Speeding Up Processing Data From Millions of Smart Meters

Jiang Zheng, Zhao Li, Aldo Dagnino
ABB Inc., US Corporate Research
940 Main Campus Drive, Raleigh, NC, USA
{jiang.zheng, zhao.li, aldo.dagnino}@us.abb.com

ABSTRACT
As an important element of the Smart Grid, Advanced Metering Infrastructure (AMI) systems have been implemented and deployed throughout the world in the past several years. An AMI system connects millions of end devices (e.g., smart meters and sensors in the residential level) with utility control centers via an efficient two-way communication infrastructure. AMI systems are able to exchange substantial meter data and control information between utilities and end devices in real-time or near real-time. The major challenge our research was to scale ABB's Meter Data Management System (MDMS) to manage data that originates from millions of smart meters. We designed a lightweight architecture capable of collect ever-increasing large amount of meter data from various metering systems, clean, analyze, and aggregate the meter data to support various smart grid applications. To meet critical high performance requirements, various concurrency processing techniques were implemented and integrated in our prototype. Our experiments showed that on average the implemented data file parser took about 42 minutes to complete parsing, cleaning, and aggregating 5.184 billion meter reads on a single machine with the hardware configuration of 12-core CPU, 32G RAM, and SSD Hard Drives. The throughput is about 7.38 billion meter reads (206.7GB data) per hour (i.e., 1811TB/year). In addition, well-designed publish/subscribe and communication infrastructures ensure the scalability and flexibility of the system.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel programming;* F.1.2 [**Computation by Abstract Devices**]: Modes of Computation – *Parallelism and concurrency;* H.3.4 [**Information Storage and Retrieval**]: System and Software – *Performance evaluation (efficiency and effectiveness).*

General Terms
Algorithms, Measurement, Performance, Design, Experimentation, Verification.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE'14, March 22–26, 2014, Dublin, Ireland.
Copyright 2014 ACM 978-1-4503-2733-6/14/03…$15.00.
http://dx.doi.org/10.1145/2568088.2576798

Keywords
Architecture, Concurrency, Parallelism.

1. INTRODUCTION
A "Smart Grid" generally refers to a modern electric utility grid that uses computer-based information and communication technology to collect and manage information in an automated fashion [22]. Smart Grid technologies are able to significantly improve the quality of the production and distribution of electricity, especially in efficiency and economics. Stimulated by the concept of Smart Grid, Advanced Metering Infrastructure (AMI) systems have been widely deployed in the world in the past several years. An AMI system usually consists of smart meters, two-way communications networks, and data management systems. Unlike traditional home energy monitors, smart electrical meters not only record consumption of electric energy in intervals of 15 minutes or even every minute, but also communicate information back to the utility for monitoring and billing purposes. Additionally, unlike legacy utility communication systems, such as Supervisory Control and Data Acquisition (SCADA), an AMI system offers an efficient two-way communication infrastructure, connecting millions of end devices (e.g., smart meters and sensors in the residential level) with utility control centers, and exchanging substantial meter data and control information between them in real-time or near real-time. AMI systems break through the traditional fences of substations and transformers, pushing forward the boundaries of grid visibility to the residential territory [20].

A Meter Data Management System (MDMS) is responsible for collecting meter data that originates from large amount of smart meters, storing and transforming meter data into information that may be used by utility applications, such as Demand Response (DR) billing, Customer Information Systems (CIS), Outage Management Systems (OMS), Distribution State Estimation (DSE), and Voltage Var Optimization (VVO). These utility applications can now access and analyze interval meter measurements via AMI to provide more sophisticated functionality. For example, DR billing systems can offer better analytic results and a more flexible pricing infrastructure based on Time-of-Use (TOU). As a key component for managing large amount of meter data and unleashing the potentials of AMI, an MDMS not only simplifies IT integration of AMI, but also facilitates the distribution of the meter data across the utility enterprise by framing the volumes of interval data retrieved from the field into manageable and understandable information packets.

In the past few years, enhancements and benefits brought by AMI and MDMS to the utility customer information system have been widely witnessed and accepted and in return have attracted more efforts to build larger scale applications on top of AMI. Based on estimations from IDC Energy Insights, the North American market in MDMS will grow up to $869.1 million (USD) in 2013, in which a smaller utility will spend up to $250,000 on MDMS solutions while large utilities will potentially sign contracts ranging from $2 million to $4 million [8].

As one of the primary vendors with a good reputation in the power grid management market, ABB has produced power grid management products that are widely accepted by major utilities in the US [21]. However, ABB's existing MDMS products were not designed to manage meter data generated by millions of smart meters. The desired MDMS is required to collect ever-increasing large amount of meter data from various metering systems (e.g., AMI and advanced meter read (AMR)), clean, analyze and aggregate the meter data to support its customer information system in the short term, and a plethora of smart grid systems and applications in the long run. The input to the desired MDMS is the interval meter data measurements collected from 3~5 million residential smart meters every 15 minutes or even every minute in the future. The collected measurements are not only energy consumption but also engineering measurements (e.g., current and voltage) and/or power quality events (e.g., outage information).

We designed a lightweight architecture to fulfill the non-functional requirements, especially in the aspects of performance and scalability, of such a system. In order to meet critical high performance requirements, various concurrency processing techniques, such as Task Parallel Library (TPL) [18] and in-memory lock-free data structures, were implemented and/or integrated in our prototype (henceforth called the *System*). Our experiments demonstrated that on average the data parser for data files was able to complete parsing, cleaning, and aggregating 5.184 billion meter reads in 42 minutes 8 seconds on a single machine with the hardware configuration of 12-core CPU, 32G RAM, and SSD Hard Drives. The throughput is about 7.38 billion meter reads (206.7GB data) per hour (i.e., 1811TB/year).

In addition, well-designed publish/subscribe that provides a flexible interface to easily integrate with other systems and applications, as well as the communication infrastructure ensure the high scalability and flexibility of the system.

The rest of this **industrial paper** is organized as follows. Section 2 introduces the background and related work. Section 3 presents research challenges and constraints, as well as critical architectural requirements. Section 4 describes the design and implementation in detail. Section 5 shows the performance evaluation results. Section 6 provides further discussion. Section 7 presents conclusions and future work of this study.

2. BACKGROUND AND RELATED WORK

In this section, background information and related work of Meter Data Management Systems (MDMS), and Data Extract, Transform, and Load (ETL) are described in Section 2.1 and Section 2.2, respectively.

2.1 Meter Data Management Systems

As a centralized storage facility, a MDMS collects meter data from various metering systems, processes them, and provides the processed meter information to various utility applications (e.g., outage management, workforce management, and customer billing system). In addition, beyond collecting data from metering systems, most MDMS can also send signals back to smart meters and control them.

At the time of this study, based on the Pike Pulse Report in 2011 [14], Oracle, eMeter, and Itron were ranking the top three of MDMS vendors in the North American Market. Oracle attained the highest overall score due to its broad MDMS product line, massive scale, geographic presence, technical innovations, and integration of MDM with other Oracle products. At the time of this study, according to [23], the announced test results demonstrated that a system, consisting of Oracle Smart Meter Gateway, Oracle utilities meter data management system, and Oracle utilities customer care and billing system, can process more than 1 billion records and generate 500,000 customer bills within an eight hour nightly window. The tests were conducted against real business scenarios, in which the meter measurements are generated by 10 million smart meters in every 15 minutes [23]. In 2011, Siemens Energy announced the acquisition of eMeter and integrated its meter data management software into Siemens smart grid product line [6] [26]. Thereby, the new product, called EnergyIP, having both the meter data management the smart grid applications, appeared on the market. Siemens announced its MDMS centralized architecture that can manage up to 50 million smart meters [6]. Itron also has a strong metering system product line, which ranges from smart meters, AMI communication infrastructures, AMI high-end meter data collectors, and meter data management, but relatively weak in developing smart grid applications based on top of its MDMS [12][27]. As far as we had known when we were conducting this study, none of the above work dealt with meter read data in one minute interval. Also our experiments demonstrated that our design and prototype was able to meet the high performance requirements in processing big data files, meanwhile maintaining a relatively small footprint.

2.2 Data Extract, Transform, and Load

In the area of data processing, Extract, transform, and load (ETL) refers to a process of (1) extracts data from outside sources, (2) transforms the data to fit operational needs, and (3) loads the data into data storage such as database or data warehouse. ETL may be parallelized to obtain higher performance.

Agarwal et al. proposed an approach of parallel processing of ETL jobs involving XML documents. Their approach parallelizes ETL jobs by performing a shallow parsing of XML documents in parallel on one or more processors. [1] The method needs to generate intermediate XML documents, while our method will not have any intermediate XML documents. Also the producer and consumer are in same processor in their method, while the producer and multiple consumers will run in different threads/processors in our method.

Candea et al. provided a method of high-throughput ETL of program events for subsequent analysis. An event tap associated with a server was utilized to transform a server event into a tuple. They used the event tap to reduce the computational burden on the database and at the same time keep the server event data in the database relatively fresh. [3] [3]Our major challenge in ETL was to process files or file stream that contains bulk data.

Chen et al. proposed an ETL method for data cleaning in electric company based on genetic neural network to handle missing values. The method was able to improve the accuracy of missing data prediction by the global search ability of genetic algorithm and the nonlinear mapping ability of neural network. [4] We focused on the throughput of the ETL process for huge and ever increased meter data.

3. ARCHITECTURAL REQUIREMENTS

In this section, we will introduce general challenges and constraints, as well as critical architectural requirements, in this research.

3.1 General Challenges and Constraints

The foremost challenge was to effectively managing a large amount of data with minimal machine footprint. An MDMS that claimed to manage millions of meters usually has a large machine footprint. For example, Oracle demonstrated a MDMS system consisting of 32 servers to manage data from 10 million meters [23]. However, the high costs of building and maintaining a large cluster may prevent the MDMS from being widely deployed. Additionally, following long-term technical strategy of adopting the Microsoft technology stack for future products, Microsoft .NET and C# technologies were required to prototype the architecture design.

3.2 Critical Architectural Requirements

Although we needed to consider many quality attributes in the process of architecture design (e.g., scalability and flexibility), performance, especially throughput, was the most critical quality attribute type.

According to the functional requirements, the meter measurements are imported either from a bulk data file (*Data File Scenario*) or from an interval-based data stream (*AMI Data Stream Scenario*). Section 3.2.1 and 3.2.2 describes these two scenarios in detail, respectively.

3.2.1 Data File Scenario

The typical Data File Scenario is to import the previous day's meter reads from a bulk data file, and process the meter data load accumulated in 24 hours in only one hour.

Depending on the load style, the file scenario can be classified into the following two cases: the *Regular Case* and the *Extreme Case*. In the *Regular Case*, the regular intervals of meter loads are 15 minutes, i.e., 96 times per day. Each meter load contain three "channels" which means three data reads (i.e., records) with the values of energy consumption, current, and voltage, respectively. So there are 288 regular reads per day for each meter. In addition, for each day, three hours of Demand Response (DR) reads with only the energy consumption value are also collected. However, DR reads are collected in only one-minute intervals (i.e., 180 data reads per day). In total 468 data reads are collected per day for each meter. So in the *Regular Case*, 1.404 billion data reads are contained in a bulk data file for 3 million meters. The *Extreme Case* further extends the DR reads from three hours to 24 hours per day, i.e., 1440 DR data reads per day.

Based on the requirements, all the meter data shall be imported, cleaned, and aggregated in one hour. To clarify our analysis, we define the concept of throughput as the number of processed meter reads per hour. Based on this definition, the throughput in the Extreme Case is 5,184 billion meter reads per hour, which is far higher than that of the major competitors (e.g., the throughput of Siemens's system was 200 million meter reads per hour and the throughput of Oracle's MDMS was about 40 million meter reads per hour) at the time of this study.

The formal performance – throughput requirement for the Extreme Case of the File Scenario is as follows:

PERF_1 - The Extreme Case of the File Scenario

Requirement Statement:

The System shall load bulk data file in CSV format that contains up to 5.184 billion meter reads (i.e., data from up to 3,000,000 meters per day), store raw data, fill missing data, aggregate data, and export aggregated data to the CIS system in 1 hour. The maximum file size is 145.152 GB under Assumption (1). The maximum throughput is to process 5.184 billion meter reads per hour (i.e., 145.152 GB data per hour under Assumption (1)).

Calculation:

*- 96 fifteen-minute reads per day * 3 channels for 288 regular reads/meter, plus 24 hours * 60 minutes for 1,440 Demand Response reads/meter. (288 + 1440) * 3 million meters for 5.184 billion meter reads per day.*

*- 5.184 billion meter reads * 28 bytes / meter reads = 145.152 GB data.*

Assumption:

(1) Data size (in CSV format) for each meter read is 28 bytes.

(2) Multiple day's file can be processed separately by multiple servers in the System in one hour.

3.2.2 AMI Data Stream Scenarios

Unlike importing data from bulk data file, a typical AMI Data Stream Scenario is to import data stream from AMI, in which meter loads are evenly distributed in a day. Also, each meter read from AMI contains more bytes because AMI meter data is in XML format instead of CSV format.

The formal performance – throughput requirements for the Regular Case and the Extreme Case of the Stream Scenario are as follows:

PERF_2 – The Regular Case of Stream Scenario

Requirement Statement:

During the non-peak times of Demand Response (i.e., for regular reads), the System shall receive data from up to 5,000,000 meters from AMI in 15 minutes interval (i.e., up to 1.44 billion meter reads per day), store raw data, fill missing data, aggregate data, and export aggregated data to the ROMO system. The maximum throughput is to process 15 million meter reads per 15 minutes (i.e., 1.5 GB data per 15 minutes).

PERF_3 – The Extreme Case of Stream Scenario

Requirement Statement:

During the peak times of Demand Response, the System shall receive, store, and manage data that originates from up to 5,000,000 meters in 1 minute interval in addition to the regular reads described in the PERF_2 requirement, i.e., up to 8.64 billion meter reads per day. The maximum throughput is to process 20 million meter reads per 1 minute (i.e., 2 GB data per 1 minute).

Calculation:

*96 fifteen-minute reads per day * 3 channels for 288 reads/meter, plus 24 hours * 60 minutes for 1,440 Demand Response reads/meter. (288 + 1440) * 5 million meters for 8.64 billion meter reads per day. (5 million "regular" meter reads per 1 minute * 3 channels + 5 million Demand Response meter reads per 1 minute * 1 channel) for 20 million meter reads per 1 minute.*

Assumption:

(1) The demand response reads are in addition to the "regular" reads during the peak times.

(2) The data stream is in XML format in CIM model [6].

(3) Data size for each meter read is 100 bytes.

Table 1 shows a summary of the performance – throughput requirements for all the cases of both the Data File Scenario and the AMI Data Stream Scenario. Among these cases, the ***Extreme Case*** of the ***Data File Scenario*** has the largest processing throughput (5.184 billion reads per hour) and data throughput (145.152 GB data per hour).

4. DESIGN AND IMPLEMENTATION

In this section, we will highlight key techniques that contribute to the high performance and scalability of the *System*.

4.1 Overall Architecture

Figure 1 shows the component diagram for the *System*. At the high level, the components of the *System* can be classified into the following four parts: the meter data input layer in the left part of the figure, the meter data storage and the message coordinator in the middle part of the figure, and the application connectors in the right part of the figure.

Meter Data Input Layer

The meter data input layer consists of three modules: the protocol translation module, the stream parser module and the file parser module. The protocol translation module supports different AMI communication protocols and information models used by AMI, translating the input data into a standard information model (e.g., IEC61968-9) and sending it to the stream parser for processing.

The stream parser construes, cleans, and aggregates the meter data received from the protocol translation module. The file parser construes, cleans, and aggregates the meter data received from the bulk file in CSV format that contains one-day meter measurements. The parsers have two types of output: cleaned raw data and aggregated data. The former will be stored into the data repository, and the latter will be stored and also sent to Smart Grid applications (e.g., CIS). Section 4.2 describes the file parser in detail.

Meter Data Storage

The meter data storage is used to store both aggregated data and cleaned raw data. It consists of two parts: a relational database (RDBMS) and flat files. The former is used to store the aggregated data, and the latter are used to store the raw data. Section 4.3 describes the hybrid flat file / RDMBS storage mechanism.

Message Coordinator

The message coordinator is a publish/subscribe infrastructure, coordinating the behaviors of components in the *System*. As connected through the publish/subscribe infrastructure, the components of the *System* are loosely coupled: by sending a message to the Message Coordinator, one component can coordinate its behavior with other components that subscribed to the sent message. Traditional applications of publish/subscribe infrastructures extend across the entire enterprise, whereas the use of publish/subscribe here is limited to the confines of the *System*. There is no need to deploy a heavy-duty middleware function to fulfill this function.

Table 1. Summary of Throughput Requirements

	Data File Scenario		**AMI Data Stream Scenario**	
	Extreme Case	**Regular Case**	**Regular DR**	**Extreme DR**
Data Format	CSV	CSV	XML	XML
# of meters	3M	3M	5M	5M
# of channels	3 for regular reads, 1 for DR reads	3 for regular reads, 1 for DR reads	3	3 for regular reads, 1 for DR reads
# of regular reads / day for each meter	288	288	288	288
# of DR reads / day for each meter	1440	180	0	1440
# of reads / day	5.184 billion	1.404 billion	1.44 billion	8.64 billion
Max processing throughput	**5.184 billion reads / hour**	1.404 billion reads / hour	15 million reads / 15 mins	20 million reads / 1 min
Data size for each read / command	28 bytes	28 bytes	100 bytes	100 bytes
Max data throughput	**145.152 GB / hour**	39.312 GB / hour	1.5 GB / 15 mins (6 GB / hour)	2 GB / 1 min (120 GB / hour)

As components in the *System* are loosely coupled and connected through messages, adding a new component to the *System* becomes easy: register messages sent by the new components to the message coordinator, identify the subscribers who are interested in these messages and link the subscribers with the newly added component by subscribing to the messages sent by the newly added component. In addition, each component only focuses on one specific function (e.g., the file parser only process the large data file), which makes the whole system easy to maintain. Section 4.5 provides detailed information in how the Message Coordinator works.

Application Connectors

Application Connectors are responsible for sending meter information tailored to certain Smart Grid applications. A connector tailors information from a standard meter information model to diversified information models and conforms to the communication protocols used by different Smart Grid applications.

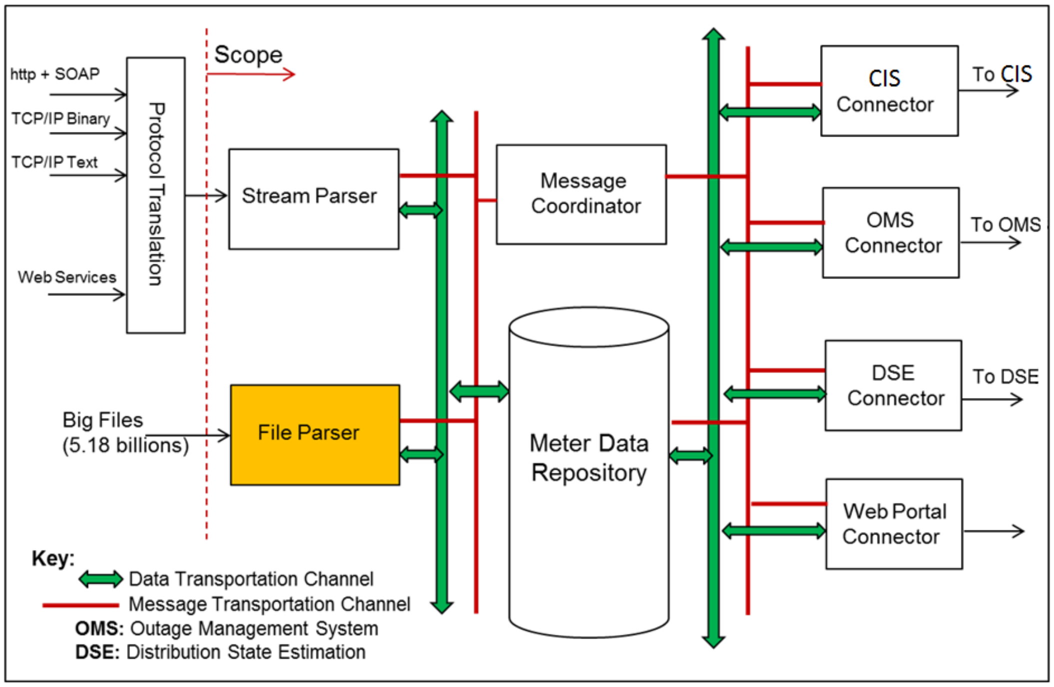

Figure 1. Overall Architecture

4.2 Meter Data Parser

The meter data parser works with large amount of meter data in the following steps.

Step 1: Large amount of mostly structured raw data is generated by various sensors, end devices, automatic data generation infrastructures, and/or other data sources. The raw data may be imported as files in different predefined formats, such as Comma-Separated Values (CSV), Extensible Markup Language (XML), or other user-defined formats. Partial of the dataset may be missing or duplicated, and there may be mistakes in the generated raw data.

Step 2: Before loading the raw data from the field, the data parser creates and initializes well-designed large-scale data structures in memory in order to stage and organize input data, and preserve aggregated datasets before storing them into databases or files. These data structures are thread-safe so that they can be accessed by multiple threads concurrently without any concurrency defects. The data parser also loads necessary metadata, such as configuration data for aggregation and summary of historical data, from databases or other data storage mechanisms. Different metadata may be loaded depending on different accumulative aggregation/analysis algorithms to be conducted in memory after loading the raw data.

Step 3: After the initialization described in Step 2, the data parser concurrently loads the raw data files or collects the raw data stream from the data sources in the stream fashion. Each thread works on one file. For each line or block of raw data, the data parser parses the raw data depending on the predefined data formats, eliminates useless text such as commas, spaces, and XML tags, and converts useful data from plain text to corresponding data types.

Step 4: Various algorithms of validation, estimation, and editing (VEE), such as foreign key validation, individual numeric data validation, duplication validation, and missing data estimation, are applied to the converted raw data in order to clean up the input data. During the scanning of the raw data, many statistical and aggregation algorithms can also be executed to accumulatively analyze and aggregate the cleaned staging data. In-memory configuration information and historical aggregation data that has been loaded in Step 2 can help to perform aggregation for broader time periods. Large-scale thread-safe data structures constructed and maintained in memory are suitable for efficiently accepting and organizing both cleaned staging data and aggregated data. These data structures also help to remove a large amount of redundant or duplicated information in input data.

The technique of layered key/value pairs is used to implement the in-memory input data dictionary. In this data structure, the value of each key/value pair is a set of key/value pairs, except for the last layer. Figure 2 illustrates an example of the layered key/value pairs (two layers) that is used as the input data dictionary for one day meter reads generated from 3 million meters.

Assuming that each meter read contain four fields: 1) meter ID, 2) timestamp, 3) read type, and 4) read value of either energy or current or voltage, depending on the read type. The set of key/value pairs in the first layer is implemented by thread-safe Concurrent Dictionary [16] in order to avoid data racing. There are 3 million entries in the first layer thread-safe dictionary. For each entry of the first layer key/value pairs, the key is a unique meter ID, and the value is a Sorted List [17]. Sorted Lists cost smaller size of memory comparing to other types of key/value pairs, because a Sorted List stores data in linear arrays but Dictionaries store data in tree structures. For each entry of the Sorted List, the key is a concatenated string. The string is one of the possible combinations of the hour and the minute in a day and the meter read type. The value of each entry of the Sorted List is a float variable that stores the read value of this meter read. During the initialization step, the float variable is set to a special value such as -1, indicating this specific meter read has not been received. When receiving a piece of meter read, the layered key/value pairs allows the data parser to find the corresponding position in the large-scale data structures for this piece of meter read using an O(1) operation. Then the float variable is changed to the read value in this piece of meter read, indicating this specific meter read has been received. With the help of the data dictionary, the data parser is able to organize and sort the whole raw datasets with an O(n) operation while loading them. Duplicated meter reads can be easily detected as well. After this step, the data parser converts input raw data into cleaned staging data and aggregated data and stores all the data in the large scale thread-safe data structures in memory.

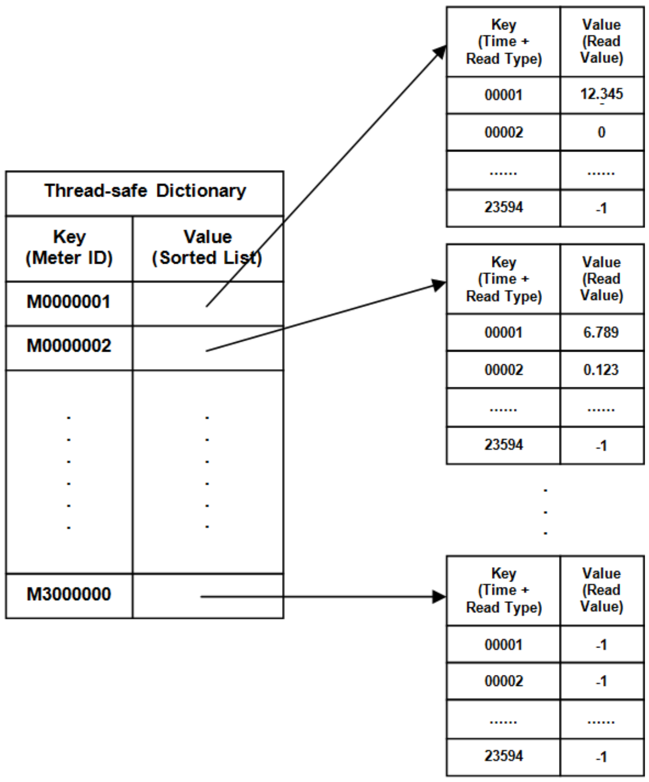

Figure 2. In-memory Data Dictionary Using Layered Key/Value Pairs

Step 5: The well-organized cleaned staging data and aggregated data in memory can be conveniently traversed and stored into relational database management systems and/or flat files on hard drives for future usage.

A key advantage of the design is the ability to receive and process data at a very high throughput. Section 5.2 shows the detailed results of performance evaluation.

4.3 Hybrid Flat-file / RDBMS Storage

The traditional RDBMS stores not only the data set but also related meta-data that are used to accelerate data retrieval. When the size of managed data set is small, the cost and performance overhead caused by the meta-data can be effectively covered because of its performance gain. For instance, the purpose of index is to define a short-cut path to access data. Because of the index, the performance of data retrieval can be significantly improved.

However, the cost and overhead of building, managing and utilizing meta-data are significantly increased when the data size is getting large. Our experiments demonstrated that inserting 1.5 billion meter reads, each of which carries the information on energy consumption and its related timestamp, into a table in an Oracle database, took more than ten hours. By contrast, creating and storing the same amount of data into flat files took less than 10 minutes on the same hardware configuration because of no such meta-data related overhead.

The performance of querying a large relational table deteriorates with the data set getting large. Based on our experiments, a query that conducts the sum operation against a data set with 1.19 million meter reads in Oracle 11g spent more than four hours. While the same operation fulfilled by flat files and streaming took less than six minutes. These experiments demonstrated that the traditional relational database management system becomes inefficiency when the data set is getting large. On the contrary, the flat file technologies become efficient under the same situation.

Nowadays, the size of raw data collected from millions of end devices at a certain time interval (e.g., every 15 minutes or even shorter) is huge and ever increased. In the Extreme Case of the Data File Scenario, the daily data collected from three millions smart meters maximally contains 5.184 billion reads (about 145GB), which already surpasses the size of data that a traditional database application can handle in its life cycle.

Figure 3 illustrates the infrastructure of the newly designed data storage solution, which is suitable for managing a large amount and ever-increasing data. Unlike traditional database technologies, the storage of the database is split into two parts: the relational DB and flat files. The relational DB is used to store aggregated data, which is in small volume, while the flat files are used to store the raw data, which is in huge volume.

The data parser collects meter measurements from data stream and/or data files. After cleaning and aggregation, the data are classified into two parts: the aggregated data and the cleaned raw measurements. The aggregated data, such as the maximal daily or monthly energy usage, are eventually stored to the relational database and the cleaned raw measurements are saved to the flat files. The query engine is used to analyze and redirect the income queries to either the relational DB or the flat files. Generally, the aggregated data stored in the relational DB can answer most queries. For those queries that cannot be handled by relational DB, such as queries on data in long time span have to be extracted from flat files, will be processed by streaming the flat files.

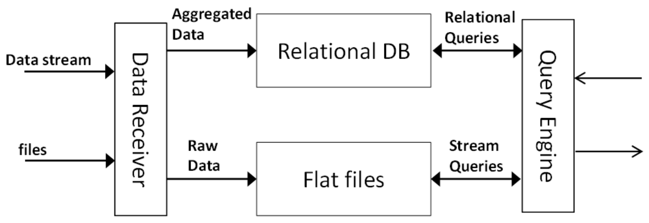

Figure 3. Hybrid Flat-file / RDBMS Storage Infrastructure

The performance of querying large data files eventually overpasses the performance of querying a large relational table when the size of the data set large enough to reach a certain point. In our experiments, conducting an aggregation operation against a large relational table with 0.8 billion reads took 30 minutes (i.e., 1.6 billion reads per hour). Conducting the same aggregation operation by streaming a large file containing 4.8 billion reads spent only 6 minutes (i.e., 48 billion reads per hour).

4.4 Communication Infrastructure

In the *System*, the scenario of transporting data through a communication infrastructure occurs in several places (e.g., transporting the collected meter data from the protocol translation module to the stream parser module and transporting the aggregated meter measurements from the CIS adapter to the CIS system). We abstracted a common communication infrastructure, which can be applied to each individual transportation scenario in the *System*.

The Communication Infrastructure is composed of nodes and bindings. A node has a unified address describing where a message should be sent and logic defining what the message should look like and how the message is sent. A binding is a communication channel decorated by a set of binding elements, which “stack” one on top of the other to create the communication infrastructure. The binding elements can be transportation protocols (e.g., HTTP and TCP/IP), encoding approaches (e.g., text or binary) and other advanced features (e.g., security). Decoupling the node and binding make it easy to combine a node with different bindings. For example, originally a node sends messages through a HTTP binding. As a node is decoupled with a binding, it only needs to construct a new TCP/IP binding and link the node with the newly created binding. In this way, a message can be sent through the TCP/IP communication channel. The permutations and combinations of the binding elements can construct diversified communication channels in reality.

Figure 4 illustrates the queue-based communication infrastructure. Instead of sending a message directly to a receiver, a sender sends a message over a queue to a receiver. The transportation processes that send messages from a sender to a queue and from a queue to a receiver are transaction-based. Therefore, during the transportation, if an error happens, the transportation processes would be rolled back. This makes the communication more stable and therefore, enhances the availability of the *System*. The queue-based communication greatly improves the communication efficiency by saving the communication bandwidth from the sender to the queue.

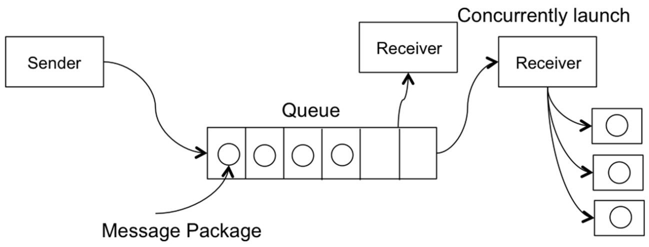

Figure 4. Queue-based Communication

Concurrently processing the received messages is another primary way to improve the throughput of the communication channel. Concretely, on receiving a package, the receiver quickly launches a new thread to process (parsing, cleaning and aggregating) the message, meanwhile the receiver itself starts receiving the next package.

The Windows Communication Foundation (WCF) in .NET 4.5 was used to implement the designed communication infrastructure. Nodes and bindings are available in WCF. Generally a node has a unified address, visible by other nodes from different locations/machines. In the WCF library, various communication channels and their decorations have been implemented. A developer only needs to configure the features of the communication channel, such as communication protocol (e.g., HTTP and TCP/IP) and security facilities, through a configuration file, rather than develop them from scratch.

Two scenarios were primarily implemented in the prototype: queue-based (MSMQ) communication and HTTP-based communication. The former is suitable when both sender and receiver are in the .NET platform, primarily implementing a communication channel with high performance and high availability. The latter emphasizes the interoperability of the communication channel: through HTTP + SOAP, components in the *System* can communicate with the component implemented by technologies other than windows and .NET (such as Linux and Java). In this case, interoperability is the major focus.

4.5 Message Coordinator

From the architectural aspect, it is important for the *System* to extend its capability to interface with the potential systems and applications, which may use different communication protocols and information models. In addition, low cost maintainability is a highly desired feature too.

A message-based publish/subscribe infrastructure, called Message Coordinator, was designed to address the above two requirements. In the publish/subscribe infrastructure, all components are connected through messages. Modifying one component will not influence other components. In addition, each component can have its own special functionalities. The above loosely coupled relationship between components increases the *System*’s maintainability and extensibility. Additionally, unlike the “formal” publish/subscribe architecture, which is across the entire enterprise and supported by a heavy commercial middleware, the proposed publish/subscribe architecture is a lightweight structure that is limited to the *System*, potentially connecting only tens of components.

The publish/subscribe infrastructure is composed of three types of components, as shown in Figure 5: a publisher, a message coordinator and a subscriber. The publisher publishes message to the message coordinator. The message coordinator maintains the relationship between the publisher and subscribers. When a new message arrives, the message coordinator identifies the subscribers of the incoming message and broadcasts the message to them. A subscriber registers itself to the message coordinator for certain messages during the initialization. After registration, the subscriber will receive the registered messages once they are published.

For example, the file parser and the stream parser of the *System* are message publishers, while the CIS connector is a message subscriber. The file parser will send out a message to the message coordinator once it finishes processing bulk data files. On receiving this message, the message coordinator will forward the message to subscribers. By parsing the message, the CIS connector knows that the data has been cleaned up and aggregated. Based on this message, it will pick up the data from the public area (either database or shared memory) and send it to CIS.

It is easy to add a new component to the message-based infrastructure. For example, assume that we want to integrate the *System* with a system that fulfills Demand Response (DR) functionality. A new subscriber, called the DR connector, is developed using Java technologies, which packs the DR information received from the file parser and sends it to the DR system; to connect to the message coordinator, the new message is defined for the DR connector. During the whole process, only the message coordinator and the DR connector are involved. The rest of the system is not affected.

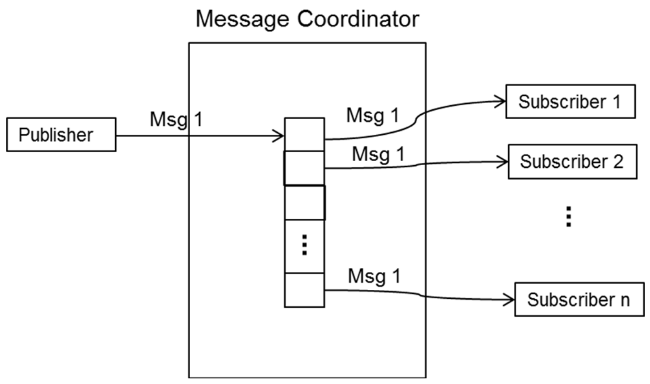

Figure 5. Message Coordinator

The publish/subscribe infrastructure may be considered as the control center of the *System*, which coordinates the status of the components in the *System* through messages. In our prototype, to make the *System* more scalable and extensible, the publish/subscribe infrastructure was hosted in Internet Information Service (IIS) and the HTTP communication protocol was used to transport messages. In such a configuration, the components located in different machines can communicate with the logic of the publish/subscribe infrastructure hosted in the IIS. As a major advantage, hosting the service in IIS automatically utilizes the functions of IIS, such as listening for incoming message and automatically waking up the publish/subscribe logic when the message arrives, greatly reducing the development efforts.

5. PERFORMANCE EVALUATION

In this section, we will present results of performance evaluation of the *System*.

5.1 Testing Environment

Two computer systems were configured for the evaluation. We highlighted our thoughts on selecting hardware components of the testing systems as follows:

- High performance CPU for intensive computation

- High throughput disk for frequent disk I/O

- Four-channel high speed memory for high performance memory operations (e.g., parsing and cleaning meter data in memory)

- Large capacity of memory for pre-storage VEE tasks

Additionally, Visual studio 2012 was selected for fully utilizing advanced data structures (e.g., the lock-free concurrent dictionary), the user-friendly multithreading API, and the latest version of the WCF implementation. Table 2 summarizes the configuration of the testing system.

Table 2. Testing System Configuration

Resource	Specification
CPU	Intel Core i7-3930K @ 3.20GHz, 6 cores with Hyper threads (12 logical cores), 15 M L3
Memory	32GB DDR3 (1600MHz) Four-Channel
Hard Disk	SSD (from Samsung and OCZ)
Network	1Gb/s
OS	Windows 7 Enterprise 64
Dev. Tool	Microsoft Visual Studio 2012 Ultimate

5.2 Data File Parser

To prototype the meter data parser described in Section 4.2, Solid-State Disk (SSD) hard drives [7] were used due to the tremendous disk I/O throughput. The new Task Parallel Library (TPL) [18] in .NET 4.5 was selected to address the high throughput requirement because TPL has easy understanding multi-thread APIs, through which launching multi tasks becomes simple and straightforward.

The input of our experiments is 12 bulk files in CSV format with meter read data including meter ID, Timestamp, read type, and read value. The size of each read (i.e., line) is within 24~29 bytes. Parsing a CSV string into memory variables was the most time-consuming task in the file parser. In comparison with several CSV parsers in state of the art, a fast CSV parser [15] was selected, which was claimed to be the fastest CSV parser in the .NET platform at the time of this study and free for use. Regarding the VEE rules, without loss of generality, a comprehensive survey was conducted against the VEE rules used by primary MDMS products in the current market, based on which we selected and implemented seven popular VEE rules as follows:

(1) *Device ID Validation*: To validate if data is received from a valid meter,

(2) *Timestamp Validation*: To validate if meter reads have valid timestamps,

(3) *Interval Validation*: To validate if meter reads have valid intervals,

(4) *Individual Numeric Data Validation*: To validate if the energy consumption value in each read is within a proper range,

(5) *Summary Numeric Data Validation*: To validate if the total energy consumption value for each meter is within a proper range,

(6) *Duplication Validation*: To validate if the received data has duplicated reads, and

(7) *Missing Data Estimation*: To estimate missing mete read data.

Additionally, the data processing includes a billing aggregation based on Time-of-Use (TOU). The TOU-based billing aggregation calculates the bill charges for each meter for a day according to the pre-defined TOU configuration. The implemented TOU configuration includes the following impact factors: (1) Daylight Saving Time; (2) Season; and (3) Peak

Hours. The input of this data analytics is the 15-minute interval data of consumption for each meter.

Table 3 shows the performance of the Data File Parser based on different input workloads. We conducted experiments five times for each workload. For a testing workload with 0.1404 billion meter reads from 0.3 million meters, it took about 11 seconds to complete the initialization step, i.e., creating and initializing large-scale thread-safe data structures in memory. Then it spent around 71 seconds on completing the whole data processing, including concurrently loading data files, reading and parsing lines, converting text to proper data types for analysis, 7 VEEs, and the TOU-based billing aggregation. For the workload of the *Regular Case*, the initialization time and processing time was 3 minutes 37 seconds and 12 minutes 19 seconds on average, respectively. With regard to the *Extreme Case*, on average it spent 9 minutes 28 seconds on completing the initialization followed by 42 minutes 8 seconds on finishing up the data processing. As a normalization, the experiments demonstrated that in the *Extreme Case* the data parser for data files has a throughput about 7.38 billion meter reads (206.7GB data) per hour (i.e., 1811 TB/year) for parsing, cleaning, and aggregating meter reads data on a single machine with the system configuration shown in Table 2. Unfortunately we did not conduct more experiments to show the linear scalability more clearly due to budget limitations.

Table 4 lists the data processing time for each of the 12 threads for all these experiments. All threads completed their work at almost the same time in the testing case and *Regular Case*. In the *Extreme Case*, the differences between the fastest thread and the slowest thread in these five runs ranged from 31 seconds to 48 seconds. Comparing to more than 40 minutes' data processing time, we still consider that the performance of the data file parser is stable.

Table 3. Performance of Data File Parser

Workload (billion meter reads)	Avg. Initialization Time (min:sec)	Avg. Processing Time (min:sec)	Avg. Processing Throughput (million reads per sec)
0.1404	0:11	1:11	1.9775
1.404	3:37	12:19	1.8999
5.184	9:28	42:08	2.0506

The data parser appropriately allocates the load among various computer resources, such as CPU power, RAM, and hard drives, to achieve best performance (specifically, responsiveness and throughput). The data parser is able to utilize all CPU cores (90+% overall CPU usage) to load the raw data files (or collect the raw data stream) and perform data cleaning and analysis.

5.3 Communication Infrastructure

The permutations and combinations of features of the communication channel generate a variety of communication scenarios (HTTP with no security, HTTP with security and TCP/IP with security). Enumerating and evaluating all of these scenarios is beyond the budget of this study. Therefore, we chose the queue-based communication scenario as shown in Figure 4, which is a relatively complex and highly stable communication approach.

Table 4. Performance of Each Thread for Each Test

Workload	Test	Completion Time for Each Thread (minute:second)
0.1404 billion meter reads	1.1	1:11, 1:11, 1:11, 1:11, 1:11, 1:11, 1:11, 1:11, 1:12, 1:12, 1:12, **1:12**
	1.2	1:10, 1:10, 1:10, 1:10, 1:10, 1:10, 1:10, 1:11, 1:11, 1:11, 1:11, **1:11**
	1.3	1:10, 1:10, 1:10, 1:10, 1:11, 1:11, 1:11, 1:11, 1:11, 1:11, 1:11, **1:11**
	1.4	1:10, 1:10, 1:10, 1:10, 1:10, 1:10, 1:10, 1:10, 1:10, 1:10, 1:11, **1:11**
	1.5	1:10, 1:10, 1:10, 1:10, 1:10, 1:10, 1:11, 1:11, 1:11, 1:11, 1:11, **1:11**
1.404 billion meter reads	2.1	12:03, 12:08, 12:10, 12:11, 12:11, 12:12, 12:12, 12:14, 12:15, 12:15, 12:15, **12:16**
	2.2	12:06, 12:09, 12:09, 12:09, 12:09, 12:10, 12:10, 12:10, 12:12, 12:13, 12:14, **12:16**
	2.3	12:05, 12:10, 12:14, 12:16, 12:16, 12:17, 12:17, 12:18, 12:19, 12:19, 12:19, **12:21**
	2.4	12:11, 12:12, 12:15, 12:15, 12:16, 12:16, 12:16, 12:17, 12:17, 12:18, 12:19, **12:23**
	2.5	12:05, 12:07, 12:09, 12:12, 12:12, 12:13, 12:14, 12:14, 12:15, 12:15, 12:17, **12:18**
5.184 billion meter reads	3.1	41:25, 41:30, 41:30, 41:31, 41:36, 41:42, 41:42, 41:45, 41:45, 41:50, 41:58, **41:59**
	3.2	41:21, 41:26, 41:31, 41:31, 41:31, 41:37, 41:40, 41:41, 41:44, 41:47, 41:50, **42:08**
	3.3	41:25, 41:26, 41:29, 41:35, 41:37, 41:40, 41:42, 41:42, 41:48, 41:51, 42:01, **42:13**
	3.4	41:23, 41:33, 41:40, 41:44, 41:44, 41:48, 41:49, 41:51, 41:56, 41:58, 41:59, **42:03**
	3.5	41:48, 41:53, 41:55, 41:56, 41:57, 41:58, 41:59, 42:08, 42:11, 42:13, 42:13, **42:19**

The queue-based communication architecture is composed of a sender, a receiver and a queue. In the prototype, both sender and receiver were .NET console applications. The queue used the MSMQ 4.0 service, the queue service in the .NET platform. To make the communication more stable, the queue was decorated with the transaction feature. In other words, the communication between .NET consoles (e.g., sender and receiver) and queue was transaction based: if a transaction fails, the delivery would be rolled back.

The sender packed each meter measurement into an XML string. The example of the XML formatted meter measurement was as follows:

```
<Meter>
  <MeterID>abcd1234</MeterID>
  <Timestamp>10:15 4/15/2012</Timestamp>
  <Energy>10.3</Energy>
  <Category>1</Category>
</Meter>
```

A package, the basic unit of the transportation, consisted of 250 of the above XML formatted meter measurements. Transporting 3 million meters data required 12,000 packages, and transporting 5 million meters data required 20,000 packages. The receiver was integrated with VEE functionalities used in the Data File Scenario. The receiver concurrently launched a new thread to conduct VEE and aggregation algorithms on receiving a package.

Table 5 illustrates the performance of the queue-based transportation. On average it took about 1 minute and 40 seconds to transport 3 million meters data between two computers and conduct VEE and aggregation, and about 3 minutes and 12 seconds to transport and process 5 million meters data.

Table 5. Performance of Queue-based Transportation

Workload (records)	Number of Packages	Average Transportation Time (minute:second)
3 million	12,000	1:40
5 million	20,000	3:12

Based on the performance requirements, during the regular meter read phase, 5 million meters measurements arrive in every 15 minutes. Since the queue-based transportation took less than 4 minutes, the requirement of receiving the meter data stream from 5 million meter measurements can be met. Regarding delivering the aggregated meter measurements for 3 million meters data generated in the Data File Scenario to CIS, with the consideration of the transportation time, the total time spent was about 50 minutes on parsing, cleaning, and aggregating, and 1 minute and 40 seconds on transportation, which met the one hour requirement.

In the Regular Case, where the meter data includes regular meter reads and DR reads, the total data that needs to be transferred was (15 + 1) x 3 = 48 million meter reads (15 is the DR reads from 3 million meter in 1 minute). Considering the transfer and cleaning of the 3 million meters data took 1:40, transporting the 48 million meter reads spent about 26.6 minutes beyond the 15 minutes scope. Two alternative solutions for this situation: one is to increase the transportation speed using 10Gb/s network card, which is ten times faster than the current network in theory. The other one is to keep the current setup and increase the size of the queue. In the Regular Case, the three hours DR reads can be temporarily put into the queue and processed after the peak hour.

In the Extreme Case, as the DR reads come in every minute, the queue solution wouldn't work. We had to construct several 10Gb-based communication channels to transport data from one machine to the other. The messaging based architecture allows us to perform such transportation.

6. OTHER QUALITY ATTRIBUTES

Regarding scalability, the *System* is able to be scaled up and scaled out to meet the performance requirements in the future. Scaling up refers to enhancing hardware for the existing machine, while scaling out means extending the *System* across machines. Since the Message Coordinator is deployed in the IIS, and HTTP is used as the primary communication protocol, it is easy to build a system across multiple machines.

With regard to maintainability, the message-based system is composed of loosely coupled modules. When changing the functionality of a module, the source code changes will be restricted only within the module itself, or within the modules that are directly connected to the initially changed module in the modules call graph to adapt the updated messages or newly defined messages.

In terms of reliability, as the components in the architecture are loosely decoupled, it is easy to build "hot" backup for important components. In addition, the queue-based communication can effectively handle the network failures and therefore increase the reliability of the whole M3 system.

Additionally, two aspects of security, authentication and authorization of the local machines or a domain, were implemented by the .NET infrastructure. The communication security is implemented by the WCF.

7. CONCLUSIONS AND FUTURE WORK

AMI systems are an important element of the Smart Grid as they offer an efficient bidirectional communication infrastructure. AMI systems connect millions of end devices with utility control centers and exchange substantial meter data and control information between them in real-time or near real-time. As deployment of AMI systems become more ubiquitous, the amount of smart meters and data handled by these systems continues to grow exponentially. Therefore, it is imperative to design a system capable of collecting, cleaning, analyzing, aggregating and manipulating this data to support smart grid applications and semi-automated decision making.

This paper discusses the development of a lightweight architecture that is able to manage data that originates from millions of smart meters to enhance the capabilities of the Smart Grid. We implemented the prototype system using various concurrency processing techniques, including new Task Parallel Library and latest Windows Communication Foundation, fast CSV parser, in-memory lock-free data structure, layered key/value pairs, hybrid flat-file/RDBMS storage, and SSD, to satisfy critical high performance requirements. Our experiments demonstrated that in the *Extreme Case* the throughput of the data parser for data files is about 7.38 billion meter reads (206.7GB data) per hour (i.e., 1811TB/year) for parsing, cleaning, and aggregating meter reads data on a single machine with the hardware configuration of 12-core CPU, 32G RAM, and SSD Hard Drives. In addition, well-designed publish/subscribe and communication infrastructures ensure the scalability and flexibility of the system.

It is important that the implementation of an AMI system requires incorporating important quality attributes in the system such as maintainability, reliability, and security, besides high performance and scalability. In our future work, we will continue to enhance the architecture to address other quality attributes, and balance the tradeoffs of architectural design.

Another discussion was that, Hadoop [2] might not be very suitable for implementing the *System* mainly because that there is a high risk of not meeting throughput requirements when Hadoop processes small amount of data with a small cluster. Executing Hadoop on a limited amount of data on a small number of nodes may not demonstrate particularly high performance as the overhead involved in starting Hadoop programs is relatively high. Other parallel/distributed programming paradigms may perform much better on two, four, or perhaps a dozen machines. [9][10][28] Hadoop is built to process "web-scale" data on the order of terabytes or petabytes. It is not recommended to use Hadoop if the data and computation fit on one machine. Hadoop requires large footprint to demonstrate its power in processing really huge data. In the Extreme Case, the performance (throughput) goal was to complete processing 5.184 billion meter reads in one hour. Our experiments proved that one computer was enough to process this amount of data. Our design and prototype provide a lightweight way to conduct validation, estimation and editing as well as some analytics in memory for relatively large amount of data. However, it would be more convincing if we have hands-on performance and scalability measurements to compare the Hadoop implementation with our existing implementation. It would be more useful to explore and understand the expected boundaries and scope limitations of the alternate solutions. We will establish Hadoop clusters, write MapReduce programs, and conduct more experiments on Hadoop in the future.

8. ACKNOWLEDGMENTS

We would like to thank our software development teams for their assistance in understanding the requirements of the *System*. We are also very thankful of Marisa Zindler, Kevin Burandt, and Samantha Hines for their expert knowledge.

9. REFERENCES

[1] Agarwal, M. K., Bhide, M. A., Kotwal, S., Mittapalli, S. K., and Padmanabhan, S. March 2011. Parallel Processing of ETL Jobs Involving Extensible Markup Language Documents. U.S. Patent, Pub. No.: US 2011/072319 A1.

[2] Apache. Hadoop. http://hadoop.apache.org/. Retrieved November 2013.

[3] Candea, G., Argyros, A., and Bawa, M. July 2008. High-throughput Extract-Transform-Load (ETL) of Program Events for Subsequent Analysis. World Intellectual Property, Pub. No.: WO 2008/079510 A2.

[4] Chen, X. and Zhang, X. October 2010. Extract-Transform-Load of Data Cleaning Method in Electric Company. In Proceedings of the International Conference on Artificial Intelligence and Computational Intelligence 2010. 345-349.

[5] Dean J. and Ghemawat, S. December 2004. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the Sixth Symposium on Operating System Design and Implementation. OSDI'04.

[6] Edwards, C. A., Johnson, L. M., King, C. S., Prasad, N., Wambaugh, J. O., and Lofgren, T. D. March 2008. Message-Bus-Based Advanced Meter Information System with Applications for Cleaning, Estimating, and Validating Meter Data. United States Patent, Pub. No.: US 2008/0074284 A1.

[7] Ekker, N., Coughlin, T., and Handy, J. January 2009. Solid State Storage 101: An introduction to Solid State Storage. Technical Report. Solid State Storage Initiative (SNIA).

[8] Geschickter, C. August 2010. The Emergence of Meter Data Management (MDM): A Smart Grid Information Strategy Report. Technical Report. GTM Research.

[9] Glover, A. Java development 2.0: Big data analysis with Hadoop MapReduce. http://www.ibm.com/developerworks/java/library/j-javadev2-15/index.html. Retrieved Nov. 2013.

[10] Google. Introduction to Parallel Programming and MapReduce. http://code.google.com/edu/parallel/mapreduce-tutorial.html. Retrieved November 2013.

[11] IEC. 2009. IEC61968-9, Application integration at electric utilities – System interfaces for distribution management – Part 9: Interfaces for meter reading and control.

[12] Itron, "Itron Enterprise Edition™ Meter Data Management", https://www.itron.com/na/productsAndServices/Pages/Itron Enterprise Edition Meter Data Management.aspx. 2011.

[13] Li, Z., Wang, Z., Tournier, J., Peterson, W., Li, W., and Wang, Y. October 2010. A Unified Solution for Advanced Metering Infrastructure Integration with a Distribution Management System. In Proceedings of the First IEEE International Conference on Smart Grid Communications. SmartGridComm'10. 566 - 571.

[14] Lockhart, B. and Gohn, B. 2011. Pike Pulse Report: Meter Data Management - Assessment of Strategy and Execution for 11 Leading MDM Vendors. Tech. Report. Pike Research.

[15] Lorion, S. November 2011. The Fast CSV Reader. http://www.codeproject.com/Articles/9258/A-Fast-CSV-Reader.

[16] Microsoft. ConcurrentDictionary <TKey, TValue> Class. http://msdn.microsoft.com/en-us/library/dd287191(v=vs.110).aspx. Retrieved November 2013.

[17] Microsoft. SortedList<TKey, TValue> Class. http://msdn.microsoft.com/en-us/library/ms132319(v=vs.110).aspx. Retrieved November 2013.

[18] Microsoft. Task Parallel Library (TPL). http://msdn.microsoft.com/en-us/library/dd460717.aspx. Retrieved November 2013.

[19] Mohagheghi, S., Stoupis, J., Wang, Z., Li, Z., and Kazemzadeh, H. October 2010. Demand Response Architecture: Integration into the Distribution Management System. In Proceedings of the First IEEE International Conference on Smart Grid Communications. SmartGridComm'10. 501-506.

[20] National Energy Technology Laboratory. February 2008. Advanced Metering Infrastructure. Technical Report. U.S. Department of Energy.

[21] Network Manager. Retrieved November 2013. ABB Inc. http://www.abb.com/industries/us/9AAC30300663.aspx

[22] Office of Electricity Delivery and Energy Reliability. February 2008. The Smart Grid: An Introduction. Technical Report. U.S. Department of Energy.

[23] Oracle. Meter-To-Cash Performance Using Oracle Utilities Applications on Oracle Exadata and Oracle Exalogic. January 2012. Technical Report.

[24] Ramachandran, V., Hubbard, D., and Skog, J. October 2005. Method and System for Validation, Estimation and Editing of Daily Meter Read Data. United States Patent, Pub. No.: US 2005/234837 A1.

[25] Rustagi, A. September 2008. Parallel Processing for ETL Processes. U.S. Patent, Pub. No.: US 2008/222634 A1.

[26] Siemens/eMeter, "Siemens to Acquire eMeter to Enhance Smart Grid Offering", http://www.emeter.com/company/news/2011-press-releases/siemens-to-acquire-emeter-to-enhance-smart-grid-offering/. December 2011.

[27] Sonderegger, R. May 2010. System and Method of High Volume Import, Validation and Estimation of Meter Data. United States Patent, Pub. No.: US 2010/0117856 A1.

[28] Yahoo Developer Network. Hadoop Tutorial. http://developer.yahoo.com/hadoop/tutorial/module1.html. Retrieved November 2013.