been involved in several more GridStat application deployments.

CHAPTER SIX

CONCLUSION

The prototype status router is implemented in Java. Care has been taken to avoid memory allocation (and hence garbage collection) when moving packets through the router. Beyond this, no unusual optimizations have been used. For production use a language closer to the hardware, such as C, or even implementation in specialized network processors, [4], should be considered. Nevertheless, the current Java implementation on current home-PC-class hardware is able to service large numbers of PMU data streams in comparison to the requirements specified by the EIPP real-time task team.

In the experiments reported in section 4.3, the packet combining feature described in section 2.3.2 played little role. A small amount of packet combining was observed during the experiments. It was not enough to be a major factor in the results of the experiments.

One of the mentioned goals of this thesis is to show that GridStat is suitable as the communication medium for PMU data. Using the requirements specified by the EIPP real-time task team, the experiments of section 4.3 show that GridStat is appropriate for this task. However, GridStat currently is unable to ensure end-to-end delay jitter is kept below 10ms. There remains some question of the weight of this problem, as no jitter requirements were published in the EIPP recommendations. Some delay jitter may be acceptable, but that is likely application specific. As mentioned previously we feel that a more optimized implementation of GridStat will help lower delay jitter.

The development of our GridStat PDC emulator shows how support for periodic updates and redundant paths in the infrastructure leads to simple application code. There is no code in the publishers related to redundant path routing. The subscriber has code only to request redundant paths. Otherwise the application is unchanged from the single-path case. Exploration of the utility of operational modes for PMU applications also remains for future work.

The research conducted for this thesis did not involve work directly on GridStat, but rather on applications utilizing GridStat. In this fashion we were able to evaluate the ease with which an application developer interacts with GridStat. While GridStat itself is robust and easy to interface with, our GridStat Java Application Framework simplifies task of GridStat application development even further.

6.1 Future Work

The complexity of the communication relationships in the power grid's control system is an ongoing challenge for the industry. GridStat's goals are complementary to those of efforts such as IEC standards 61850 and 61970 that provide frameworks for defining object and service models for field equipment and information models for control centers. There is a great deal of research to be done to see how publish-subscribe communications meshes with these models.

The following sections mention specific areas for future GridStat research.

6.1.1 Trust Management

Another challenging aspect of the modern power grid is the complexity of the relationships between the businesses that operate its components. Economic competitors must nevertheless cooperate to assure reliable grid operation. There is legitimate fear that information exchanged for the latter purpose could be used or manipulated to provide competitive advantage. PMU data, which reveals the power transfer, is therefore potentially sensitive. The infrastructure must deliver data on time, but also with integrity and only to legitimate subscribers. Research in trust management will inform the incorporation of these guarantees into the QoS brokers and status routers, [5].

6.1.2 Group Subscription

It would be interesting to examine the tradeoffs of moving GridStat's group subscription interface deeper into the GridStat system. Currently it exists as a module in GridStat subscribers, and all other GridStat components are kept unaware of status items' participation in groups. In particular, we feel there would be value in examining the issues related to making status routers statefully aware of group subscriptions.

The potential benefits of a group subscription aware status router are not immediately apparent, but two ideas merit investigation.

Packet Combining

GridStat status routers are designed to combine incoming packets destined for the same outgoing link when the situation presents itself as mentioned in section 2.3.2. This packet combining is intended to increase the performance of a GridStat network by

reducing the number of packets in the network with the intention of increasing the overall network throughput. Currently packet grouping occurs when the system is under load and packets arrive faster than they may be serviced individually.

If a status router is aware that some number of inbound status items are part of a group, it can wait for the arrival and combination of all of the status items in a group before forwarding them. It seems possible that such a scheme could result in higher overall network throughput than non-group aware packet combining strategies.

Rate Filtering Restructure

To have successful rate filtering of a group subscription, all of the status items from a group that are passed through the filter must have the same timestamp. In order to meet the timing requirements of a group subscription without adding group subscription state to status routers, the current GridStat implementation ensures that all rate filtering is done deterministically. This means that even status items that are not part of a group are filtered in this way. This has the consequence that all filtered traffic becomes clumped around particular time offsets, which artificially increases the network contention packets experience. If status routers were aware of group subscriptions, this problem could be avoided by having a group subscription specific time offset.

Potential Problems

While the two previously mentioned advantages arise from making status routers group subscription aware, this ability does not come without some tradeoffs. Status routers are designed to be simple and fast. Adding the additional state involved in making

status routers aware of group subscriptions will complicate status routers. Part of any investigation will have to be an examination of the impact of this increased complexity on status router performance and size.

6.1.3 Command Message Dissemination

If GridStat is going to be used as the primary network communications infrastructure for the power grid, then it is essential that work be done to support control message distribution in GridStat. Issuing control messages fits a very different paradigm than the publish-subscribe model supported by GridStat. Unlike status variables that are periodic, control messages must be assumed to occur with no predictable frequency, and should be able to be delivered on demand. In addition, an application issuing a control message will expect a confirmation of the successful delivery, and perhaps execution of the control message. GridStat needs to be extended into a hybrid system that primary supports publish-subscribe, but can additionally support point-to-point guaranteed on-demand message delivery.

APPENDIX

APPENDIX A

PMU EMULATOR APPLICATION SOURCE CODE

A.1 PMUPublisherApplication.java

```
package edu.wsu.gridstatapplication.publisher.PMU;

import edu.wsu.gridstatapplication.common.*;
import edu.wsu.gridstatapplication.publisher.*;

/**
 * <p>Title: Gridstat PMU Publisher</p>
 *
 * <p>Description: Inform base class of appropriate
 * wrapper and panel classes.</p>
 *
 * <p>Copyright: Copyright (c) 2004</p>
 *
 * <p>Company: Washington State University</p>
 *
 * @author Ryan Johnston
 * @version 1.0
 */
public class PMUPublisherApplication extends PublisherApplication
{
  public PMUPublisherApplication(String[] args) throws BadTypeException
  {
    super(args, PMUPublisherWrapper.class, PMUPublisherPanel.class);
  }

  public static void main(String[] args) throws BadTypeException
  {
    PMUPublisherApplication p = new PMUPublisherApplication(args);
  }
}
```

A.2 PMUPublisherWrapper.java

```
package edu.wsu.gridstatapplication.publisher.PMU;

// GridStat Dependencies
import edu.wsu.gridstat.common.Constants;
import edu.wsu.gridstat.publisher.common.*;
import edu.wsu.gridstatapplication.common.*;
import edu.wsu.gridstatapplication.common.adapter.*;
import edu.wsu.gridstatapplication.publisher.*;
import edu.wsu.gridstatapplication.phasor.*;

// Java Dependencies
import java.util.logging.Logger;
import java.util.concurrent.Semaphore;

/**
 * <p>Title: GridStat PMU Publisher Wrapper</p>
 *
```

```
 * <p>Description: Wraps a GridStat publisher in an easy to use
        interface</p>
 *
 * <p>Copyright: Copyright (c) 2005</p>
 *
 * <p>Company: Washington State University</p>
 *
 * @author Ryan Johnston
 * @version 1.0
 */

public class PMUPublisherWrapper extends PublisherWrapper
{
  private Thread runner;
  private boolean paused;
  private boolean publish;
  protected Semaphore semaphore;
  protected Logger logger;

  //Attributes from publisherInfo
  protected int variableId;
  protected String variableName;
  protected String publisherName;
  protected String edgeSeverName;
  protected String publicationType;
  protected int pubRate;
  protected int highVal;
  protected int lowVal;
  protected VariableAdapter value;

  public PMUPublisherWrapper(Publisher pub) throws BadTypeException
  {
    super(pub);
    this.logger = Logger.getLogger("edu.wsu.gridstat.publisher");
    this.semaphore = new Semaphore(1, true);
    this.variableName = PropertySheet.getString("StatusName");
    this.publisherName = PropertySheet.getString("PublisherName");
    this.edgeSeverName = PropertySheet.getString("EdgeServerName");
    this.pubRate = PropertySheet.getInteger("Rate");
    this.highVal = 100;
    this.lowVal = -100;
    this.value = PropertySheet.getProperty("Value");
    this.publicationType = PropertySheet.getString("PublicationType");
  }

  /**
   * The <code>disconnect</code> method is used to disconnect from the
        closest
   * ControlServer.
   * <BR>
   * @return Returns the gridstat publisher's response
   */
  public short disconnect()
  {
    short errorCode;

    if((errorCode = this.publisher.disConnectFromEdge()) <
        Constants.COMMAND_OK)
    {
      return errorCode;
    }
```

```
  this.publisher.cleanUp();
  return Constants.COMMAND_OK;
}

/**
 * The <code>connect</code> method is used to connect to the closest
 * ControlServer.
 * <BR>
 * @return Returns the gridstat publisher's response
 */
public short connect()
{
  short errorCode;

  if(this.publisher.isConnected())
  {
    this.logger.fine("Already connected to server.");
    return Constants.COMMAND_OK;
  }
  if((errorCode = this.publisher.connectToEdge()) <
        Constants.COMMAND_OK)
  {
    this.logger.severe(Constants.getErrorMsg(errorCode));
    return errorCode;
  }

  this.logger.fine("Connected to server.");
  return Constants.COMMAND_OK;
}

/**
 * The <code>unpublish</code> method is used to stop publishing
        messages, and
 * unpublish from the ControlServer.
 * <BR>
 * @return Returns the gridstat publisher's response
 */
public short unpublish()
{

  short errorCode;

  // Unregister the publishing from the ControlServer
  if((errorCode = this.publisher.unregisterPublish(this.variableId)) <
        Constants.COMMAND_OK)
  {
    return errorCode;
  }

  // Stop the thread that are publishing messages
  if(this.runner != null)
  {
    this.runner.interrupt();
  }
  this.publish = false;

  return Constants.COMMAND_OK;
}

/**
 * The <code>publish</code> method is used to start a thread that
        publishes
```

```
 * messages at the specified rate.
 * <BR>
 * @return Returns the gridstat publisher's response
 */
public short publish()
{
  short errorCode;
  // See if we are connected to the ControlServer
  if(this.publisher.isConnected() == false)
  {
    // Make sure that we can connect
    if((errorCode = this.connect()) < Constants.COMMAND_OK)
    {
      return errorCode;
    }
  }

  this.logger.fine("Publishing:\n" + this.toString());

  // Register the publishing to the ControlServer
  try
  {
    this.variableId = PMUPacket.registerPMUPhasor(this.publisher,
         this.variableName, Constants.PRIORITY_MEDIUM, this.pubRate,
         this.pubRate, 6, 2);
  }
  catch(PublisherException ex)
  {
    return Constants.ERROR_PUB_ALREADY_REGISTERED;
  }

  this.runner = new Thread()
  {
    public void run()
    {
      logger.fine("Starting Thread: " + publisherName + "@" +
          edgeSeverName + ":" + variableName + ", interrupted");
      PMUPacket packet = new PMUPacket(6, 2);
      try
      {
        while(!runner.interrupted())
        {
          // If it is paused we will stop here
          semaphore.acquire();
          semaphore.release();

          // The following decision tree determines which automated value
           change to make
          if(publicationType.equals("linear"))
          {
           packet.linearStep();
          }
          else if(publicationType.equals("weighted"))
          {
           packet.weightedRandomize();
          }
          else
          {
           packet.randomize();
          }
          publisher.publish(variableId, packet.getBytes());
```

```
        // Update the UI
        if(PropertySheet.containsProperty("gui"))
        {
         value.setValue(packet.toString());
        }

        // Sleep for a while
        Thread.sleep(pubRate);
       }
     }
     catch(InterruptedException ie)
     {
      logger.fine("Thread: " + publisherName + "@" + edgeSeverName +
        ":" + variableName + ", interrupted");
     }
     catch(Exception e)
     {
      logger.fine("Exception in Random publisher run: " + e);
     }

     logger.fine("Thread done: " + publisherName + "@" + edgeSeverName
        + ":" + variableName + ", interrupted");
   }
  };

  // Run the publishing thread
  this.runner.start();
  this.publish = true;
  return Constants.COMMAND_OK;
}

public void togglePause()
{
  if(this.paused)
  {
   this.paused = false;
   this.semaphore.release();
  }
  else
  {
   this.paused = true;
   try
   {
    this.semaphore.acquire();
   }
   catch(InterruptedException ie)
   {
    this.logger.severe("PublisherGUI, pausePublish, Exception: " +
        ie);
    return;
   }

  }
}

public String toString()
{
  String ret = new String();
  ret += "Publisher Name:\t" + this.publisherName + "\n";
  ret += "Variable Name :\t" + this.variableName + "\n";
  ret += "Edge SR Name  :\t" + this.edgeSeverName + "\n";
  ret += "Data Rate     :\t" + this.pubRate + " ms/msg\n";
```

```
    ret += "Generator Type:\t" + this.publicationType + "\n";
    ret += "Bounds        :\t(" + this.lowVal + ", " + this.highVal +
           ")";
    return ret;
  }

  public Object getColumn(int column)
  {
    switch(column)
    {
      case 0:
        return this.edgeSeverName;
      case 1:
        return this.publisherName;
      case 2:
        return this.variableName;
      case 3:
        return new Boolean(this.publish);
    }
    return null;
  }

  public boolean getIsPublishing()
  {
    return this.publish;
  }

  public String getEdgeManager()
  {
    return this.edgeSeverName;
  }

  public String getPublisherName()
  {
    return this.publisherName;
  }

  public String getStatusName()
  {
    return this.variableName;
  }
}
```

A.3 PMUPublisherPanel.java

```
package edu.wsu.gridstatapplication.publisher.PMU;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

import edu.wsu.gridstatapplication.common.*;
import edu.wsu.gridstatapplication.common.adapter.*;
import edu.wsu.gridstatapplication.common.ui.*;

/**
 * <p>Title: Gridstat PMU Publisher</p>
 *
 * <p>Description: </p>
```

```
 *
 * <p>Copyright: Copyright (c) 2004</p>
 *
 * <p>Company: Washington State University</p>
 *
 * @author Ryan Johnston
 * @version 1.0
 */
public class PMUPublisherPanel extends BasePanel
{
  public PMUPublisherPanel(BaseApplication ba)
  {
    super(ba);
    try
    {
      jbInit();
      addActionListeners();
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }

    PropertySheet.setProperty(new TextFieldAdapter("EdgeServerName",
          this.edgeServerNameField));
    PropertySheet.setProperty(new TextFieldAdapter("StatusName",
          this.statusNameField));
    PropertySheet.setProperty(new TextFieldAdapter("PublisherName",
          this.publisherNameField));
    PropertySheet.setProperty(new IntegerFieldAdapter("Rate",
          this.rateField));
    PropertySheet.setProperty(new TextFieldAdapter("Value",
          this.valueField));
    PropertySheet.setProperty(new IntegerFieldAdapter("Length", null));

    LinkedList<JRadioButton> datatypelist = new
          LinkedList<JRadioButton>();
    PropertySheet.setProperty(new RadioAdapter("DataType",
          datatypelist));

    LinkedList<JRadioButton> pubvalueslist = new
          LinkedList<JRadioButton>();
    pubvalueslist.add(this.linearRadio);
    pubvalueslist.add(this.randomRadio);
    pubvalueslist.add(this.weightedRadio);
    PropertySheet.setProperty(new RadioAdapter("PublicationType",
          pubvalueslist));

    PropertySheet.setProperty(new BooleanFieldAdapter("Alert", null));
  }

  private void addActionListeners()
  {
    connectButton.addActionListener(this.baseApplication);
    //connectButton.addActionListener(this);
    exitButton.addActionListener(this.baseApplication);
    //exitButton.addActionListener(this);
    pauseButton.addActionListener(this.baseApplication);
    //pauseButton.addActionListener(this);
    publishButton.addActionListener(this.baseApplication);
    linearRadio.addActionListener(this);
    randomRadio.addActionListener(this);
```

```
  weightedRadio.addActionListener(this);

}

private void jbInit() throws Exception
{
  this.setLayout(borderLayout1);
  esnLabel.setText("Edge Server Name:");
  edgeServerNameField.setText("");
  connectButton.setText("Connect");
  bottomPanel.setLayout(gridLayout2);
  publishButton.setText("Publish");
  pauseButton.setEnabled(false);
  pauseButton.setText("Pause");
  exitButton.setText("Exit");
  statusNameLabel.setText("Variable Name:");
  statusNameField.setText("");
  pubValuesLabel.setText("Pub Values");
  pubNameLabel.setText("Publisher Name:");
  publisherNameField.setText("");
  linearRadio.setText("linear");
  rateLabel.setText("Rate:");
  rateField.setText("");
  randomRadio.setText("random");
  weightedRadio.setText("weighted");
  valueLabel.setText("Value:");
  valueField.setText("");
  valueField.setEditable(false);
  borderLayout1.setHgap(2);
  borderLayout1.setVgap(2);
  gridLayout2.setColumns(5);
  gridLayout2.setHgap(2);
  gridLayout2.setVgap(2);
  topPanel.setLayout(gridBagLayout1);
  this.add(bottomPanel, java.awt.BorderLayout.SOUTH);
  bottomPanel.add(connectButton);
  bottomPanel.add(publishButton);
  bottomPanel.add(pauseButton);
  bottomPanel.add(exitButton);
  this.add(topPanel, java.awt.BorderLayout.CENTER);
  topPanel.add(pubValuesLabel, new GridBagConstraints(2, 1, 1, 1, 0.0,
         0.0, GridBagConstraints.WEST, GridBagConstraints.NONE, new
         Insets(0, 0, 0, 0), 0, 0));
  topPanel.add(rateLabel, new GridBagConstraints(0, 3, 1, 1, 0.0, 0.0,
         GridBagConstraints.WEST, GridBagConstraints.NONE, new
         Insets(0, 0, 0, 0), 0, 0));
  topPanel.add(valueLabel, new GridBagConstraints(0, 4, 1, 1, 0.0, 0.0,
         GridBagConstraints.WEST, GridBagConstraints.NONE, new
         Insets(0, 0, 0, 0), 0, 0));
  topPanel.add(esnLabel, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
         GridBagConstraints.WEST, GridBagConstraints.NONE, new
         Insets(0, 0, 0, 0), 0, 0));
  topPanel.add(pubNameLabel, new GridBagConstraints(0, 2, 1, 1, 0.0,
         0.0, GridBagConstraints.WEST, GridBagConstraints.NONE, new
         Insets(0, 0, 0, 0), 0, 0));
  topPanel.add(statusNameLabel, new GridBagConstraints(0, 1, 1, 1, 0.0,
         0.0, GridBagConstraints.WEST, GridBagConstraints.NONE, new
         Insets(0, 0, 0, 0), 0, 0));
  topPanel.add(statusNameField, new GridBagConstraints(1, 1, 1, 1, 1.0,
         0.0, GridBagConstraints.WEST, GridBagConstraints.HORIZONTAL,
         new Insets(0, 0, 0, 0), 0, 6));
```

```
  topPanel.add(publisherNameField, new GridBagConstraints(1, 2, 1, 1,
         1.0, 0.0, GridBagConstraints.WEST,
         GridBagConstraints.HORIZONTAL, new Insets(0, 0, 0, 0), 0, 6));
  topPanel.add(rateField, new GridBagConstraints(1, 3, 1, 1, 1.0, 0.0,
         GridBagConstraints.WEST, GridBagConstraints.HORIZONTAL, new
         Insets(0, 0, 0, 0), 0, 6));
  topPanel.add(edgeServerNameField, new GridBagConstraints(1, 0, 2, 1,
         1.0, 0.0, GridBagConstraints.WEST,
         GridBagConstraints.HORIZONTAL, new Insets(0, 0, 0, 0), 0, 6));
  topPanel.add(valueField, new GridBagConstraints(1, 4, 1, 1, 1.0, 0.0,
         GridBagConstraints.WEST, GridBagConstraints.HORIZONTAL, new
         Insets(0, 0, 0, 0), 0, 6));
  topPanel.add(linearRadio, new GridBagConstraints(2, 2, 1, 1, 0.0,
         0.0, GridBagConstraints.WEST, GridBagConstraints.NONE, new
         Insets(0, 0, 0, 0), 0, 0));
  topPanel.add(weightedRadio, new GridBagConstraints(2, 4, 1, 1, 0.0,
         0.0, GridBagConstraints.WEST, GridBagConstraints.NONE, new
         Insets(0, 0, 0, 0), 0, 0));
  topPanel.add(randomRadio, new GridBagConstraints(2, 3, 1, 1, 0.0,
         0.0, GridBagConstraints.WEST, GridBagConstraints.NONE, new
         Insets(0, 0, 0, 0), 0, 0));
}

/**
 * cleanup This method is intended to contain any clean up code that
         the
 * application requires.
 */
public void cleanUp()
{
}

/**
 * Invoked when an action occurs.
 *
 * @param e ActionEvent
 */
public void actionPerformed(ActionEvent e)
{
  if(e.getActionCommand() == "Connect")
  {
    this.connectButton.setText("Disconnect");
    this.edgeServerNameField.setEnabled(false);
    this.rateField.setEnabled(false);
    this.publisherNameField.setEnabled(false);
    this.statusNameField.setEnabled(false);
  }
  else if(e.getActionCommand() == "Disconnect")
  {
    this.connectButton.setText("Connect");
    this.publishButton.setText("Publish");
    this.pauseButton.setText("Pause");
    this.pauseButton.setEnabled(false);
    this.edgeServerNameField.setEnabled(true);
    this.rateField.setEnabled(true);
    this.publisherNameField.setEnabled(true);
    this.statusNameField.setEnabled(true);

  }
  else if(e.getActionCommand() == "Pause")
  {
    this.pauseButton.setText("Unpause");
```

```
    }
    else if(e.getActionCommand() == "Unpause")
    {
      this.pauseButton.setText("Pause");
    }
    else if(e.getActionCommand() == "Publish")
    {
      this.connectButton.setText("Disconnect");
      this.publishButton.setText("Unpublish");
      this.edgeServerNameField.setEnabled(false);
      this.rateField.setEnabled(false);
      this.publisherNameField.setEnabled(false);
      this.statusNameField.setEnabled(false);
      this.pauseButton.setEnabled(true);
    }
    else if(e.getActionCommand() == "Unpublish")
    {
      this.publishButton.setText("Publish");
      this.pauseButton.setText("Pause");
      this.pauseButton.setEnabled(false);
    }
    else if(e.getActionCommand() == "linear" || e.getActionCommand() ==
           "random" || e.getActionCommand() == "weighted")
    {

            (PropertySheet.getProperty("PublicationType")).setValue(e.get
           ActionCommand());
    }
  }

  protected BorderLayout borderLayout1 = new BorderLayout();
  protected JLabel esnLabel = new JLabel();
  protected JTextField edgeServerNameField = new JTextField();
  protected JButton connectButton = new JButton();
  protected JPanel bottomPanel = new JPanel();
  protected JButton publishButton = new JButton();
  protected JButton pauseButton = new JButton();
  protected JButton exitButton = new JButton();
  protected JLabel statusNameLabel = new JLabel();
  protected JTextField statusNameField = new JTextField();
  protected JLabel pubValuesLabel = new JLabel();
  protected JLabel pubNameLabel = new JLabel();
  protected JTextField publisherNameField = new JTextField();
  protected JRadioButton linearRadio = new JRadioButton();
  protected JLabel rateLabel = new JLabel();
  protected JTextField rateField = new JTextField();
  protected JRadioButton randomRadio = new JRadioButton();
  protected JRadioButton weightedRadio = new JRadioButton();
  protected JLabel valueLabel = new JLabel();
  protected JTextField valueField = new JTextField();
  protected GridLayout gridLayout2 = new GridLayout();
  protected JPanel topPanel = new JPanel();
  protected BorderLayout borderLayout2 = new BorderLayout();
  protected GridBagLayout gridBagLayout1 = new GridBagLayout();
}
```

BIBLIOGRAPHY

[1] Bakken, D. “Middleware”, Chapter in *Encyclopedia of Distributed Computing*, Urban, J. and Dasgupta, P., eds., Kluwer Academic Publishers, to appear. Available at http://www.eecs.wsu.edu/~bakken/middleware.pdf.

[2] Carroll, J. “Eastern Interconnect Phasor Project: TVA’s Super Phasor Data Concentrator Architecture” presentation at the Eastern Interconnect Phasor Project Meeting, April, 2005, Chattanooga, TN. Retrieved June 2005:

[3] Chandy, K. M. and Lamport, L. “Distributed snapshots: determining global states of distributed systems”, *ACM Transactions on Computer Systems*, Volume 3, Issue 1, pg63-75, Febuary 1985.

[4] Comer, D. Network Systems Design Using Network Processors, Prentice Hall, 2003.

[5] Drugan, O., Dionysiou, I., Bakken, D., Plagemann, T., Hauser, C., and Frincke, D. “On the Importance of Composability of Ad Hoc Mobile Middleware and Trust Management”, *Proceedings of the 2nd International Service Availability Symposium*, Berlin, Germany, 2005. Lecture Notes in Computer Science, Springer, 2005, to appear.

[6] Gjermundrød, H., Bakken, D., Hauser, C. “GridStat Architecture and Mechanisms”, in preparation.

[7] Hart, D., Uy, D., Phadke, A., Forsman, S., and Kunsman, S. “PMUs: Overview and Application to Electric Power Networks”, *Proceedings of the 4th Multiconference on Systemics, Cybernetics, and Informatics* (SCI 2000/ISAS 2000), volume IX, Orlando, USA.

[8] Hauer, J. and Taylor, C. “Information, Reliability and Control in the New Power System”, American Control Conference, Philadelphia, PA, June, 1998.

[9] Hauser, C., Bakken, D. and Bose, A. "A Failure to Communicate: Next-Generation Communication Requirements, Technologies, and Architecture for the Electric Power Grid", *IEEE Power and Energy*, 3(2), March/April, 2005, 47-55. Retrieved June 2005: http://gridstat.net/GridStat-Power-Energy-March05.pdf.

[10] IEEE. "IEEE Standard for Synchrophasors for Power Systems", IEEE Std 1344-1995(2001), New York, 2001.

[11] Kamwa, I., Grondin, R., and Hebért, Y. “Wide-Area Measurement Based Stabilizing Control of Large Power Systems—A Decentralized/Hierarchical Approach”, *IEEE Transactions on Power Systems*, 16(1), February, 2001.

[12] Lamport, L. “Time, clocks, and the ordering of events in a distributed system”, *Communications of the ACM*, Volume 21, Issue 7, pg558-565, July 1978.

[13] Larsson, M., Gardner, R., and Rehtanz, C. “Interactive Simulation and Visualization of Wide-area Monitoring and Control Applications”, submitted to Power Systems Computation Conference, Liège, Belgium, 2005.

[14] Larsson, M., Rehtanz, C., and Bertsch, J. “Real-time Voltage Stability Assessment for Transmission Corridors”, Proceedings of IFAC Power Plants and Power Systems Control Conference, Seoul, Korea, 2003.

[15] Liu, Y. “GPS/Internet Based Frequency Monitoring Network” presentation at the Eastern Interconnect Phasor Project Meeting, April, 2005, Chattanooga, TN. Retrieved June 2005: http://phasors.pnl.gov/Meetings/2005 April/presentations/Post this oneEIPP_TVA_Liu_05.pdf

[16] Martin., K. *Phasor Measurements at the Bonneville Power Administration.* Power Systems and Communications Infrastructures for the Future, Beijing, China, September 2002.

[17] Martinez, C., Parashar, M., Dyer, J. and Coroas, J. (eds.), Phasor Data Requirements for Real Time Wide-Area Monitoring, Control and Protection Applications (Final Draft), Consortium for Electric Reliability Technology Solutions (CERTS), January, 2005. Retrieved June 2005: http://phasors.pnl.gov/resources_realtime/EIPP_RealTimeGroup_DataRequirements_Draft5 - Jan 26 05.pdf

[18] Mittelstadt, W., Krause, P., Overholt, P., Sobajic, D., Hauer, J., Wilson, R., and Rizy, D. "The DOE Wide Area Measurement System (WAMS) Project—Demonstration of Dynamic Information Technology for the Future Power System", EPRI Conference on the Future of Power Delivery, Washington, D.C., April 1996. Retrieved June 2005: http://www.osti.gov/bridge/purl.cover.jsp?purl=/254951-e5g22f/webviewable/

[19] Object Management Group. Data distribution service for real-time systems specification. Technical Report Version 1.0, Object Management Group, December 2004.

[20] Pardo-Castellote, G. "OMG Data-distribution Service: Architectural Overview". *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops* (ICDCSW03), Providence, RI, May 2003.

[21] Quintero, J. and Venkatasubramanian, V. "A Real-Time Wide-Area Control Framework for Mitigating Small-Signal Instability in Large Electric Power

Systems", *Proceedings of the 38th Annual Hawaii International Conference Software and Systems*, Hawaii, 2005.

[22] Tanenbaum, A., van Steen, M. *Distributed Systems Principles and Paradigms*. Prentice Hall, Upper Saddle River, New Jersey, 07458, 2002.

[23] Taylor, C., Erickson, D., Martin, K., Wilson, R. and Venkatasubramanian, V. "WACS—Wide-Area Stability and Voltage Control System: R&D and Online Demonstration", *Proceedings of the IEEE* (Special Issue on Energy Infrastructure Systems), 93(5), May, 2005.

[24] Tomsovic, K., Bakken, D., Venkatasubramanian, V. and Bose, A. "Designing the Next Generation of Real-Time Control, Communication and Computations for Large Power Systems", *Proceedings of the IEEE* (Special Issue on Energy Infrastructure Systems), 93(5), May, 2005. Retrieved June 2005: http://gridstat.net/Power-GridStat-ProceedingsIEEE.pdf

[25] van't Hag, J. "Data-Centric to the Max — The SPLICE Architecture Experience", *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops* (ICDCSW03), Providence, RI, May 2003.

[26] Zinky, J., O'Brien, L., Bakken, D., Krishnaswamy, V., and Ahamad, M.. "PASS: A service for efficient large scale dissemination of time varying data using CORBA", International Conference on Distributed Computing Systems, Austin, TX, June 1999.