Designing a Reliable and Secure Store for AMI Data

ECE-542: Fault Tolerant Digital System Design Final Project

Kartik Palani, Vignesh Babu

August 17, 2015

Abstract

With the power grid becoming smarter, there is a sudden boost in the amount of data that is being generated by the Advanced Metering Infrastructure that monitors and collects electricity consumption information from consumers. This data needs to be stored reliably and securely. There also exist smart grid applications that require this data in real time to perform accurately. The existing implementations are either slow in delivering real time data or are vulnerable to Denial of Service attacks. In this paper, we describe a reliable and secure hierarchical storage model for AMI data that allows for real time querying.

1 Introduction

The emergence of a smarter electric grid based on advances in information technology and communication calls for an increased need for efficiency, security, reliability and quality of service. A vast majority of the applications in the smart grid are characterized by an exponential growth in data. Data collected from a vast majority of sensors, called smart meters, is used to drive applications such as demand-response, real-time pricing and load forecasting. With a growing demand for this data, there need to be means that allow for fast information retrieval from voluminous data stores. The number of smart meter installations continues to grow every year and now stands at 50 million installed units. A utility company states that its 2 million smart meters produce about 22 gigabytes of data everyday. This data is sampled at 15 minutes from each of the smart meters. These meters however, have the capability to be sampled at a frequency of 1 Hz. Also, currently, utilities extract very less information from the sensors despite the fact that the meters can extract a lot more data about the home electricity usage. In a study we performed, we noted that each meter extracting about 44 electricity parameters from a home every second contributes about 50 MB of data everyday. At the current deployment level that amounts to about 18 GB of data every year from one house. The obvious big issue with this is that large amounts of data needs to be stored but other issues are that data collected should be reliable, data should be accurate and there should be methods for quick querying of data. The data stored can vary from real-time, most recent second data to data archived over years.

In this paper, we propose a model for building a reliable and secure store for Advanced Metering Infrastructure (AMI) data. The main goal of the data store is to provide access to real-time data and to allow for fast querying while maintaining data reliability and security.

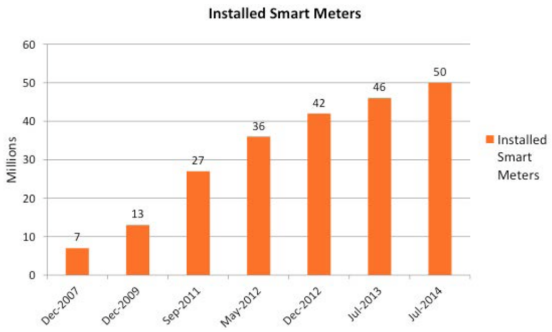


Figure 1: Smart Meter Installations in the US

2 Related Work

A lot of the work in the domain suggests that the best way to deal with this data is to store all the data in a data cloud. Having proven its scalability for Internet-scale data, the cloud seems to be the obvious choice for many researchers. One of the proposed solutions is to build a platform-centric cloud

storage where the smart grid data will be available as a service. Another work carries this thought forward and proposes a hybrid cloud providing Platform as a service and Infrastructure as a service and proves manageability and scalability by applying the model to the Los Angeles power grid. Figure 2 shows the basic architectue of such cloud based storage systems. The biggest drawback of applying this system to the existing AMI architecture is that the data available in the cloud will lag from real data available by at least 30 minutes since in the current implementation by utilities, there is a push to the cloud only every half hour. Thus, real-time applications such as Load Monitoring can not be run on the data.

Another model that has been employed by some of the utility companies is that if there is a query that requires real time data, the smart meters are polled for data prematurely (before their 30 minute push time). The drawback to this method is that there is unexpected load on the bandwidth that occurs due to the request. Also, it becomes easier for an attacker to cause a Denial of Service attack by creating fake requests for real time data that would flood the network.

In our solution, we build on top of the manageability and scalability of cloud storage methods. We also separate the querying server from the cloud such that it receives a broader view of the storage system and hence takes decisions based on prior knowledge of stored data. In this paper, we set out to build a system that not only allows for a large-scale reliable storage but also allows for real-time access to data by applications.

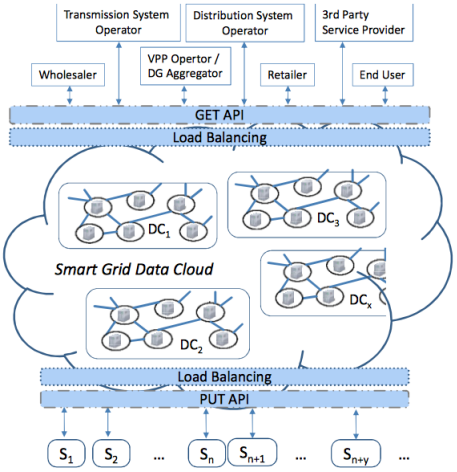


Figure 2: The Smart Grid Cloud Proposed by Rusitschka et. al.

3 The Hierarchical Architecture

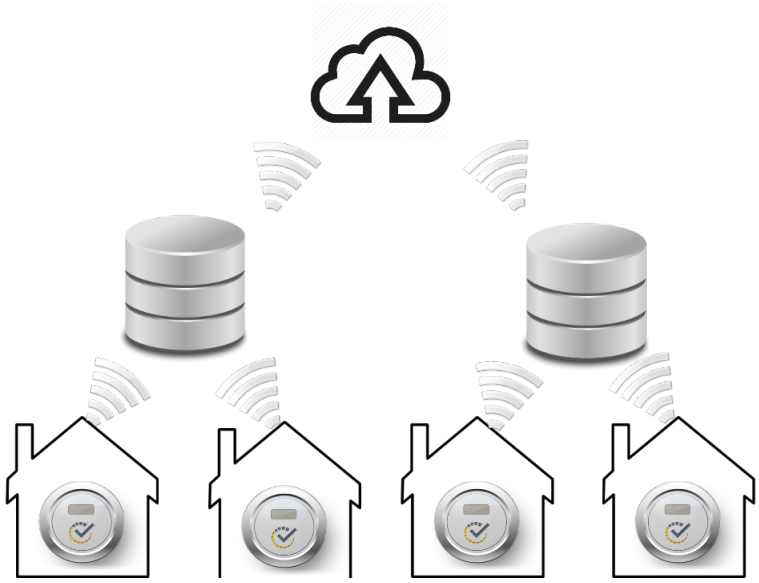


Figure 3: The Hierarchical Storage Architecture

In order to achieve real-time query responses in a data store, applications need to have the ability to directly access the local storage at various levels of the system. We draw inspiration from computing systems, where the cache stores a small amount of data but allows for fastest access, followed by the memory where more data can be stored but access is relatively slower and finally persistent memory where large volumes of data can be stored but loading data takes longer.

We propose a hierarchical architecture as shown in Figure 3. The model employs a Publish-Subscribe type communication mechanism. The hierarchy consists of three levels:

- Home Area Storage (HAS): Smart Meters come with inbuilt storage capabilities. However, since they are embedded devices they are not built with a large storage space. In our design, we understand this storage constraint and only store data at each home for about 15 minutes before pushing it to the next level. This level stores real-time data upto the nearest second.
- Network Area Storage (NAS): Every neighborhood has a data aggregator which subscribes to data from the smart meters (which act as publishers) in its vicinity. Every 15 minutes they recieve the newly published data from its publishers. It stores data for a day before pushing

it out to the cloud. This level contains data at a lag of 15 minutes.

- Cloud Storage: The neighborhoods publish their data to the cloud where the data stays forever. We used a lot of existing research in the area of cloud storage to implement resiliency in the cloud in our implementation.

Since each of these levels of storage have different requirements for reliability and security, we build these parameters in separately for each level. The details of implementation at each of these levels is explained in the sections below.

4 Home Area Storage

The core infrastructure of AMIs is based on smart electric meters installed in homes and industries. Every home in the US is expected to be equipped with a smart meter by 2020. A smart meter has the following capabilities:

- real-time or near-time registration of electricity use and possibly electricity generated locally e.g., in case of photovoltaic cells
- offering the possibility to read the meter both locally and remotely (on demand)
- remote limitation of the throughput through the meter (in the extreme case cutting of the electricity to the customer)
- interconnection to premise-based networks and devices (e.g., distributed generation)
- ability to read other, on-premise or nearby commodity meters (e.g., gas, water)

These smart meters are in essence low storage, low processing embedded devices. Having understood that, our querying mechanism makes sure that the queries at this level require less processing power. Also, we store data in the meters only for 15 minutes which translates to about 54,000 entries in the database. Thus, the smart meter does not have to do a lot of work to find an entry when a query comes in.

In our implementation, we use Raspberry Pi development boards to emulate smart meters (since we did not have access to real meters). We picked this board because it has an ARM processor with 256 Mb of memory and 8Gb of storage which is similar to the configuration found on many of these meters. In order to make sure that the data being generated was accurate, we fed in data from last September obtained from one house and the Raspberry Pi published this data with new timestamps every second.

4.1 Reliability in HAS

At the smart meter level, it is very important that the system stay reliable because a failure at this level means loss of data from a whole house till the meter gets fixed. Hence, we decided to implement system redundancy at this level in order to guarantee high availability.

Triple Modular Redundancy is used to build redundancy. There are three Raspberry Pi devices that collect data from the sensor and pass it on to a voter which then stores the data in a SQL database.

There is a hot spare that is kept active. If the voter feels that one of the modules in the TMR has failed, it brings up the hot spare and reports a failure to the administrator. This way, there can be a new spare that can be installed in case of a failure. Figure 4 shows the setup used to monitor consumption at one house with three modules used for the TMR and one used as a hot spare.

Every time a new module is added to the setup, the module is authenticated by the query server (discussed later in the paper). This involves an exchange of a certificate that was issued by the utility. We used Python's PyCrypto library to implement a digital certificate issuance and PKI between the node and the Query server.

When the data needs to be published, the node creates a secure tunnel (TLS based) with the subscriber and then transmits data from its database. This makes sure that there is no passive or active man in the middle attack.

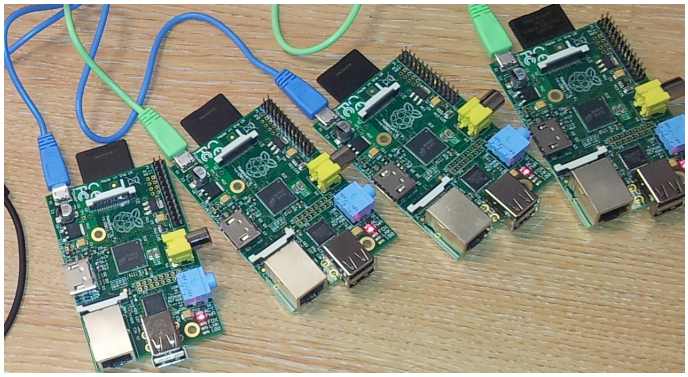


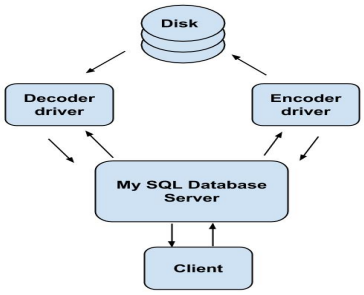
Figure 4: The HAS storage implementation

5 Network Area Storage

Traditional AMI networks have groups of smart meters interacting with each other using a neighbourhood area network. Smart meters relay power consumption and energy parameters of their respective houses to an aggregator node. The aggregator node in turn would immediately relay the received information to a reliable cloud storage database. There are however, some inherent bandwidth and latency related issues with this approach. Smart meters usually push data in small batches at intermittent intervals of upto 30 minutes. This means that the data stored at the cloud could potentially lag behind the actual real time data by upto a period of 30 minutes. Further, data is transferred in small batches which are only a few kilobytes in size and the associated protocol overhead is often non negligible. To circumvent these problems, we propose the design of a Network Area Storage (NAS) system which acts as an intermediary data store. The purpose of the NAS is multi-fold. It collects data bursts from smart meters and stores them on a local redundant array of disks, for a predefined longer period of time. Subsequently, the stored data is flushed to the cloud storage in large batches. Data transmission in large batches raises the possibility of introducing efficient compression schemes to improve overall network utilization. NAS also functions as relay node which can relay queries and from applications to smart meters or even answer queries thereby allowing the applications to process near real time metering data.

We modelled NAS as a RAID like system with redundant array of disks. It uses a SQL database to store metadata and supports erasure coding schemes to ensure data reliability even during the event of multiple disk failures. The general architecture is shown in the figure given below. Our implementation contains uses a MySQL server which interfaces with clients (smart meters/ query servers) and Encoder and Decoder drivers. The server process is capable of accepting and handling simultaneous concurrent connections from multiple distinct clients. It supports two request "ADD" and "RETRIEVE".

ADD requests are followed by a key value pair, with the key being a function of client ID and a unique record identifier like the current timestamp. ADD requests are usually issued by the smart meters within the neighbour hood area network. RETRIEVE requests on the other hand only specify the key to be retrieved. When the NAS server receives



(a) *NAS architecture*

an ADD request, it adds an entry to the MySQL database with the received key as the primary key and the value being the location and identifiers of the file fragments which would be generated by the encoder. After adding the entry to the database, the encoder is instantiated and it generates different file fragments and stores them in the specified fragment locations. Upon receiving a RETRIEVE request, the MySQL database is queried to retrieve the location and fragment identifier information which are then fed into a Decoder. The decoder automatically detects all disk failures and decodes the available data fragments into the original value for transmission back to the client.

We chose to use a relational database for our implementation because the queries are often structured and relational databases can support complex queries. Each NAS is expected to store only a limited amount of data which is periodically flushed. The maximum amount of data to be stored is usually in the order of only a few Gigabytes which falls well within the capabilities of a traditional RDBMS. In the following sections, we discuss the erasure coding components and communication components of the storage system.

5.1 Erasure coding in NAS

We considered 4 different erasure codes in our implementation and evaluated them based on some defined metrics. Jerasure [1] and liberasure [3] are standard open source erasure coding libraries which were used in our design. In this section we briefly, describe each erasure coding technique and highlight its strength and weaknesses in terms of three metrics : storage efficiency, average encode latency and average decode latency. Many research works also consider the ease of recovery in their evaluations but we chose to skip this because recovery mechanisms are often not time critical in the AMI infrastructure because data is

only received in bursts after a reasonably long time. Storage efficiency of the erasure coding scheme was calculated as the size of the original data to be stored divided by the total size of all stored data. Mean decode and encode latencies were measured by initializing the client and server on the same machine to eliminate network overhead, and subsequently sending ADD and RETRIEVE requests to estimate the response times.

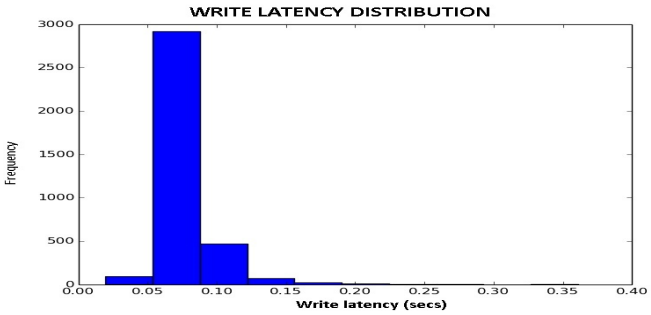
We considered 3 MDS codes : Reed Solomon Vandermonde, Cauchy Reed Solomon (CRS) and RAID-6 and a non-MDS code : flat XOR based HD combination code in our evaluations.

5.1.1 Reed Solomon Vandermonde code

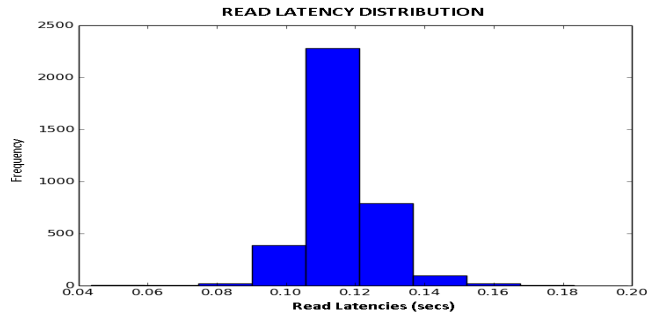
Classical (n,m) Reed Solomon codes generally use a (n+m x n) encoder matrix A to encode data blocks. The matrix A is often constructed with the concatenation of an n dimensional Identity matrix with a special distribution matrix. The distribution matrix is chosen so that any n rows of A are linearly independent and hence the code can support upto m disk failures. Reed Solomon Vandermonde codes [2] use Vandermonde matrices as their distribution matrix. It makes the code highly flexible and storage efficient but encoding and decoding procedures require expensive vector multiplications and inversions in Galois field Arithmetic. The following histograms show the read and write latency distribution for the erasure code.

5.1.2 Cauchy Reed Solomon code

On the contrary to the previous code, Cauchy reed solomon codes use a cauchy matrix [] as their distribution matrix. The advantage of using a cauchy matrix is that the Galois field arithmetic size can be reduced to as low as possible and is not constrained to a few discrete values as in the previous code. This speeds up the expensive multiplication and inversion operations. Jerasure also introduces some optimizations described in [5], where each element in the field is projected into a bit vector. This essentially converts the encoding matrix into a bit matrix and replaces all operations with XORs. Cauchy Reed solomon codes are however not very storage efficient because they divide each data block further into smaller padded packets and encode each packet using a bit coding



(a) *Write latency distribution*



(b) *Read latency distribution*

Figure 6: *Reed Solomon Vandermonde code*

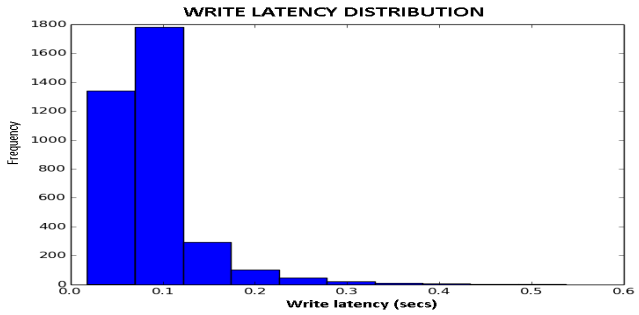
matrix.

5.1.3 RAID-6 codes

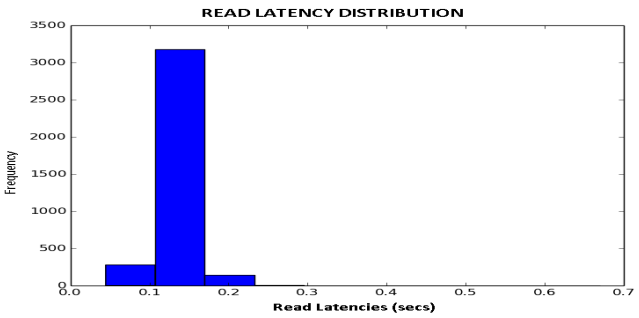
RAID-6 codes traditionally support two disk failures. One of the coding block is simply computed as the XOR of all data blocks whereas the other coding block is computed as the XOR of each data block multiplied by a the corresponding power of 2 in Galois field arithmetic (Anvin's optimization [7]). This special type of multiplication operation can be realized efficiently. Hence, RAID-6 achieves faster read and write latencies compared to the previous two schemes.

5.1.4 Flat XOR – HD codes

They are non MDS codes in the sense that an increment in the number of coding blocks does not necessarily increase the number of tolerable disk failures. Flat XOR based combination codes are described in [4], and they are computed as a special combination of subset of disks. The subsets are chosen so that the recovery process is quick. Unlike the previously discussed MDS codes, to achieve a tolerance of a given number of disk failures, Flat XOR codes require many more redundant disks and the code is thus highly inefficient in terms of storage efficiency. However, encoding and decoding operations are much faster than

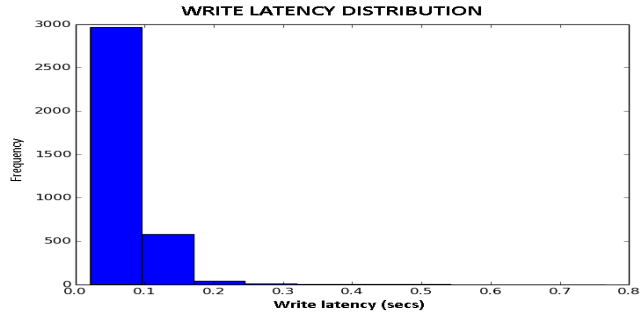


(a) *Write latency distribution*

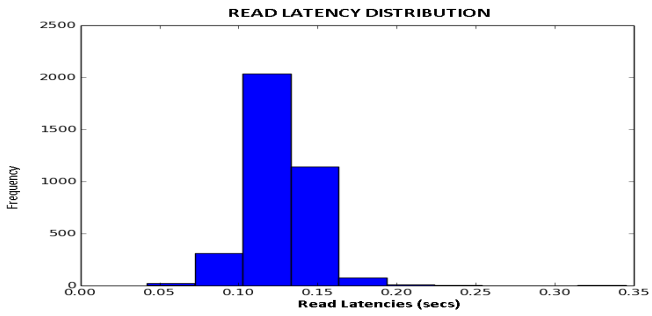


(b) *Read latency distribution*

Figure 7: *Cauchy Reed Solomon code*



(a) *Write latency distribution*



(b) *Read latency distribution*

Figure 8: *RAID-6 code*

the other 3.

5.2 Evaluation metric

The following table shows the average read, write latencies and storage efficiencies of each erasure coding scheme discussed before. For ease of direct comparison, all erasure codes were configured to support exactly 2 disk failures.

Table-1 : Parameter values			
Erasure code	Storage Efficiency(%)	Mean Read Latency (sec)	Mean Write Latency(sec)
ReedSol Vandermonde	33.82	0.077	0.116
CRS	18.11	0.087	0.128
RAID-6	37.45	0.085	0.127
Flat XOR	23.64	0.067	0.117

As expected, Flat XOR codes achieve the fastest read and write latencies. All the other MDS codes seem to have comparable access latencies primarily because our configuration supports only 2 disk failures and the encoder matrix was sufficiently small. So we think that the Field arithmetic overhead did not manifest itself in the access latencies. The storage efficiencies on the other hand are reasonably different.

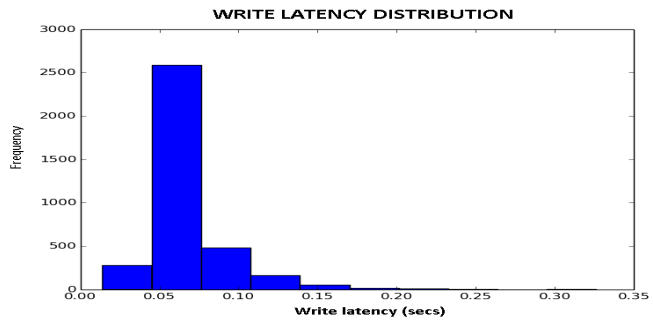
To evaluate the erasure codes we considered the average access latency (avg of read and write latencies) and a cost factor. Let C be the cost per bit of storage and let SE be the storage efficiency. Given a data of size S, and a storage efficiency of SE, the total cost of storage equals (S*C)/SE.

$$Evaluation\ metric\ =\ cost\ of\ storage/Mean\ request\ latency.$$

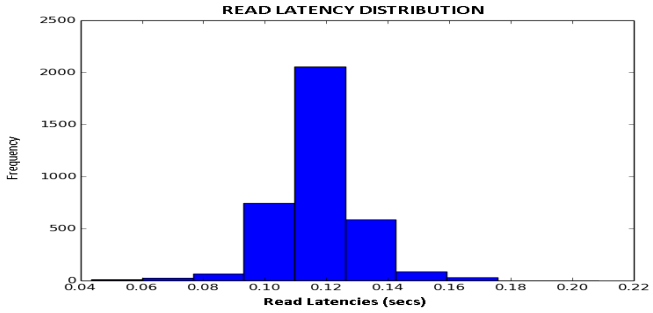
It quantifies the price paid to achieve a given request latency. (Cost to benefit ratio). Since, S and C are constants, they are simply removed from the equation. The table containing the value of the metric for different erasure codes is given below :

Table-2 : Metric scores	
Erasure code	Score
ReedSol Vandermonde	30.64
CRS	51.36
RAID-6	25.19
Flat XOR-HD	45.97

From the table, we find that RAID-6 codes are the best suited because they have the lowest cost



(a) *Write latency distribution*



(b) *Read latency distribution*

Figure 9: *Flat XOR HD code*

to benefit ratio. Hence our implementation uses RAID-6 codes for redundancy.

5.3 Communication component

The communication module of the NAS creates a secure communication channel for interaction with all the other components. Both encryption and message integrity checks through PKI, is supported by our current prototype implementation.

6 Query Server

The Query Server interfaces with the Application and routes queries to different hierarchical nodes. When each hierarchical node flushes its database to the higher layer, it sends a flush message to the query server after the flush is complete. The flush message at the smart meter level consists of the minimum and maximum identifiers of the flushed data. The flush message at the NAS level does not include any identifiers but it merely serves as an indication of the start of a new epoch.

We implement an epoch based routing update algorithm where the routing table is refreshed at the start of each epoch and continuously updated during the epoch. The Query server builds a routing table based on the minimum and maximum identifiers of batch transfers at the smart meter level. The routing continuously maintains the minimum and maximum identifier from each node during an epoch and it uses three simple rules to route queries to the correct node. The routing algorithm is shown below :

The routing algorithm takes in a node id and the query/identifier/timestamp as its input and determines where the value is located.

Algorithm 1 *ROUTE*[*Query*]

```
while True do
  if new epoch started then
    while current epoch is running do
      Maintain node[min_timestamp], node[max_timestamp]
      if node[Query] < node[min_timestamp]
      then
        Send Query to Cloud
      else
        if node[max_timestamp] ≥
        node[Query] ≥ node[min_timestamp]
        then
          Send Query to NAS
        else
          Send Query to node
        end if
      end if
    end while
    Reset node[min_timestamp], node[max_timestamp]
  end if
end while
```

7 Cloud level storage

The third layer of our hierarchical storage system comprises of the cloud. A NoSQL database is employed at the cloud and listens to query and flush requests from the query server and the NAS respectively. NAS nodes periodically flush all their stored data once every 24 hours, to the cloud and notify the query server after the process is completed. During the flush process, NAS nodes do not accept any new incoming connections to maintain strong consistency. Our prototype implementation uses a MongDB cluster which was setup on the Google cloud. We chose to use NoSQL databases at the cloud level because of their fast query processing times and the ability to handle large numbers of clients.

8 Heartbeat Server

Error detection is an important aspect in a large scale system. The administrators need to be aware of device failures in order to perform regular replacements of failed systems and hard disks. We implemented a heartbeat server to monitor the heartbeat from every node in the system. We developed a scalable multithreaded heartbeat monitor that tracks the liveliness of all nodes in the infrastructure.

We use UDP based packets since, we can afford the loss of a few packets once in a while given that the timeout is large enough compared to the heartbeat messages being sent. This is a scalable solution compared to TCP based monitors since we save on precious bandwidth and we lower the load of acknowledgement messages on the server. We took special care to maintain synchronization between the threads in order to guarantee that no heartbeat is missed at the system level.

Figure 10 shows a block diagram of an implementation similar to ours:

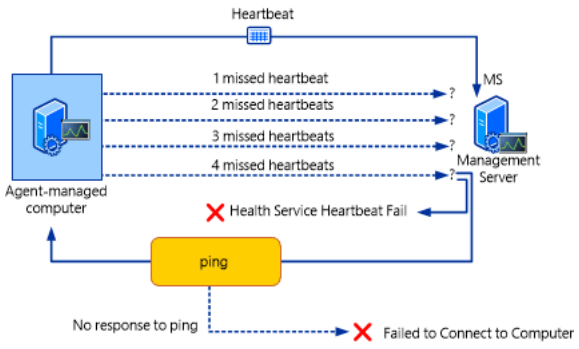


Figure 10: The Heartbeat Monitor

9 Performance

We evaluated our system against 5 kinds of queries:

1. Data in Home Area Storage
2. Data in Network Area Storage
3. Data in NAS and HAS
4. Data in the Cloud
5. Data in Cloud and HAS

These queries were run on two systems. One system was our implementation of the hierarchical storage model and the second one was the existing architecture followed by utilities where the data is stored directly in the cloud and if a query for real time data comes then, there is a pull issued to data in the smart meter. The results are illustrated in Figure 11. The results show the average query latency of 50 queries of each type.

We noticed that for all real-time queries, our system performed significantly better than the existing infrastructure. However, for query 4 where the data queried was directly from the cloud, the existing architecture performed better. Our average response time to query 4 was 1.2 seconds and we feel this an acceptable number since a query to the cloud directly means that the data requested is archived data and hence some delay in issuing such data is acceptable.

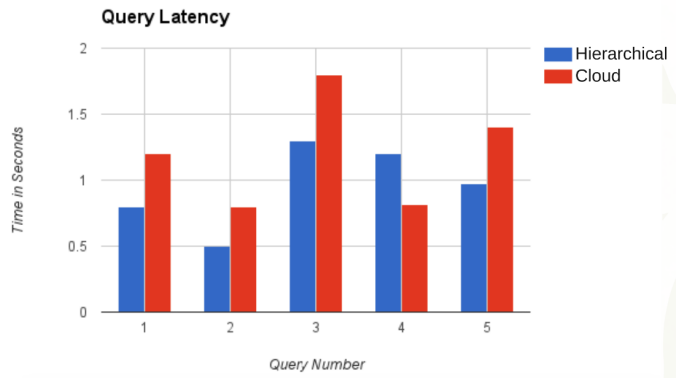


Figure 11: Performance of the hierarchical system against the existing cloud storage system

10 Future Work

The results we show are from our implementation of the hierarchical storage. In order to guarantee such results, we need to create a simulation of the storage system and the querying server. Another aspect that we did not stress on is the network redundancy. We hope that adding network redundancy will make the system highly resilient and secure. We also plan to improve the performance of the querying server by applying fast query processing algorithms.

11 Conclusion

In this paper, we describe a reliable and secure hierarchical storage model for AMI data. We motivate the need for such a model due to the existence of smart grid applications that require real time data to perform accurately. The existing implementations are either slow in delivering real time data or are vulnerable to Denial of Service attacks. We then implement

the hierarchical model using Raspberry Pi boards and build into the system reliability and security. Our final results prove that the model we described provides low latency for real time queries and is secure against disk and system failures.

12 References

1. Plank, James S., Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

2. Plank, James S. "Erasure Codes for Storage Systems: A Brief Primer." ; login: the Usenix Magazine. Vol. 38. 2013.

3. https://bitbucket.org/tsg-/libe

4. Greenan, Kevin M., Xiaozhou Li, and Jay J. Wylie. "Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs."Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010.

5. Plank, James S., and Lihao Xu. "Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications." Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on. IEEE, 2006.

6. Dimakis, Alexandros G., et al. "Network coding for distributed storage systems."Information Theory, IEEE Transactions on 56.9 (2010): 4539-4551.

7. Plank, James S., Adam L. Buchsbaum, and Bradley T. Vander Zanden. "Minimum density RAID-6 codes." ACM Transactions on Storage (TOS) 6.4 (2011): 16.

8. Rusitschka, S.; Eger, K.; Gerdes, C., "Smart Grid Data Cloud: A Model for Utilizing Cloud Computing in the Smart Grid Domain,"

9. Qian Wang, Kui Ren, Shucheng Yu, and Wenjing Lou. 2011. "Dependable and Secure Sensor Data Storage with Dynamic Integrity Assurance."

10. Moslehi, K.; Kumar, R., "Smart Grid - a reliability perspective,"

11. Rodrigues, R.; Liskov, B.; Chen, K.; Liskov, M.; Schultz, D., "Automatic Reconfiguration for Large-Scale Reliable Storage Systems," Dependable and Secure Computing, IEEE Transactions on , vol.9, no.2, pp.145,158, March-April 2012

12. Pavan Edara, Ashwin Limaye, and Krithi Ramamritham. 2008. Asynchronous in-network prediction: Efficient aggregation in sensor networks. ACM Trans. Sen. Netw. 4, 4, Article 25 (September 2008)

13. IEI report, Utility-scale Smart Meter Deployments: Building Block of the Evolving Power Grid

14. Yogesh Simmhan, Saima Aman, Alok Kumbhare, Rongyang Liu, Sam Stevens, Qunzhi Zhou, Viktor Prasanna, "Cloud-Based Software Platform for Big Data Analytics in Smart Grids," Computing in Science and Engineering, vol. 15, no. 4, pp. 38-47, July-Aug., 2013

15. Bo Sheng, Qun Li, and Weizhen Mao. 2010. Optimize Storage Placement in Sensor Networks. IEEE Transactions on Mobile Computing 9, 10 (October 2010)