Implementation of Failover Handling of Broker Network in Big Active Data

CS237 Project Final Report

Team 6: Ali Nickparsa, Yoshimichi Nakatsuka, Yuya Shiraki

June 13 2018

1 Introduction

While the majority of today's Big Data systems are passive, there is a widespread need for a scalable system that continuously and reliably captures the Big Data and facilitates the timely and automatic delivery of new information to a large population of interested users. Big Active Data project [1] meets the mentioned requirements by extending the AsterixDB, a scalable open-source Big Data Management System which offers native data storage and indexing as well as querying of datasets in local files that enables efficiency for smaller as well as large queries. In addition, it has an open data model that handles complex nested data as well as flat data, and a full query language that supports declarative querying over multiple data sets. Along with AsterixDB as the data cluster, the BAD system consists of data sources (Data Publishers) and end users (Data Subscribers), and it handles notification management and distribution by utilizing a broker network. The system must have extensible support for a variety of high-volume data feeds so that data from many different Data Publishers can be continuously and reliably ingested and stored for use by BAD applications. Given a set of registered data interests from the universe of Data Subscribers, the BAD platform must capture and monitor their potentially many interests against the evolving state of the stored data – in an efficient, shared, and scalable manner – in order to detect the need to generate and send out new notifications. To that end, the platform can provide a notion of information channels, based on application-defined parameterized queries, that can be subscribed to by users. When new notifications are generated by the BAD Data Cluster, they must be disseminated to their target users. The data distribution responsibilities of the BAD platform are handled by a network of cooperating BAD brokers which manage the system's connections and communications with end users and their devices. The current BAD platform uses an extended version of AsterixDB which consists AQL - A rich and declarative query language, a scalable distributed dataflow platform, rich data types and fast data feeds. Brokers provide a go-between layer between the BAD Data Cluster and the actual, geographically distributed, end users of a given BAD application. Based on the interests of the users, BAD application manager will create a set of channels – continuous queries with parameters. The end users will subscribe to a channel via the broker node they are connected to. The broker nodes are responsible for handling client (called BAD client) registration, managing subscriptions and delivering results for those subscriptions. Each broker has two parts: a "client-facing" part managing the clients and an "Asterix- facing" part handling interactions with the Asterix backend. A simple workflow of interaction between a client and the broker is as follows: The client registers via the broker and logs in. The client then subscribes to one or more available channels using desired parameter values, which are passed to the Asterix backend by the broker. The backend notifies the broker when new results are populated in the subscribed channels. The broker, in turn, notifies the client, and the client acts to fetch the results as desired. Broker nodes are implemented as RESTful servers written in Python using the Tornado web framework.

2 Motivation

It is essential to maintain a continuous and reliable information feeding to subscribers. Thus, the possible failures which affect the quality of subscriptions must be handled. Every broker has a potential to fail due to disasters, hardware issues, software errors, or human errors. Having the resilience in the system to detect the failure as well as to automatically recover from that is crucial to continuous feeding of information to subscribers in a timely manner. The current implementation is susceptible to broker failure, due to lack of implementation of coordination service for brokers. Suppose that a broker fails, and its connected user will need to connect to other broker and resume its subscription. Avoiding a coordination service will tend to essence for storing all the brokers IP addresses which requires additional storage at the client's side and message overhead due to continuous sending of brokers' IP addresses in case of change. Hence, by implementing a Broker Coordination Service (BCS), when a new broker node joins the broker network, it registers with the BCS server. After registration, the broker can accept clients for possible subscriptions to channels. A client connects to the BCS server to receive the address of the broker to which it should connect for subsequent services. In this project we propose a failover handling mechanism of a broker to have resiliency in the broker network. We view it beneficial not only for having resiliency but also for other aspects such as load balancing and handoffs because some of the features built in our project can be reused.

3 Related Work

Many system rely on automatic failover handling mechanisms to gain resilience against failures and achieve high availability of the system. In fact, systems we daily use implement some aspect of such mechanisms. In this section, we explore different techniques of failover handling proposed in real world services in the context of the following three aspects: 1) Replication, 2) Failure Detection, and 3) Recovery from failure.
Chubby [2] is Google's lock service which also provides a reliable storage for distributed systems, used in many of Google's services such as Google File System [3] and Big Table [4]. A Chubby instance (also known as *cell*) consists of several servers called *replicas*. The members of the cell elect a master using an election protocol. Chubby realizes replication by putting a copy of the database in all replicas and by having the update to the database be propagated through the cell from the master using a consensus protocol. To implement a failure detection mechanism, Chubby makes each client in the cell sends a *KeepAlive* message to the master to check it is up and running and the master replies to that message to advocate that it is still up and running. When a master is suspected to have failed, the remaining replicas in the cell re-elect a master.
Dynamo [5] is Amazon's highly available key-value store. It is used in Amazon's e-commerce platform to implement the company's cart service which must be designed to withstand tens of millions of cart checkouts during the high season. Dynamo implements replication by forming storage hosts into a virtual ring and replicating data across multiple storage hosts on the ring. Similar to Chubby, Dynamo uses a message-and-response based failure detection. Storage hosts in the ring periodically send messages to each other to check whether they are still up and running. In the case a host is suspected to have failed, the old host is simply removed from the ring and a new host is added to replace the old host.
Ceph [6] is a distributed file system developed by researchers at UCSC. It is designed to achieve high performance, reliability, and scalability. Ceph first divides a file to a list of objects and then maps those objects into *placement groups* (PGs) using a hash function. Each PG is assigned to multiple hosts using another hash function. To accomplish replication of the file, each object derived from the file is stored into the multiple hosts assigned to the PG. Ceph makes its hosts to actively monitor whether others are still up or not. When a host has not heard from another host for some time, it sends out a explicit ping to determine that it has actually failed. Similar to Dynamo, as soon as the host is determined to have failed, a new host is added to replace the failed host.
Zookeeper [7] is a service that provides coordinating processes for distributed applications. It is used in many distributed systems such as Yahoo! crawler, Yahoo! Message Broker, and Hadoop [8]. One of the services that Zookeeper can provide is failure detection. Zookeeper provides its users an abstract set of data nodes (known as *znodes*) that is organized based on a hierarchical name structure. Users are able to update the data objects of the znodes and receive update of the changes through the Zookeeper API. Using this feature, Zookeeper can be used to maintain a global view of the active nodes and receive automatic notifications whenever there is a change in the active nodes.
There have been also many research on handling failovers in literature. XNET [9] proposes a crash/failover scheme in a situation where a pub/sub system recovers from a crashed router. Routers which are placed

downstream to the crashed router suspect that a router has crashed after a long timeout. After suspecting that the crashed router will not recover soon, the downstream router connects to a different backup router and replicates its subscription information to that router. Kazemzadeh et al. [10] handle broker failures in distributed pub/sub systems by having all brokers share a replicated copy of the topology of the network. Once a broker detects a connection loss and suspects that the broker that it was connected to has failed, it then tries to create a connection with the next available broker that is one hop away according to the topology. If that connection attempt fails, it will try to connect to the next neighbor.

4 System Design

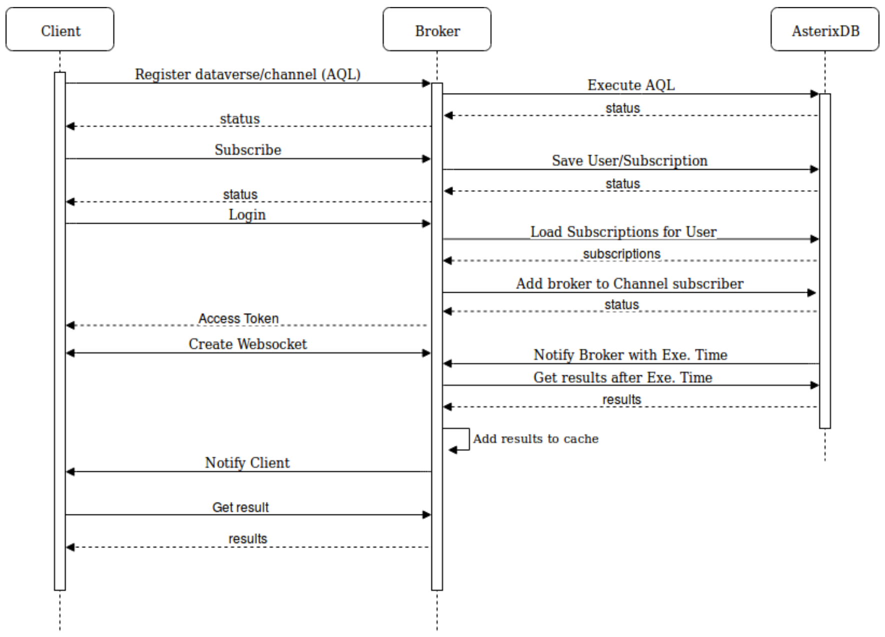
In this section, we explain how the BAD network is implemented and what features we added to implement our failover mechanism. The implementation are done in addition to the BAD demo implementation[11].

Figure 1a shows the initial design of the call flow of the BAD network. Since the figure shows all the details, we will mention only the overall features and the problems we noticed while understanding the call flow. The main feature we noticed was that the broker nodes do not hold any of the information of the clients or their subscribed channels. Apparently what the broker does is that it loads all the information as a form of AQL and sends it to AsterixDB. When the broker is notified that there are new results for their subscribed channel, the broker caches the result for future use and then sends the result to the client. The problems we noticed during our code understanding phase were: 1) The IP address of the broker in the client side code was hard coded, and 2) The broker sends all the results that are stored in its cache when the *Get result* function is called by the client.
In order to implement our proposed failover feature, we modified the original code and added features and functions as shown in Figure 1b. The modified parts are highlighted in red. We mainly implemented the following three features.

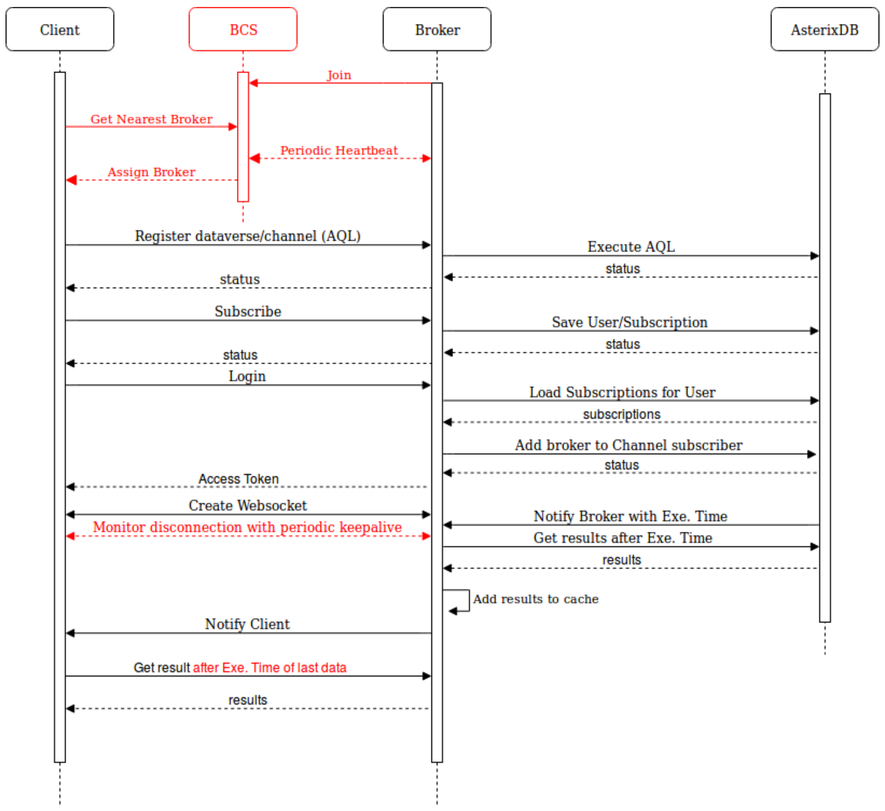
- We made use of the Broker Coordination Service (BCS) (namely, *Broker Discovery*) code. All brokers will first register themselves to the BCS when they initialize. When the BCS receives a request for a broker from a client, it sends back the information of the nearest broker back to the client. While this is running on a process, another process periodically sends a heartbeat to check whether all registered brokers are up and running.
- In our demo, we used the web version of the client, so each client and its assigned broker are connected via Websocket. We modified the client side code so that it monitors the connection state of the Websocket using a periodic *keepalive* call.
- When the client calls the *Get result* function, it sends a parameter *Execution Time* with it. This makes the broker send all cached data **after** the Execution Time, therefore preventing duplicate copies of data.

In an environment with two brokers, one client and a BCS, the failover mechanism works as follows.

- The Client is connected via Websocket to Broker1. Broker1 fails for some reason.
- The BCS sending out the heartbeat packet to all the brokers with a certain timeout interval detects that Broker1 has failed and deletes Broker1 from its active broker list.
- Meanwhile, the Client monitoring a status of the Websocket by sending out a keepalive packet with a certain timeout interval detects disconnection.
- The Client sends a request to the BCS for another broker to connect.
- The BCS hands the information of Broker2, as it is the only remaining broker in the list. This ends the failover sequence.



(a) Original call flow of BAD network



(b) Call flow of BAD network with added features

Figure 1: System Call Flow

5 Evaluation

When a broker fails, clients connected to the failed broker is going to detect the disconnection, discover a new broker via asking BCS, and reconnect to the newly assigned broker. In order to achieve the seamless failover, as explained in the motivation section, the delay from when a broker fails to when the client finish establish a new connection has to be as small as possible.

5.1 Issues faced during implementation and their solution

The following issues were faced during implementation.

1. Environment setup
2. Debugging distributed system
3. Designing additional REST calls
4. Implementation of background procedure

The difficulty in the first issue is that choosing an appropriate environment for running AsterixDB, Brokers, and clients. AsterixDB required more than 3GB memory and several dependencies including Apache Maven and a certain Java version for compilation. Running Broker required superuser privileges. For the solution, our team decided to have a desktop machine and use SSH to develop with a team.
The difficulty in the second issue is that running multiple process at the same time and finding out the issue causing errors. For the solution, the team built Docker environment to run multiple processes at the single host, and used Tmux screen to monitor multiple processes at the same time. Also, TCPdump was used to find out there are messages with intended data passed between containers.
The difficulty in the third issue is related to adding BCS or heartbeating requires new calls to be made. For the solution, as shown in System Design Section, we wrote the entire call flow within AsterixDB, Broker, and Client. We added BCS into it to decide what REST calls are needed in between each node.
The difficulty in the fourth issue is that periodic heartbeat or monitoring connection status requires background procedure to run in addition to main procedure. There are several ways to solve these issues including the use of timer wheel called IOLoop in Python Tornado, using multi-threaded process, and using interupts. For the solution, we used IOLoop for a client and multi-threads for a broker.

5.2 Evaluation Plan

In order to evaluate the whole system in the synchronized clock, this project used Docker environment built on top of a desktop machine. The specification of tested environment is as shown in Table 1.

CPU	Intel Core i7-3770 CPU 3.40GHz
Memory	16GB
Disk	1000GB
OS Version	Ubuntu 14.04.5 LTS
Docker Version	18.05.0-ce
Container Image	ubuntu 14.04, 578c3e61a98c

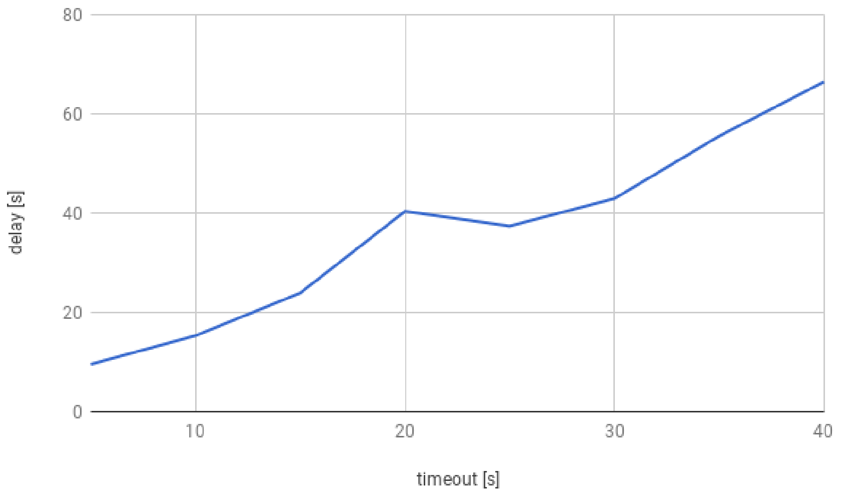
Table 1: Simulation Environment

The simulation environment setup consists of 1 AsterixDB, 1 BCS, 2 Brokers, and multiple clients all running in a distinct container. The private IP network is built to connect these containers, and all the node talks to each other. Initially, Brokers know IP addresses of Brokers and AsterixDB, and clients know an IP address of BCS.

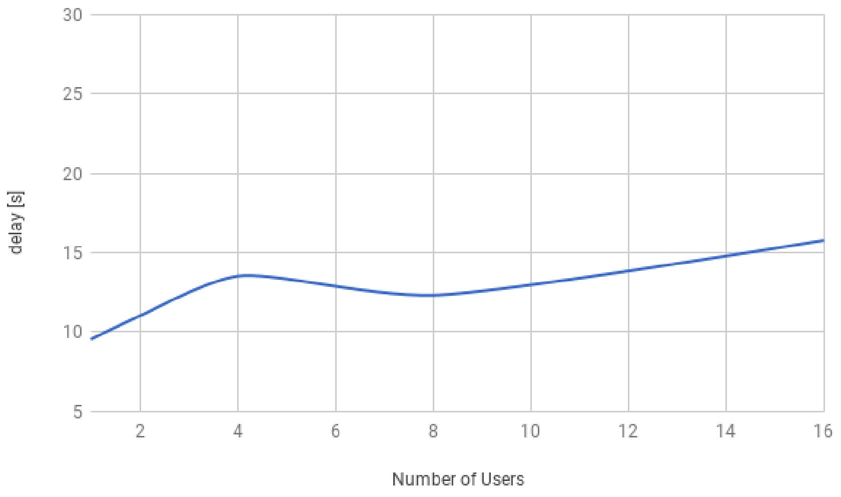
5.3 Simulation Result

The first evaluation is on the relationship between a re-connection delay of a client and a timeout period a client uses for checking a status of a Websocket connecting a broker and a client. In this evaluation, the timeout period in BCS to heartbeat all the registered brokers are fixed to 5 seconds, and the timeout of a client is increased from 5 seconds to 40 seconds.

The second evaluation is on the relationship between a re-connection delay of all the clients and the number of clients connected to a single broker. In this evaluation, the number of clients is increased as 1, 2, 4, 8, and 16.
The results of respective evaluations are shown in Figure 2.



(a) Re-connection delay vs. timeout period of a client



(b) Re-connection delay vs. number of clients to handover

Figure 2: Results of evaluation

5.4 Analysis of results

Figure 2a shows correlation between timeout and delay. In other words, as the timeout increases, the delay between failure of broker to re-connecting on a new broker also increases. The reason of this behavior is obvious because if a timeout period is longer, it also takes longer to detect the disconnection of Websocket at a client and initiate the re-connection. Also, sometimes update of brokers list in BCS is taking long time due to a need for TCP timeout to decide whether connection is down or not, which is causing clients to be assigned with dead brokers which causes another timeout period to decide whether connection has failed or not. However, it is not as simple as just making timeout period smaller because processing the heartbeat so

frequent would also cause BCS and brokers to have a higher load and may cause delay for processing main procedure. Therefore, memory and CPU usage may need to be measured along with increase of timeout periods.
Figure 2b does not really show any correlation between the number of clients to be handovered at the same time and re-connection delay. It is because BCS and Broker have enough capacity to process multiple requests at the same time. Using the REST API simplifies re-connection procedure, which may be contributing to expand the capacity. However, if we increase the number of client to hundreds or thousands, having only one BCS and concentration of new clients' connection to a single broker would hit networking hardware or software bottleneck to process requests. Therefore, in addition to the work of this project, a load balancing at BCS or Brokers have to be carefully considered.

6 Conclusion and Future Work

In this project, we propose an automatic failover handling mechanism for the brokers in the Big Active Data (BAD) system to provide continuous flow of information to the clients. Our failover handling mechanism consists of two parts; failure detection and failure handling. The Broker Coordination Service (BCS) periodically sends a heartbeat message to the brokers to check the broker's status and to update the broker list it holds. As soon as the client detects disconnection between him and the originally connected broker (failure detection), it sends a message to the BCS to request for an information about a new broker that he can then reconnect (failure handling).
Evaluations have shown that while timeout and failover delay may have an obvious correlation, we have pointed out that there is a trade-off between the decrease of failover delay and increase of heartbeat handling load. In addition, though we have shown that our failover mechanism could withstand a small amount of clients, we must take other mechanisms into consideration when improving the scalability of our scheme. Overall, we consider the following as possible future work.

- Have the clients store the time of the last received result from the broker and create a continuous flow of results even if experiencing failure of broker
- Implement a decentralized failover handling mechanism and compare it with our centralized mechanism based on metrics such as scalability, performance, and complexity
- Explore other functions that may benefit our scheme such as load balancing and handoff mechanisms of clients

References

[1] M. J. Carey, S. Jacobs, and V. J. Tsotras, "Breaking bad: a data serving vision for big active data," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM, 2016, pp. 181–186.

[2] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 335–350.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, *The Google file system*. ACM, 2003, vol. 37, no. 5.

[4] 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 Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS operating systems review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[6] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[7] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9. Boston, MA, USA, 2010.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 2010, pp. 1–10.

[9] R. Chand and P. Felber, “Xnet: a reliable content-based publish/subscribe system,” in *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE, 2004, pp. 264–273.

[10] R. S. Kazemzadeh and H.-A. Jacobsen, “Reliable and highly available distributed publish/subscribe service,” in *Reliable Distributed Systems, 2009. SRDS’09. 28th IEEE International Symposium on*. IEEE, 2009, pp. 41–50.

[11] M. Y. S. U. Steven Jacobs and e. a. Michael Carey, “A bad demonstration: Towards big active data,” in *Proceedings of the VLDB Endowment*, vol. 10. VLDB, 2017, pp. 1941–1944.