A Dynamically Scalable Cloud Data Infrastructure for Sensor Networks

Tonglin Li[1], Kate Keahey [2,3], Ke Wang[1], Dongfang Zhao[1], and Ioan Raicu [1,2]

[1]Department of Computer Science, Illinois Institute of Technology, Chicago, IL, USA
[2]MCS Division, Argonne National Laboratory, Lemont, IL, USA
[3]Computation Institute, University of Chicago, Chicago, IL, USA

ABSTRACT

As small, specialized sensor devices become more ubiquitous, reliable, and cheap, increasingly more domain sciences are creating "instruments at large" - dynamic, often self-organizing, groups of sensors whose outputs are capable of being aggregated and correlated to support experiments organized around specific questions. This calls for an infrastructure able to collect, store, query, and process data set from sensor networks. The design and development of such infrastructure faces several challenges. The challenges reflect the need to interact with and administer the sensors remotely. The sensors may be deployed in inaccessible places and have only intermittent network connectivity due to power conservation and other factors. This requires communication protocols that can withstand unreliable networks as well as an administrative interface to sensor controller. Further, the system has to be scalable, i.e., capable of ultimately dealing with potentially large numbers of data producing sensors. It also needs to be able to organize many different data types efficiently. And finally, it also needs to scale in the number of queries and processing requests. In this work we present a set of protocols and a cloud-based data streaming infrastructure called WaggleDB that address those challenges. The system efficiently aggregates and stores data from sensor networks and enables the users to query those data sets. It address the challenges above with a scalable multi-tier architecture, which is designed in such way that each tier can be scaled by adding more independent resources provisioned on-demand in the cloud.

1. INTRODUCTION

The last several years have seen a raise in the use of sensors, actuators and their networks for sensing, monitoring and interacting with the environment. There is a proliferation of small, cheap and robust sensors for measuring various physical, chemical and biological characteristics of the environment that open up novel and reliable methods for monitoring qualities ranging from the geophysical variables, soil conditions, air and water quality monitoring to growth and decay of vegetation. Structured deployments, such as the global network of flux towers, are being augmented by innovative use of personal mobile devices (e.g., such as use of cell phones to detect earthquakes), use of data from social networks, and even citizen science. In other words, rather than construct a single instrument comprised of millions of sensors, a "virtual instrument" might comprise dynamic, potentially ad hoc groups of sensors capable of operating independently but also capable of being combined to answer targeted questions. Projects organized around this approach represent important areas ranging from ocean sciences, ecology, urban construction and research, to hydrology. This calls for an infrastructure able to collect, store, query, and process data set from sensor networks. The design and development of such infrastructure faces several challenges. The first group of challenges reflects the need to interact with and administer the sensors remotely. The sensors may be deployed in inaccessible places and have only intermittent network connectivity due to power conservation and other factors. This requires communication protocols that can withstand unreliable networks as well as an administrative interface to sensor controller. Further, the system has to be scalable, i.e., capable of ultimately dealing with potentially large numbers of data producing sensors. It also needs to be able to organize many different data types efficiently. And finally, it also needs to scale in the number of queries and processing requests. In this paper we present a set of protocols and a cloud-based data store called WaggleDB that address those challenges. The system efficiently aggregates and stores data from sensor networks and enables the users to query those data sets. It address the challenges above with a scalable multi-tier architecture, which is designed in such way that each tier can be scaled by adding more independent resources provisioned on-demand in the cloud.

2. DESIGN AND IMPLEMENTATION

2.1 Design considerations

Write scalability and availability: The system needs to support many concurrent writes from a large sensor network, which continuously captures data and sends it to the cloud storage. The system should be always available for writing. For achieving these goals, we propose to use a

multi-layer architecture. A high performance load balancer is used as the first layer to accept and forward all write requests from sensor controller nodes evenly to a distributed message queue, which works as a write buffer and handles requests asynchronously. A separate distributed Data Agent service keeps pulling messages from the queue, preprocess it and then write to the data store.

The capability to present various data types: There can be many different kinds of sensors in a sensor network, each of them can read different number and types of values. There is no fixed scheme for data formats from the different types of sensors. Therefore we need a flexible data schema so to enable a unified API to collect and store the data, and to organize data in a scalable way for further use query and analytics. To address this issue, we design a flexible message data structure that easily fits into a large category of scalable distributed databases, called column-oriented databases (or BigTable-like data stores). This design enables us to elevate the rich features, performance advantage and scalability from column-oriented databases, as well as to define a unified data access API.

2.2 Architecture

We design a loosely coupled multi-layer architecture to boost the scalability while maintaining good performance. As shown in fig 1, the system is composed of a sensor controller node and a data server that is both written to by the sensors and read from by the clients. On the server side, there are 5 layers of components, namely load balancer, message queue, Data Agent, database, and query execution engine. Each layer can be deployed on a dedicated or shared virtual cluster. If any layer becomes bottleneck, it can be scaled easily by simply adding more resource.

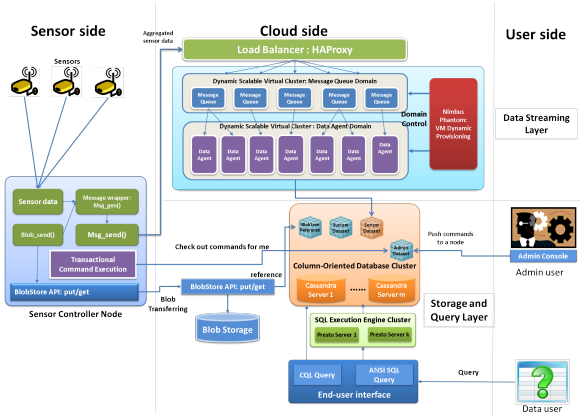

Figure 1: System architecture. Sensor controller nodes send messages and blobs to the cloud storage through APIs. Load balancer forwards client requests to data streaming layer. Nimbus Phantom controls dynamic scaling of queue servers and data agent servers.

2.2.1 *Sensor Controller Node*

The sensor controller node accepts data captured from sensors and wraps into a basic message. The client can also send multiple messages in batch through a single transfer as needed. When the clients need to send a big file such as an image by a full-spectrum camera, it will first send the blob through our API to the cloud blob store(such as Amazon S3) or our own file-based blob storage system, and then send a reference message to the database. This reference message is organized in the same way as the basic message, which contains all the metadata, but in the data field, it holds a URL link or a pointer to that file. All these communication are enhanced by our transactional transferring mechanism.

2.2.2 *Designing dynamic scalable services*

All component layers in our system are organized as scalable services, most of which can scale in a dynamic and automatic manner. To allow them to do so, a couple of requirements have to be satisfied [1]. First, the parallel instances of a service can run on multiple machines independently. This ensures a service can be scaled out. Second, a new instance of the service should be able to discover and join a running service. This ensures live scaling of a service, which means the service can scale any time on demand without halting.

Load balancer service: A load balancer is used on top of the whole architecture. The load balancer does not only balance the workload, but also offers a single access point to the clients so as to hide the potential change (such as scaling or failure) in the message queue layer. Load balancer is setup on a big virtual machine.

Message queue service: The message queue service is asynchronously replicated across multiple VMs, which ideally are located close to each other. We choose Availability and Partition tolerance from the CAP theorem [2] and assume that the connection between the VMs is reliable, which is reasonable within a single cloud provider. In this way, any single failed queue won't cause any data loss. A new message queue server can join a cluster easily. We used a simple script to setup and start new message queue service, join the cluster and update the load balancer config file, and then run a reload on load balancer server to finish the system scaling.

Data Agent service: The only job that a Data Agent does is pull messages from message queues, preprocess them, and then push to the storage service. There is no communication and dependency between any two Data Agents. For adding new Data Agents, users only need to tell the new agent where to pull messages and where to push to, which can easily been set as start parameters. Thus both requirements are satisfied.

Storage service: To meet the various needs of sensor network applications, we design a hybrid storage service that combines a column-oriented database [3] and a blob store. Most of sensor readings are small, and can be put into the database while big files are send to the blob store. For each blob in the blob store, there is a tuple in the database, through which the blob is presented as a regular sensor reading. Thus users can access the data via a unified API.

2.2.3 *Design discussion*

In This work we use dedicated message queue service (RabbitMQ) and data storage (Cassandra) in order to provide best response time(or latency). One possible alternative solution is to use cloud services such as Amazon SQS and DynamoDB respectively. Uses of cloud service can simplify the system implementation and deployment. However this convenience is at no cost. As we observed before, the response time of both SQS and DynamoDB are multiple times slower than most of user deployed software services. Economic cost is also a big concern.

2.3 Transactional command execution

In sensor networks, administrators often need to carry out diagnosis and system maintenance by running a series of commands remotely. Conventional remote login such as SSH or Telnet won't work as desired because of the unreliable communication channels. The command execution subsystem must be able to recover from most of the interruptions and communication failures. For solving this problem, we designed and implemented a transactional protocol and stateful data middleware to track the command execution sessions and to persist the results. When a user needs to execute a series of command, s/he firstly sends an execution request to the middleware, which assigns an incremental session ID to the request and then forwards the request to a dedicated database table on cloud. The commands will not be executed immediately. Instead, controllers check out the available commands from database periodically and then execute them sequentially. We use pull method on controllers instead of opposite, because the cloud side doesn't know if and when a usable network connection is available, so it must be the controller that starts a communication and get the commands. If a controller finds more than one sessions in the database, it will firstly execute the session that with a smallest ID. As the commands are executed on a controller, the results are given a sequence number and push to the database. The user can query the database any time to see if there is any result available. Since both commands and results are persisted in the database, the data lose caused by connection failure is minimized.

2.4 Implementation

We have implemented the sensor controller client, user query client, administration client, data agent servers and all the adaptors between components in Python [4]. We choose RabbitMQ as message queue server, and Cassandra [5] as the column-oriented database for storage backend. For dynamic scaling on various tiers of the system, we adopted Nimbus Phantom [1], an automatic cloud resource manager and monitoring platform, which enable us manage each tier independently. This work has been integrated into its ongoing parent project, Waggle, at Argonne National Laboratory. The whole system is currently running with real sensors and collecting environmental data.

3. EXPERIMENTAL RESULTS

As we writing this paper, the Waggle sensor network is still in development and doesn't have many sensor controllers. So we used synthetic benchmarks to evaluate our system's performance and functionality on a public science cloud, FutureGrid. This method actually simulated the worst case of the real-world scenario: all sensor controllers happen to send data at the same time, while the normal case is that they send data in a random manner. We claimed that WaggleDB can handle highly concurrent write requests and provide high scalability. To demonstrate this we evaluated the system with up to 8 servers and 128 clients, in terms of latency, bandwidth, client/server-side scalability and dynamic scaling. We used Tsung benchmarking tool to generate 128 clients on up to 16 virtual machines, and ran up to 8 servers on other virtual machines. We sent 10,000 write requests in a tight loop from each client to the load balancer, which then forwarded requests to queue servers concurrently. Each request was a fixed-length string message.

3.1 Concurrency and client-side scalability

To determine the capability of handling concurrency and client-side scalability of WaggleDB system, we measured average latency, aggregated data bandwidth and throughput with 1 server. The request latency consists of the time spent on data transferring and request processing. To understand the latency composition better, we measured the latencies with different message sizes, from 10 bytes to 10k bytes. Figure 2 shows that while the client scale increased by 32 times, the average latency only increased by 2.2 times. This implied great potential of client-side scalability. Note that the experiments were conducted with client-side tight loops that were only bounded by CPU performance and network bandwidth. Since the real-world clients in sensor networks generally only contact servers occasionally, it's safe to claim that the single-server system can handle many more than 32 real clients. It's worth noting that the latency differences between 10 to 10k bytes message sizes were very small ($<$ 20%). Within measured message size range, latency was not sensitive to message size. This implied that request processing (open/close socket, acknowledgement) takes more time than data transferring within 10k bytes range.

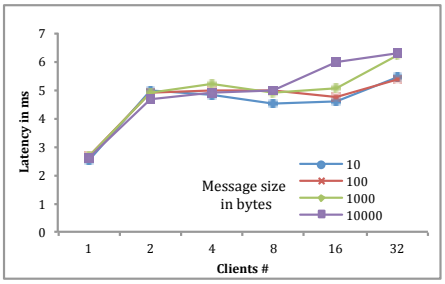

Figure 2: Average latency only slowly increased with client number.

3.2 Concurrency and server-side scalability

To determine the server-side scalability and measure the overall performance, we fixed the message size to 1000 bytes and performed similar experiment with up to 128 clients and 8 servers. As more clients joined, the latencies of all server scales increased as shown in figure 3. The more the servers were used, the less the latency increased. On the single-node system, latency started to increase rapidly on 32-client scale and above. This suggested that single server got saturated when serving more than 32 clients.

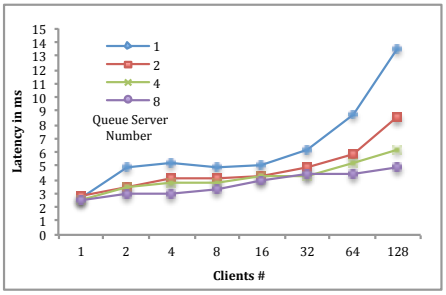

Figure 3: Latency comparison. The more the servers were used, the less the latency increased.

3.3 Dynamic scaling

To verify the functionality and performance impact of dynamic scaling, we measure the latency while scaling queue server and Data Agent server layers on the fly. We firstly tested queue server layer. We fixed the number of clients to 128, started the experiment with single-queue server, ran 30 seconds, doubled the queue servers, and repeated. Eventually we scaled queue server layer to 8 nodes. In figure 4, the column chart shows the average latency decreased significantly. Take single-server system as a baseline, scaling to 2, 4 and 8 server brought 36%, 55% and 64% performance gain respectively. The curve chart shows the real-time latency in logarithmic scale. The three high peaks were caused by the load balancer configuration reloading (new list of servers).When adding more than 4 queue servers, the latency decreased slower, because it already close to the ideal value and system was nearly idle.

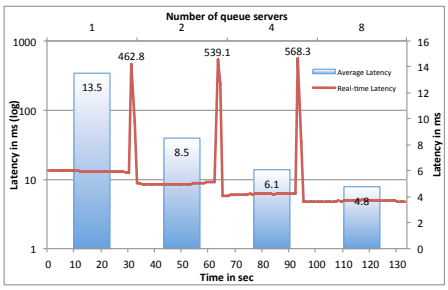

Figure 4: Real-time and average latency on dynamic scaling queue servers.

4. RELATED WORK

Our work adopts known techniques and extends previous work on data model and storage. However we are not aware of other implemented systems that cover versatile data model, scalable storage, transactional interaction, and especially dynamic scalability on each system tier. The communication and storage pattern used in our system can be found in other types of systems, such as system state management [6], which was more designed to handle relatively constant workload. On scalable distributed storage systems for cloud and data intensive applications, there have been works done on NoSQL databases, file systems. On data storage tier, we use NoSQL databases instead of conventional SQL databases, due to their limited scalability on multi-node deployment. There are various types of NoSQL databases for different applications: Key-value stores (ZHT [7] [8]), document stores (MongoDB), and column-oriented databases (BigTable [3]), etc. There are also works done on distributed file systems that support very high concurrent read and write such as FusionFS [9]. Similar as queue service in WaggleDB, Liu's work utilize burst buffer [10] in scalable parallel file systems. ON data streaming perspective, JetStream [11] enables event streaming across cloud data centers.

5. CONCLUSIONS

In this work we present a set of protocols and a cloud-based data streaming infrastructure called WaggleDB that address those challenges. The system efficiently aggregates and stores data from sensor networks and enables the users to query those data sets. It address the challenges for accommodating sensor network data streams in cloud with a scalable multi-tier architecture, which is designed in such way that each tier can be scaled by adding more independent resources provisioned on-demand in the cloud. The featured high availability and scalability, flexible data scheme and transactional command execution make it a good candidate for sensor network data infrastructure in cloud era.

6. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation grant NSF-1054974. This work used the Future Grid test bed funded by the National Science Foundation under Grant No. 0910812.

7. REFERENCES

[1] K. Keahey, P. Armstrong, J. Bresnahan, D. LaBissoniere, and P. Riteau, "Infrastructure outsourcing in multi-cloud environment," FederatedClouds '12, pp. 33–38, ACM, 2012.

[2] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, pp. 4:1–4:26, June 2008.

[4] T. Li, K. Keahey, R. Sankaran, P. Beckman, and I. Raicu, "A cloud-based interactive data infrastructure for sensor networks," IEEE/ACM Supercomputing/SC'14, 2014.

[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," SIGOPS, 2010.

[6] T. Li, I. Raicu, and L. Ramakrishnan, "Scalable state management for scientific applications in the cloud," BigData Congress '14, pp. 204–211, 2014.

[7] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," IPDPS '13, pp. 775–787, 2013.

[8] T. Li, R. Verma, X. Duan, H. Jin, and I. Raicu, "Exploring distributed hash tables in highend computing," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, pp. 128–130, Dec. 2011.

[9] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "Fusionfs: Towards supporting data-intensive scientific applications on extreme-scale high-performance computing systems," IEEE BigData'14, 2014.

[10] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," MSST'12, 2012.

[11] R. Tudoran, O. Nano, I. Santos, A. Costan, H. Soncu, L. Bougé, and G. Antoniu, "Jetstream: Enabling high performance event streaming across cloud data-centers," DEBS '14, 2014.