Hang Thi Thu Nguyen
Peeyush Gupta
Group No. 22

Load Balancing In Big Active Data (BAD)

1 Introduction To BAD

In this project, we aim to develop load balancing strategies for the Broker Network in the context of Big Active Data project [10]. The goal of BAD project is to build a system that can accumulate and monitor petabytes of data of potential interest to millions of end users in real time. BAD platform captures and analyzes data coming from outside data sources (Data Publishers) to produce and deliver interesting information to interested users in millisecond. The system has components providing two broad areas of functionality: BAD Data Cluster handles big data acquiring, managing and monitoring; BAD Broker Network handles notification management and distribution.

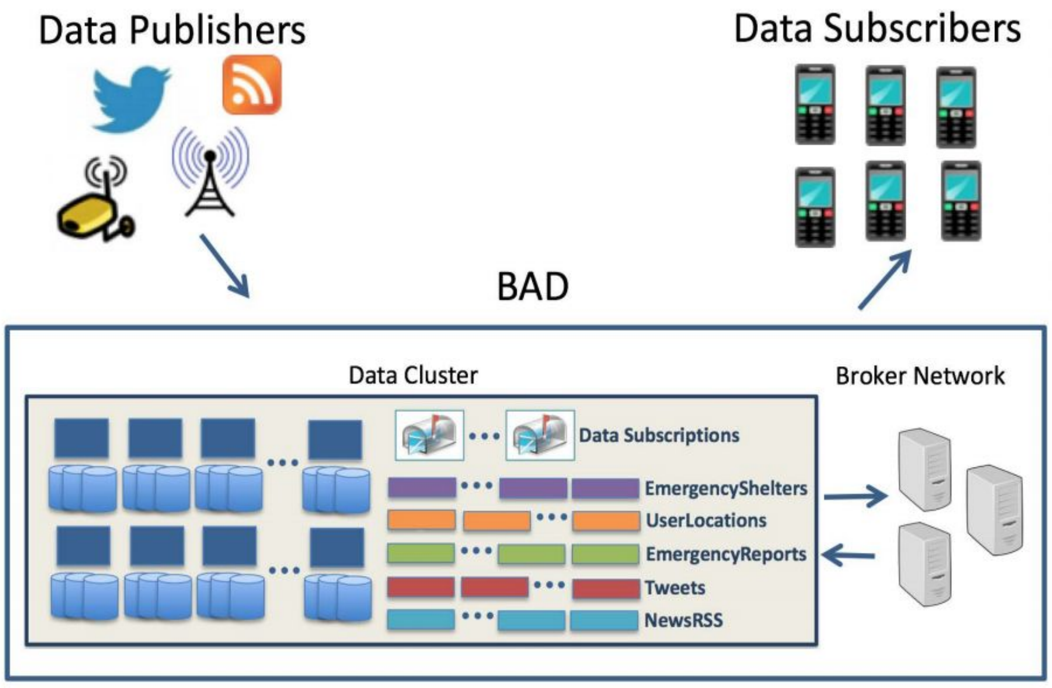

Figure 1: Big Active Data System Architecture

The BAD system has 3 classes of users:

- **Data Publishers:** provide data in the form of streams of incoming records into the Data Cluster via data feeds.
- **Application Managers:** create and manage parameterized channels that can be subscribed to for applications based on the expected user interest.
- **Data Subscribers:** subscribe to one or several channels through subscriptions to specific channels with parameter values of interest.

Each subscription specified by an user is registered in the BAD Data Cluster with unique subscription id per channel via Broker Network. Data Cluster issues subscription ids and keeps track of the broker responsible for receiving or forwarding notifications for each subscription. Result creation occurs in Data Cluster and is separated from result delivery that is the responsibility of Brokers. Subscribers thus can be allocated to different brokers when they move around.

In order to manage subscriptions and published results, BAD creates application datasets, Brokers dataset, Channels dataset. Along with the datasets mentioned before, the BAD system create two other datasets for each new created channel namely: Channel_Subscriptions and Channel_Results. Channel_Subscriptions schema includes subscriptionid, parameter value and the hosting broker for each subscription to the channel. Channel_Results schema contains subscriptionid, delivery Time and individual result for each subscription to the channel. The data stored in these datasets are then later can be fetched by brokers to deliver notifications to correct set of subscribers.

2 Load Balancing in BAD

In the current broker network implementation for BAD each user (data consumer/subscriber) needs to know the list of available brokers and can arbitrary choose the broker node to contact when they want to join the BAD system for the first time to subscribe for subscriptions. This may lead to one broker being connected to lots of clients with lots of subscriptions while other brokers may not get any connections at all, causing the broker network to be unbalanced. Also a broker can suddenly become overwhelmed when it has to serve clients which are having lots of subscriptions or are having subscriptions with lots of related publications.

As part of the project we have tried to solve both the above mentioned problems by developing both static and dynamic load balancing techniques.

3 Related Work

The distributed broker network in BAD system is based on the Pub/Sub model. Load balancing in pub/sub systems like TERA [2], PolderCast [3], SCRIBE [4], has been widely studied since the early days but the techniques are specific to the pub/sub architecture and the application type itself. There are various sub problems when designing load balancing framework such as: estimating the load on brokers, detecting overloading situations, how and what to offload etc. Different research studies have different ways to calculate load due to the inherent architecture of their pub/sub models and the specific application types. In order to evenly distribute the load over the brokers, the overlay network may be divided into different zones, partitions, or grouped into clusters such that each broker mainly responsible for the load causing by its own zone or cluster. For example, in Meghdoot [6] P2P based pub/sub architecture, the event space is partitioned among peers, each peer owns one zone. The load is combined of storage load and event propagation load in which the subscription storage load can be reduced by splitting the zone and the event propagation load can be decreased by creating an alternative path. The load balancing framework in PADRES system [1] uses event routing architecture to create a distributed load exchange protocol called PADRES Information Exchange. The other major part of the framework is the detector and mediator. All these modules together in PADRES are work together to provide dynamic load balancing in a distributed content based pub/sub system environment [8]. In another cluster-based Publish/Subscribe system [9], the broker network is organized into clusters where each broker knows all brokers in its own cluster and at least one broker in every other clusters. The inter-cluster connections forms multiple rings to support publication propagation through all the clusters and reach all clients subscribing to different clusters. In this work, they consider three main load components: ring publish load - the load to propagate publication over the ring, cluster publish load - the load to forward publication to subscribed brokers in the same cluster, and cluster subscription load - the load to send results to all subscribed clients. An offload process for different load components is invoked in any broker when the incoming message rate exceeds the threshold processing capacity of the broker.

4 Our Approach

The distributed brokers in BAD system located at different geographical areas may suffer from uneven load distribution due to different population densities, interests, and usage patterns of end users. There are different situations that may cause some brokers get overloaded. For example, when an emergency happens at some area, number of subscriptions in that location may suddenly increase as users want to receive info about the shelters. The broker in that hotspot areas is easily get overwhelmed due to unexpectedly high processing load for result retrieval and delivery. In BAD design, the results from all channels for all subscriptions are stored in the Asterix backend. Client can contact any broker node to request for the subscription results. When a new result produced by any channel comes, a notification will be sent to subscribed brokers and forwarded to subscribed clients. Different channels may have different levels of popularity. A channel may produce higher result frequency or larger volume of result content as it contain more information. This skew in the load of different channels may adds up at a broker when many users try to contact the same broker for subscription from that heavy channel.

In BAD system, a broker discovery server is built to assign broker to client when client newly join the system for subscription and to keep track of what clients that any broker is currently serving. Based on the inherent characteristic of the system, we have implemented two levels of load balancing for BAD all with the participation of the broker discovery server. First level is the static load balancing that tries to balance the load on brokers by picking up broker using one of the static load balancing techniques, whenever a client request for a broker to connect to. The next level is the dynamic load balancing where the system figures out an overloaded broker and a broker which can take the load and distributes the load by offloading clients from overloaded broker to the underloaded broker.

4.1 Broker Discovery Service

The discovery server plays the role of central coordinator for management of the broker system in general and for load balancing in particular. When a new broker joins the system, it first register with the discovery server. Every broker from then on will calculate and update its load information to the discovery after a configured interval of time. In our implementation, the discovery server has in memory hash tables storing cached data that about the load on every broker in the system. The discovery server has various modules like overload detector, load collector, data cache and load balancers as shown in figure 2, which provide two main services corresponding to two load balancing techniques: static load balancing performed by Static Load Balancer service and dynamic load balancing performed by Dynamic Load Balancing service. Discovery server calculates load score for each broker based on its load info, which helps in providing weight based scheduling and also used to detect overloaded brokers, is a weighted sum of the performance metrics of the broker load such as CPU and memory usage and the number of subscriptions, the number of current user connections that the broker is now servicing and also the length of the message queue in the broker's cached data.

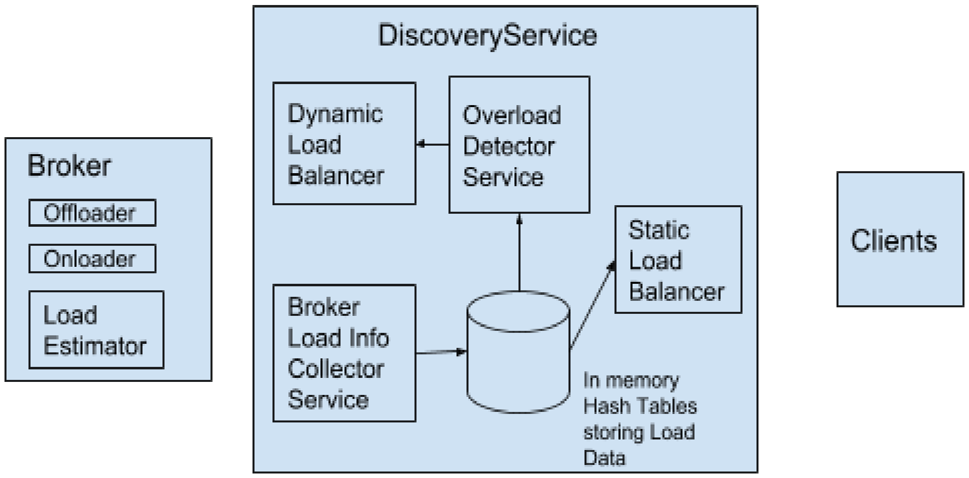

Figure 2: Broker Discovery Service (Load Balancer)

4.2 Static Load Balancing

For static load balancing as shown in figure 3.1, when a client newly join the system for the first time to do the subscription or when an existing client come online to retrieve subscription results, the client will first contact the discovery server to be assigned with a broker for the request service. The discovery server does a simple job of picking up the lightest loaded broker from the cached data and send its information to the client. This approach of allowing any client connect to any broker is doable due to the fact that any broker can retrieve the user subscription and channel subscription metadata from the Asterix backend to fetch subscription results to any client request.

4.3 Dynamic Load Balancing

In broker discovery server, we implement an always running independent thread that keep checking the broker load info in the in memory cached data. It sorts this hash table to find out all the overloaded and underloaded nodes and trigger the offloading process . The overloaded brokers redirect some clients to other underloaded brokers and remove those clients' subscriptions to release load. Figure 3.2 shows the messages being passed through different components of the system once overload is detected at one of the broker by the discover service.

6 Evaluation And Conclusion

Our test setup involved 4 Brokers, 10 Clients, 100 Subscriptions and 1000 Publications. Through the discovery server clients were able to connect to brokers based on one of the static load balancing strategy(chosen by the client) among least load score, round robin, random, nearest. In addition to overloaded brokers were being detected by the discovery service and eventually overloaded brokers were able to offload clients chosen through offload algorithm and making the chosen client reconnect to other underloaded broker.

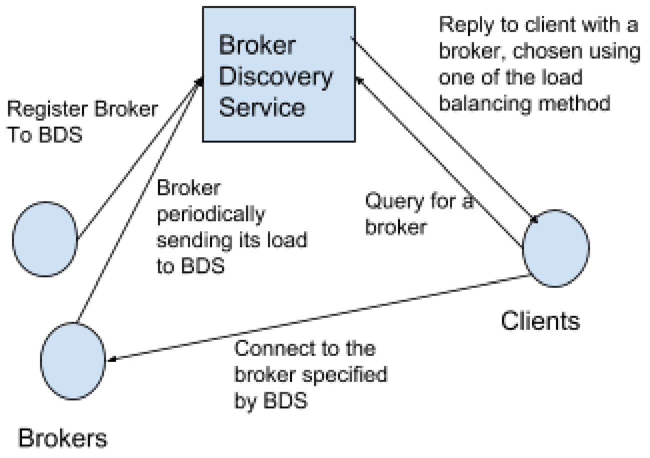
Broker Discovery Service
Register Broker To BDS
Broker periodically sending its load to BDS
Reply to client with a broker, chosen using one of the load balancing method
Query for a broker
Clients
Connect to the broker specified by BDS
Brokers

Figure 3.1: Static Load Balancing

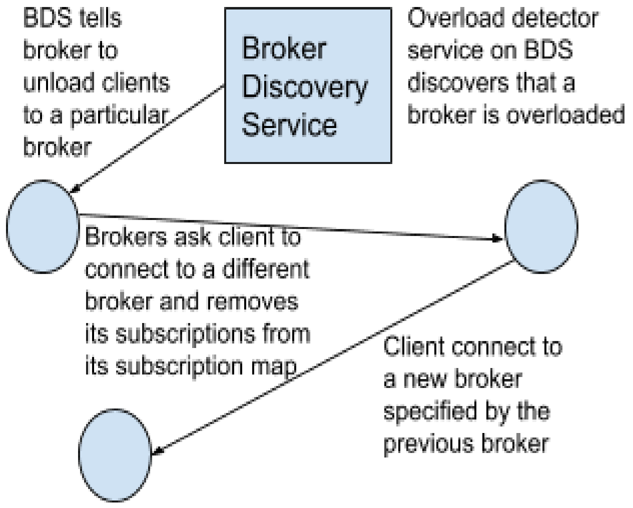
BDS tells broker to unload clients to a particular broker
Broker Discovery Service
Overload detector service on BDS discovers that a broker is overloaded
Brokers ask client to connect to a different broker and removes its subscriptions from its subscription map
Client connect to a new broker specified by the previous broker

Figure 3.2 Dynamic Load Balancing

Our solution involves a centralized coordinator based load balancing architecture built on top of existing BAD broker code. This single point of failure can be removed if brokers pass load information to each other. For future work, a better load metrics accounting for factors like network congestion, latency, overhead can be investigated. Work can be done to find better algorithms to pick which clients or subscriptions should be moved.

7 References

1. Jacobsen, H.-A., Cheung, A. K. Y., Li, G., Maniymaran, B., Muthusamy, V. & Kazemzadeh, R. S. (2010). **The PADRES Publish/Subscribe System**. *In A. Hinze & A. P. Buchmann (ed.), Principles and Applications of Distributed Event-Based Systems (pp. 164-205) . IGI Global . ISBN: 9781605666983.*

2. Ji, S., Jacobsen, H., Wei, J., & Ye, C. (2015). MERC: **Match at Edge and Route intra-Cluster for Content-based Publish/Subscribe Systems**. *Middleware*.

3. Baldoni, R., Beraldi, R., Quema, V., Querzoni, L., & Tucci-Piergiovanni, S. (2007). TERA: T**opic-based event routing for peer-to-peer architectures**. *In ACM International Conference Proceeding Series (Vol. 233, pp. 2-13). DOI: 10.1145/1266894.1266898*

4. Setty V., van Steen M., Vitenberg R., Voulgaris S. (2012) **PolderCast: Fast, Robust, and Scalable Architecture for P2P Topic-Based Pub/Sub**. *In: Narasimhan P., Triantafillou P. (eds) Middleware 2012. Middleware 2012. Lecture Notes in Computer Science, vol 7662. Springer, Berlin, Heidelberg*

5. **SCRIBE: A large-scale and decentralized application-level multicast infrastructure IEEE Journal on Selected Areas in communications (JSAC)**, *Vol. 20, No. 8. (October 2002) by M. Castro, P. Druschel, A. Kermarrec, A. Rowstron.*

6. Gupta A., Sahin O.D., Agrawal D., El Abbadi A. (2004) **Meghdoot: Content-Based Publish/Subscribe over P2P Networks**. *In: Jacobsen HA. (eds) Middleware 2004. Middleware 2004. Lecture Notes in Computer Science, vol 3231. Springer, Berlin, Heidelberg*

7. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp , Scott Shenker (2001) **A Scalable Content-Addressable Network**. *In ACM SIGCOMM 2001*

8. Alex King Yeung Cheung and Hans-Arno Jacobsen (2006) **Dynamic Load Balancing in Distributed Content-based Publish/Subscribe**. *Middleware 2006*

9. Hojjat Jafarpour, Sharad Mehrotra and Nalini Venkatasubramanian **Dynamic Load Balancing for Cluster-based Publish/Subscribe System**

10. Carey, M. J., Jacobs, S. & Tsotras, V. J. (2016). **Breaking BAD: a data serving vision for big active data**. *In A. Gal, M. Weidlich, V. Kalogeraki & N. Venkasubramanian (eds.), DEBS (p./pp. 181-186), : ACM. ISBN: 978-1-4503-4021-2*