Implementation of the Load Balance Algorithm in the BAD System

Dixin Zhou(dixinz), Minghe Zhang(minghez2), Meng Yuan(yuanm9)

1. Introduction

Over the past three decades, the low cost of hardware, the development of communication technology and the growth in speed of computer have enabled computers to deal with much more resources. In a distributed system, processors can serve more clients and manage large-scale communications. It is possible that some processors get overloaded while others are idle in a distributed network. It becomes imperative to solve this problem as the Internet grows explosively. We need to come up with a technique that can manage the processor loads and keep them balanced. Load balance can improve the overall system performance by distributing the load appropriately. The load strategy makes it possible to force processors equally busy and finish at approximately the same time. Besides the benefits of performance improvement, it also prevents small jobs from long starvation, shortens the response time and elevates the throughput.

The big active data (BAD) system is a scalable system that continuously and reliably captures big data and deliveries the information to interested users automatically. Load balancing algorithms are globally classified into two categories: static and dynamic ones. Based on the existing algorithms, we implement two load balance algorithms on the BAD system, a static one and a dynamic one. The interactions among clients, brokers, and the database are discussed at first to give an overview of the system. Then the load balancing algorithm and its implementation are illustrated. To validate the effectiveness of our algorithm, a series of controlled experiments have been performed, one experiment adopts the random selection, one experiment adopts the static load balance algorithm and another one adopts the dynamic load balance algorithm. The three experiments are compared based on the mean of the standard variance of the broker loads. The experimental results show that our load balance algorithm can distribute broker load effectively and help the system achieve relative balance.

This paper is organized as follows: Section 2 introduces load balancing algorithms, the AsterixDB and the BAD system. Section 3 discusses the system design decisions and gives a system overview. Section 4 describes implementation details of the system. Section 5 demonstrates three experiments and presents experimental results and analysis. And we conclude our work and some future directions in Section 6.

2. Related Work

For years, the field of load balancing issues in distributed systems has drawn many researchers' attention. Abundant load balancing algorithms and strategies are proposed to try to issue the unevenly distributed load in distributed systems. For example, the Round-Robin algorithm[1] [2] is a classical one to handle the load balance issue. Besides, many other useful algorithms are proposed, such as randomized algorithms[3], central manager algorithms[4] and threshold algorithms[5]. Various load balancing strategies is proposed based on the basic algorithms described above, for example, the dual threshold load balancing method proposed by Talukder[6] is a modern version of the threshold algorithm. At present, load balancing algorithms can be categorized as static or dynamic, centralized or decentralized, cooperative or non-cooperative, topology-dependent or topology-independent in the literature[7]. In this paper, we intend to study the load balancing algorithms as static or dynamic. In the static load balancing(SLB) algorithms, load balancing decisions are made deterministically based on system-related probabilistic knowledge during the compile time, and they will remain unchanged during the runtime. The advantages of SLB algorithm are its formal mathematical analysis and simple implementation, while the drawback of SLB algorithms is its inability to adapt to the changing system states. In order to overcome the weaknesses of the SLB algorithms and to allow for more adaptions to the changing system status, the dynamic load balancing(DLB) algorithms have been put forward. Contrary to SLB algorithms, DLB algorithms make decisions at runtime based on the system states at that point and no decisions will be made during the compile time. When a processing node is overloaded, tasks will be migrated from the heavily loaded computing node to a lightly loaded one. The advantages of the DLB algorithm are that no previous knowledge of the system behavior is required, and the system will be less suffered from the degraded performance resulted from the uneven load

distribution and achieves elevated throughput and shorter response latency. While one disadvantage of the DLB algorithms is that there can be much overhead, communication overhead, operational overhead and transportation overhead are unavoidable.

This BAD system we work on extends an existing scalable open-source Big Data Management System(BDMS), called AsterixDB, thus the AsterixDB system will be briefly introduced before the introduction to the BAD system. As a wealth of available digital information and people's awareness of the fortune that can be brought by the large amount data, the NSF-sponsored Asterix project is set out to develop a next-generation system to ingest, manage, index, query and analyze mass quantities of semi-structured data[8]. AsterixDB uses Dataverse, Datatypes and Datasets to describe, store and manage internal Big Semi-structured Data, and offers external data adaptors to support direct access to externally resident data. And AsterixDB adopts Asterix Query Language(AQL) to query and manipulate data. The architecture of the AsterixDB roughly consists of cluster controller and node controller with or without metadata, the cluster controller is responsible for receiving users' queries via an HTTP-based API and returning the results to the requesters either synchronously or asynchronously. The cluster controller also translates the AQL statements to job descriptions which then are dispatched to node controller for execution.

The BAD system will now be introduced. Virtually many of today's Big Data systems are passive in nature, meaning that they will return result only when there is a user request, while the BAD system is an active system, suggesting that BAD system can continuously and reliably capture Big Data to enable timely and automatic delivery of new information to a large pool of interested users as well as supporting analyses of historical information[9]. The BAD system consists of BAD data cluster, application administrators and BAD brokers[10]. The BAD data cluster is responsible for data storage, the application administrators are responsible for channel creation and management, and the BAD brokers are the links between the BAD data cluster and end users. End users subscribe channels along with parameters to brokers, data publishers feed the data into the BAD data cluster, and whenever there is an update of the data concerned with users' interests, the AsterixDB will notify the brokers first and then the brokers will notify particular users and return the updated information at users' requests.

3. Design

In this section, the main structure of the load balancing project in the BAD system is introduced, including the project architecture, the transaction flow, the implementation framework, and the load estimation approach.

3.1 Architecture

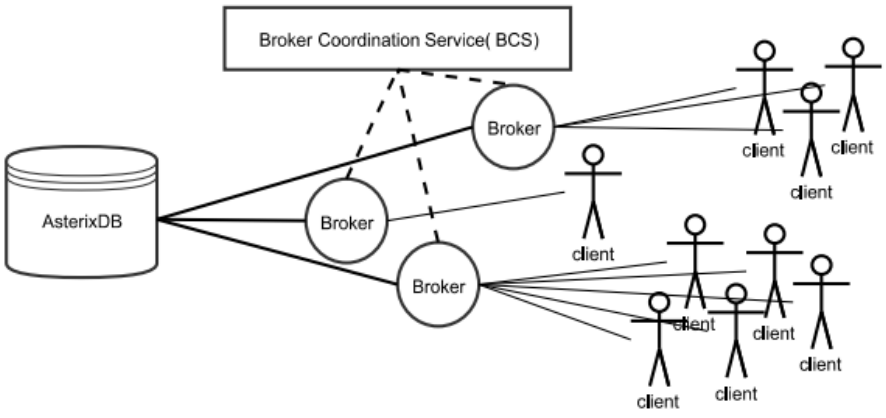

Fig.1. Architecture of Load balancing Project in BAD System

Based on the architecture of the BAD system, we design the architecture of the load balance project, as shown by Fig.1. There are three important components in the original BAD system, namely the Asterix database, the broker network, and the clients. In order to achieve the goal of load balance in the broker network, we implement a Broker Coordination Service (BCS) to facilitate the load balance

procedure. The main responsibility of the BCS includes the following three aspects. First, the BCS should communicate with brokers to acquire their latest load information. Second, the BCS should communicate with the client to tell him the least loaded broker at that time with which he should connect. Third, the BCS should dynamically make the migration decision on which client in which broker should be migrated to which broker.

3.2 Tornado Framework

The BAD system is built upon Tornado[11], a Python web framework and asynchronous networking library. Tornado can be roughly divided into four major components: a web framework, client- and server-side implementations of HTTP, an asynchronous networking library, and a coroutine library. Real-time web features require a long-lived mostly-idle connection per user. In a traditional synchronous web server, one thread will be allocated to one user, which can be quite expensive. In order to minimize the cost of concurrent connections, Tornado uses a single-threaded event loop, meaning that all application code should aim to be asynchronous and non-blocking because only one operation can be active at a time. Coroutines are the recommended way to write asynchronous code in Tornado, and Python yield keyword is used to suspend and resume execution instead of a chain of callbacks. In a Tornado web application, client and server communicate through HTTP requests. The server consists of one or more RequestHandler subclasses, and an Application object which routes incoming requests to handlers based on a routing table maintained. Every time a user request comes, the request will be dispatched to and then served by its according handler.

3.3 Transaction Flow

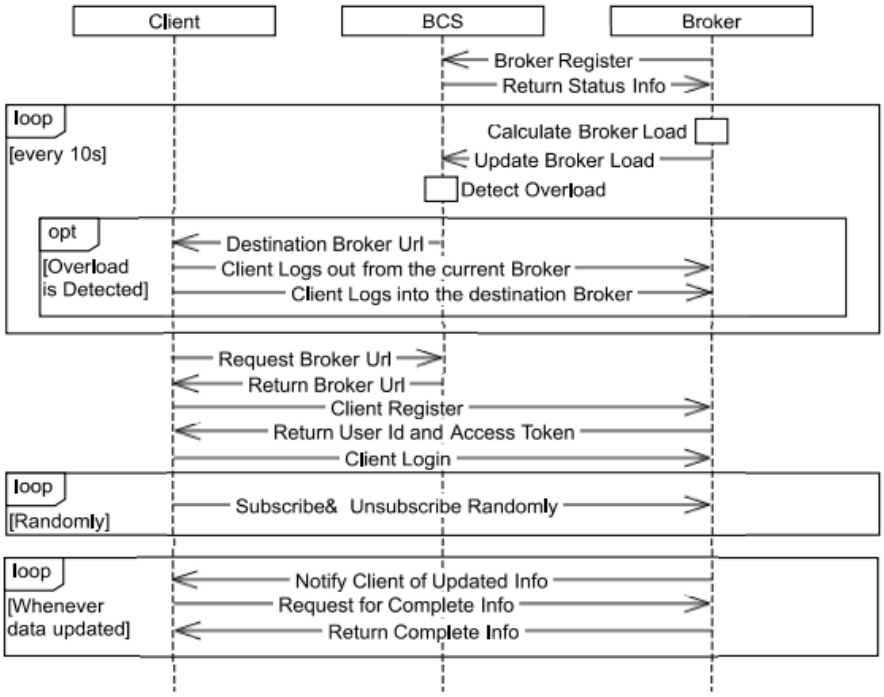

Fig.2. Sequence Diagram of Load Balancing Project in BAD System

In this section, the interactive behavior among the four components mentioned above is introduced. As the Asterix database only interact with the brokers and do not directly interact with the BCS and clients, we only illustrate the interactive behavior among the clients, the brokers, and the BCS.

The sequence diagram among the clients, the brokers, and the BCS is shown in Fig.2. The Asterix database and the BCS should be started first, and then, brokers should be started. When a broker starts running, it first registers itself to the BCS, and then receives the status information of the registration. Periodically, the broker estimates its current load and sends it to the BCS. The BCS receives the load information, updates the overall load information, and then starts overload detection. If an overload is detected, the BCS will make a migration decision, and then sends a migration signal along with the

destination broker URL to the selected client. Afterward, the client will start the migration procedure, including logging out from the current broker and then logging into the destination broker.

Besides, new clients will be started. Whenever a client is started, he will first contact the BCS to ask for the broker URL. After he receives the broker URL, he will register with that broker and then log into it with the returned information, including the user id and access token. The client can un/subscribe random channels with random parameters. The client will be notified whenever there is an update in the information that the client is interested in, and after the client receives the notification, he will send a message to the broker requesting for the complete information.

3.4 Load Estimation

A crucial problem in a load balancing system is how to estimate load accurately and efficiently. In the project, we mainly use two dimensions to describe and estimate the broker load, namely the number of the subscriptions and the volume of data packages.

Assume that n clients $\mathrm{C} = \{c_1, c_2, \dots, c_n\}$ subscribe broker B with their subscriptions $\mathrm{Sub} = \{sub_1, sub_2, \dots, sub_n\}$, where sub_i is the subscription number of client c_i. During a given time interval t, m data packages $\mathrm{DP} = \{dp_1, dp_2, \dots, dp_m\}$ with their volumes $\mathrm{V} = \{v_1, v_2, \dots, v_m\}$ are sent from broker B to n clients C, where v_i is the volume of the data package dp_i. In this condition, the load of the broker B is represented as a two-dimensional vector.

$$Load = [S, D], \text{where } S = \sum_{i=1}^{n} sub_i \text{ and } D = \sum_{i=1}^{m} v_i$$

In the formula given above, the first dimension, S, is the total number of subscriptions of the broker, which describes the potential broker load. Intuitively, the broker with more subscriptions are potentially bear more load. And the second dimension, D, is the total volume of the data packages send in the past t seconds, which describes the actual load of the broker B in the past t seconds. The goal of collecting broker load information includes two aspects, the first one is to estimate the current broker load and arrange the client distribution, and the second one is to evaluate the performance of the load balance strategies.

4 Implementation

Our system consists of three parts: client, broker, and broker coordination service (BCS). We will discuss in detail the communication pattern between them, how the clients get notifications and migration information from brokers and BCS, respectively. The implementation of the load balancing method will also be discussed in this section.

4.1 Client

A few modifications have been made to the original Client code. The first two are that the code on the client side is wrapped into a Client class and the code for application registration, client registration and client login has been extracted and slightly modified to form their own individual functions. And the other modification is the addition of three functions, namely the userRequst function, the userMigration function and the subUpdate function. The userRequest function is used for the communication between the client and the BCS, the client asks the BCS for the initial broker URL. The userMigration function is used for client migration when he needs to. And the subUpdate function is to simulate a client's un/subscription to a channel randomly. In the subUpdate function, if the client has not subscribed to any channel, then a random channel and a random parameter belonging to that channel will be picked for user subscription. And if the client has subscribed to certain channels and parameters, then an operation will be selected randomly from the subscription or un-subscription. If the subscription is chosen, then the client will subscribe to a new channel and parameter picked randomly; if the un-subscription is chosen, then the client will unsubscribe a randomly picked channel and parameter he has subscribed to.

4.2 Broker

A few modifications have been made on the original code. One is the establishment of the websocket between the broker and the BCS. Another one is the brokerLoadUpdater function which is responsible for updating load information to the BCS, including the total subscription number, the volume of the received data packages, and the information of load contributed by its subscribers. We sum the number of subscriptions of its clients to get the total number of subscriptions, and we accumulate the volume of each data package that the broker sends to acquire the volume of the data package. The volume of data package will be reset to zero after each load update so that data package sent in a certain time interval will be captured only.

4.3 Broker Coordination Service

A BCS class is implemented, and three main functions will be introduced here. The first one is the pickRandomBroker function which picks one broker randomly. The second function brokerSelection implements the static load balance algorithm, the function picks the least loaded broker at that time. Note that, the brokerSelection uses the number of subscriptions to calculate the load. The third function is the implementation of the dynamic load balance algorithm. The BCS keeps a dictionary, named clientInfo, to store the client information including the dataverseName and load. Another dictionary, named brokerLoads, records the users and total load of each broker. They will be updated in the loadUpdate function each time broker update its load information to BCS. During the runtime, the overloadDetection function is called in the loadUpdate to see if there is any overloaded node. A broker is overloaded when its load is larger than $0.2 \times \text{average load}$. All the overloaded brokers will be selected, and the same number of brokers with minimal load will also be selected. For an overloaded broker with its load Maxload, we will iterate users in the overloaded brokers and the brokers of minimum load, of which the load is remarked as Minload, to select the user that needed to be migrated and the its destination broker. A user is suitable for migration when it meets the following formula:

$$\text{client load} < 0.75 \times \text{dif}, \text{where dif} = \text{Maxload} - \text{Minload}.$$

Whenever a suitable client is found, the BCS will notify the user with the destination broker URL for migration.

5 Experiment

In order to evaluate the performance of our load balancing project in the BAD system, we design and conduct a series of experiments. The experiment section consists of two parts: the experiment design part where the experimental context is introduced, and the results and comparison part where the results of the experiments are comprehensively presented and analyzed.

5.1 Experiment Design

In order to concoct an experimental context of higher realism, we simulate random data updates and random un/subscription from clients. Before the experiments, we use the Tweetpy API to crawl some data from Twitter and build a collection of data records for the data simulation. Three controlled experiments are conducted under the same context, except the changes of the load balancing algorithm. In the experiments, the Asterix database, the BCS, the brokers and the clients are all run locally. We run one Asterix database, one BCS server on the port 5000 and three brokers on the port 9113, 9114, and 9115 respectively. We run eleven clients every 12s, and each client will randomly un/subscribe a randomly selected channel with one parameter every 30s. For the data simulation, five channels are created each of which has five parameters, and one data record will be selected randomly from the collection and inserted into the AsterixDB every 1s. And the broker will update its load information to the BCS every 10s.

The first experiment (Exp1) is designed as the controlled experiment. In Exp1, when a new client initially requests for a broker URL from the BCS, the BCS returns the URL of a randomly selected broker, and there is no migration during the runtime. The second experiment (Exp2) is designed to reveal the influence of the static load balancing strategy. In Exp2, when a new client initially requests for a broker URL, the BCS return the URL of the currently least loaded broker, and there is no migration during runtime as well. The third experiment (Exp3) is designed to reveal the influence of the dynamic load balancing strategy. In Exp3, when a new client initially requests for broker URL, the BCS return

the URL of the currently least loaded broker, and the migration for a client can happen during the runtime.

In order to compare three experiments in an unbiased way, each experiment is run 10 times, and each time last about 13 mins. Thus, for each series of experiments, roughly 80 records of broker load information can be collected.

5.2 Results and Comparison

After conducting three series of experiments, 2676 records are collected, each of which contains the load information of all three brokers. Note that, when evaluating the performance of the load balancing strategy, we use the volume of data package, the value of D, as described in Section 3.4.

For each record, the standard deviation of the three broker load values is calculated to represent the load balance performance at that time. In one series of experiments, the mean value of the standard deviations is calculated to represent the load balance performance of the experiments. Hence, 15 mean value of standard deviation of broker loads are calculated to reveal the performance of the three different load balancing strategies. And the results are shown in the Table.1 and the Fig.3.

In general, the experiment results are ideal and lives up to our expectation. First, the 10 mean values of standard deviation in the Exp1 is overall much higher than that of Exp2 and Exp3, which shows that in most cases the loads of pub/sub system without load balance strategies are prone to be unevenly distributed. Besides, the steep line of Exp1 in the line chart shows the unreliability of the systems without load balancing strategy to be implemented. In some cases, it can achieve a fairly good balance, which is around the average value of the Exp2, but it hardly equals to the average value of the Exp3. Second, the 10 values of Exp2 are much less than the value of Exp1, which shows that in the experiment environment, the static load balancing strategy is useful and efficient. Besides, the line of Exp2 in the line chart is flatter than that of Exp1, which shows that the system with static load balancing algorithm achieve a better and steadier load balance performance. Also, in some cases, the average standard deviation is large, which is caused by the subsequent subscribe and unsubscribe transactions send by the clients. At last, the 10 average standard deviation values in the Exp3 is the flattest and the lowest one. The results reveal the reliability and the efficiency of the dynamic load balancing algorithm. In our experimental results, the difference between the results of Exp2 and Exp3 are not that obvious, which may due to the experiment settings we have. More experiments of higher realism of the real-world pub/sub systems may better illustrate the power of dynamic load balancing algorithms.

Table.1. Experiment Results

	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	Ave
Exp1	1845	1408	1460	733	686	1508	2120	1135	1271	1568	1373
Exp2	520	696	694	523	1219	1196	557	928	741	742	784
Exp3	406	524	530	415	345	377	639	753	818	722	553

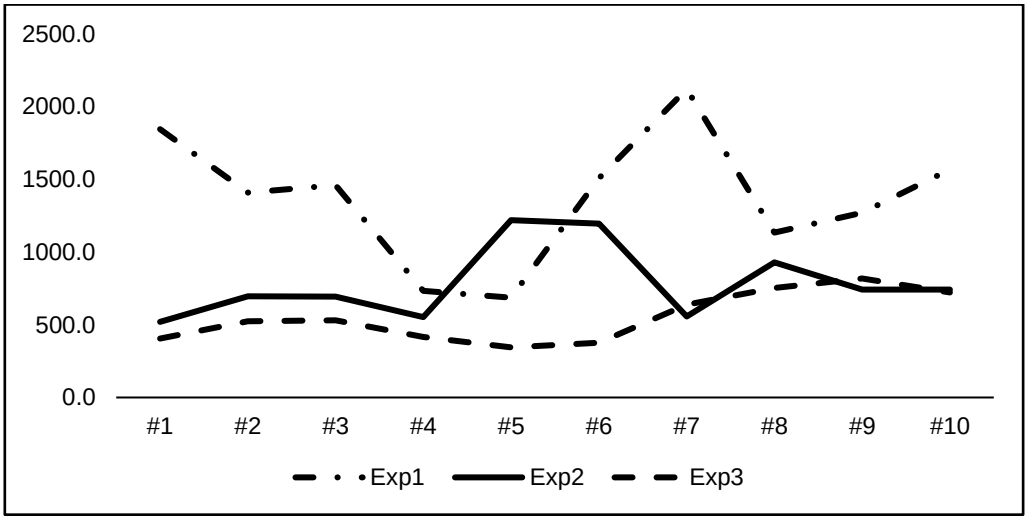

Fig.3. the Line Chart of the Experiment Results

6 Conclusion and Future Work

In the paper, we implement the load balance in the BAD system. Based on the architecture of the BAD system, a broker coordination service is implemented to realize the load balance algorithm. Two load balance strategies are implemented in our project, one is the static load balance algorithm which initially assigns a client to the currently least loaded broker, and another one is the dynamic load balance algorithm which dynamically migrates a client from an overloaded broker to a least loaded broker. Three controlled experiments are conducted to evaluate the performance of the two load balancing algorithms. As shown by the experimental results, the static load balance outperforms the random selection, and the dynamic load balance outperforms the static load balance algorithm. In a nutshell, the efficiency of our dynamic load balance algorithm is strongly proved.

Limitations exist in our project, and more work waits to be done in the future.

- Migration strategy can be improved. The current migration strategy is coarse and limited, and many more details need to be considered. For example, when a client migrates from broker A to broker B, the load that the client contributes to broker A does not necessarily equal to that contributes to broker B, because the load can vary due to the random data record updates in the AsterixDB. Besides, we also need to consider how to prevent load balance vibration.
- Experiments can be more objective. Due to the randomization characteristic of our experiments, the more experiments are performed, the more valid and objective the experimental results can be. Ten times execution of each experiment may not be adequate, more experiments and data are needed to increase the generality and precision of the experimental results.
- The BAD system can be improved. For example, the interaction between the broker network and the AsterixDB can be implemented in the pub/sub way. The brokers can subscribe certain channels of the Asterix DB and receive notifier whenever their interested data is updated.

Reference

[1] P. Samal and P. Mishra, "Analysis of variants in Round Robin Algorithms for load balancing in Cloud Computing," vol. 4, p. 4, 2013.

[2] F. Alam, V. Thayananthan, and I. Katib, "Analysis of round-robin load-balancing algorithm with adaptive and predictive approaches," in *2016 UKACC 11th International Conference on Control (CONTROL)*, 2016, pp. 1–7.

[3] M. Bramson, Y. Lu, and B. Prabhakar, "Randomized Load Balancing with General Service Time Distributions," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, 2010, pp. 275–286.

[4] S. F. El-Zoghdy and S. Ghoniemy, "A Survey of Load Balancing In High-Performance Distributed Computing Systems," vol. 1, p. 11, 2014.

[5] D. Grosu and A. T. Chronopoulos, "Noncooperative load balancing in distributed systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 9, pp. 1022–1034, Sep. 2005.

[6] A. Talukder, S. F. Abedin, M. S. Munir, and C. S. Hong, "Dual threshold load balancing in SDN environment using process migration," in *2018 International Conference on Information Networking (ICOIN)*, 2018, pp. 791–796.

[7] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Trans. Softw. Eng.*, vol. 14, no. 2, pp. 141–154, Feb. 1988.

[8] S. Alsubaiee *et al.*, "AsterixDB: A Scalable, Open Source BDMS," *ArXiv14070454 Cs*, Jul. 2014.

[9] M. J. Carey, S. Jacobs, and V. J. Tsotras, "Breaking BAD: a data serving vision for big active data," 2016, pp. 181–186.

[10] M. Qader, "A BAD Demonstration: Towards Big Active Data," 2017.

[11] "Tornado Web Server — Tornado 5.0.2 documentation." [Online]. Available: http://www.tornadoweb.org/en/stable/. [Accessed: 12-Jun-2018].