Distributed Broker Network System in Cloud

Sharad Manasali, Pratikshya Panigrahi, Lakshmipriyadarshini Velmurugan

1. Introduction

In the last decade, cloud computing has been a major trend in the industry. With cloud computing, users have access to applications all over the world through a web browser. It is, in fact, a virtualized computer system that contains all software and applications needed for businesses. It provides a different model of operation in which the service providers are liable for hardware resources, upgrade, maintenance, failover, backups, security, etc. There are no extra investments on the server, Instead, users can only pay for the services on demand. Thus using cloud computing services saves a lot of time, money and make their operations more effective [1]. Our project's main aim is to containerize the Distributed Brokers Nodes in a highly scaled, PubSub based Emergency notification system and deploy them on the cloud. Publish-Subscribe system (PubSub), is a type of database system that serves a group of subscribers interested in various events with notifications as publications for events occur. This PubSub based Emergency notification system is based on Big Active Data that uses BAD AsterixDB as its database. The broker Nodes are connected with the users providing them notifications about various emergencies based on their subscription. These Broker Nodes are managed as Kubernetes cluster to ensure availability and load balancing.

2. Related Work

Implementing distributed broker nodes on the cloud has been an active research domain and the following section of the report highlights some of the literary work based on cloud brokers and how they tie together with our project. **Stratos**[12] is a Cloud Broker Service that deploys applications and services on demand from various providers based on requirements from the users. Stratos allows the application deployer to specify important decision-making algorithms so that when a request for resource acquisition comes in, the cloud broker can take appropriate decisions. It is also responsible for connecting the resource providers with users for the acquirement and release of resources. Similarly, in our proposed work, the Broker Controller Service is responsible for assigning the brokers to the users based on the requirements such as the number of users, location and also takes care of the load balancing among the brokers. Based on the users' traffic to and from the database, the broker controller assigns the load to other available brokers.

In this era of Big Data and Cloud Computing, many applications have high scalability and availability requirements that cannot be handled by traditional centralized data management systems. To meet such requirements in a cost-effective manner, data usually has to be partitioned across commodity hardware computing clusters [2, 3, 4]. However, as clusters grow larger hardware failures become unavoidable. To tolerate hardware failures, data-replication strategies are used. For proper implementation of data-replication, we have to keep in mind what consistency constraints should be guaranteed, what is an acceptable

recovery time in case of failures, where to place different replicas, how to load balance workloads across replicas, and how to reach consensus between replicas in case of failures. Data-replication protocol is a complex task and requires a careful design as well as implementation. Over the years, many data replication strategies have been proposed [5, 6, 7]. In this thesis of **Data Replication and Fault Tolerance in AsterixDB** [8], they have described a new data replication protocol for AsterixDB that guarantees data consistency, has a low impact on throughput as well as on the computational resources required for replication and exploits the properties of Log-Structured Merge-trees (LSM-tree) [9] to achieve efficient data replication and controllable recovery time. AsterixDB is one such Big Data Management System (BDMS) that is designed to run on large clusters of commodity hardware. They have also explained how fault tolerance is implemented on top of the data replication protocol.

Similarly in **A BAD Thesis: The Vision, Creation, and Evaluation of a Big Active Data Platform** [11], the BAD platform is presented, that combines ideas and capabilities from both Big Data and Active Data (e.g., Publish/Subscribe, Streaming Engines). It supports complex subscriptions that consider not only newly arrived items but also their relationships to past, stored data. Further, it can provide actionable notifications by enriching the subscription results with other useful data. The platform extends an existing open-source Big Data Management System, Apache AsterixDB. A Big Active Data (BAD) system should continuously and reliably capture Big Data while enabling timely and automatic delivery of relevant information to a large pool of interested users, as well as supporting retrospective analyses of historical information. Because we are designing a PubSub based emergency notification system, where we have continuously keep a record of all subscribers and also all the publications about any emergencies that might happen at any time, we need a very robust and highly scalable system. The highly scalable, available, robust, fault-tolerant nature of BAD-AsterixDB (Big Active Data-AsterixDB), plus the data-replication protocol implemented on it makes it the best option for such applications.

Cloud computing has become a major trend in the industry, in recent years. Cloud computing can be thought of as a virtualized computer system that contains all software and applications needed for businesses. It provides a radically different model of operation in which the service providers are liable for hardware resources, upgrade, maintenance, failover, backups, security, etc. In **Container-based IoT Sensor Node on Raspberry Pi and the Kubernetes Cluster Framework** [10], a container-based virtualization technique has been adopted to implement an embedded IoT sensor node. Their main aim was to develop a clustered system of five Raspberry Pi (RPi) embedded boards by utilizing Docker containers and the Kubernetes cluster framework. **Kubernetes** [13] is a portable, extensible open-source platform for managing containerized workloads and services, that aids both declarative configuration and automation. Kubernetes provides a container-centric management environment and a unified API to deploy web applications, batch jobs, and databases. Containers are how apps are packaged and deployed in Kubernetes. The main reason for selecting RPi is to produce an efficient system since it has an ARM-based processor [10]. The sensor node operates by collecting temperature data and motion detected pictures from the camera sensors attached to each RPi board [10]. The pictures in the form of data are then replicated across a cluster and sent to the cloud platform, Apache Kafka [10]. Similarly, in our project, we are trying to containerize the broker nodes and BCS

and deploy them on the Kubernetes cluster. The applications built in the docker are wrapped with all the supporting dependencies into a docker container. The containers then keep running in an isolated manner on top of the parent operating systems' kernel. Those applications can run as independent services. Kubernetes manages and takes care of the runtime of the docker containers by exposing the deployments. Kubernetes ensures availability and load balancing in the cloud cluster.

3. Design

In this section, the main structure of the distributed broker network in Kubernetes is introduced, including the project architecture, the flow design, and the implementation framework.

3.1 System Architecture

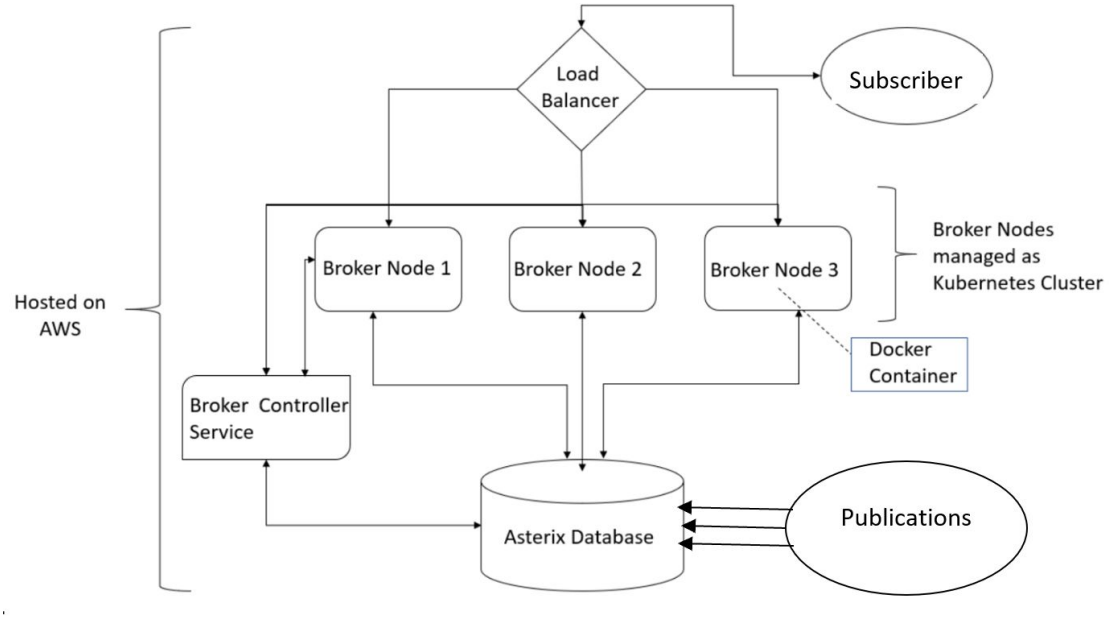


Fig 1. The architecture of Distributed Broker Network

The architecture of the Distributed Broker Network as shown in figure 1 comprises of four components, namely the brokers, the broker controller service, the clients, and the Asterix database.

The responsibilities of each of these components are delineated as follows:

a. Broker Controller Service: Acts as an initial point for broker registration and client registration. Responsible for migrating the clients to another broker node in case of a failure.
b. Broker: Registers itself with the BCS. Responsible for delivering the published information to the subscribed clients from the Asterix database.
c. Clients: Contacts the BCS to ask for a Broker URL, subscribes and unsubscribes to random channels.
d. Asterix Database: Stores all the information regarding subscriptions, and publications.

3.2 Design Flow

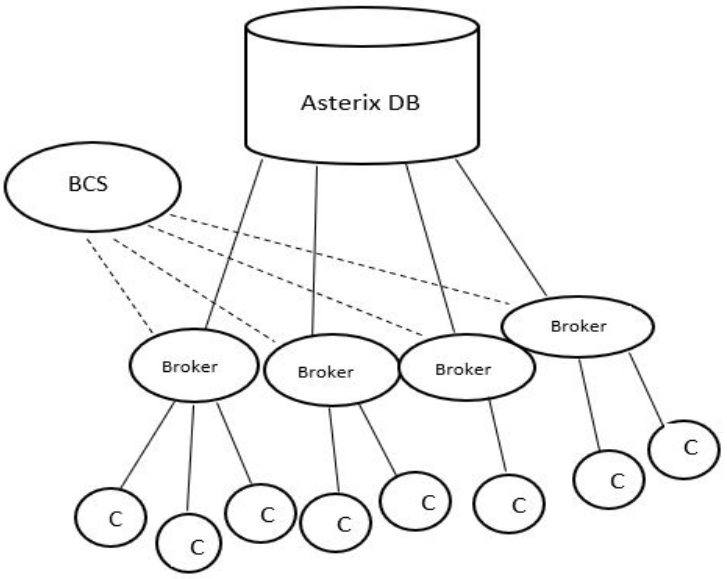


Fig 2. The Design of the Broker Network

The Asterix database and the BCS are started first. The Brokers are started next. The design of the distributed broker network is shown in Figure 2. When a broker is created, it first registers itself with the BCS and receives the status information of the registration. Periodically, the BCS makes sure that the brokers are still running and gets the load information from all the brokers. BCS updates the overall load and detects if load balancing has to be done. If so, starts a migration procedure and informs the respective clients of the change with the new destination Broker URL. The client starts the migration procedure by logging out of the current broker and logging into the new broker.

When a new client is started, it contacts the BCS and asks for the Broker URL. After receiving the Broker URL, the client registers with the Broker and (un)subscribes to random channels with random parameters. The client will be notified whenever there happens an update in the information the client is interested in. The client receives the notification and asks for more information from the Broker.

4. Implementation

The implementation of the Distributed Broker Network as shown in Figure 1 is implemented using four major systems, namely the Kubernetes Cluster, the Docker containers as Broker Nodes, AWS (Amazon Web Services) as a storage unit, and Asterix Database.

4.1 Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery [13]. In our project, Kubernetes is used to manage the Broker Nodes as a Kubernetes Cluster.

4.2 Docker Containers

Docker is a standalone software that can be installed on any computer to run containerized applications. Containerization is an approach of running applications on an operating system such that the application is isolated from the rest of the system. Docker enables to run, create and manage containers on a single operating system. In our project, the broker nodes are deployed as Docker containers.

Kubernetes allows us to automate container provisioning, networking, load balancing, and scaling across all these nodes from the command line or dashboard. The collection of nodes that are managed by a single Kubernetes instance is referred to as a Kubernetes cluster. This kind of deployment (Docker Containers on Kubernetes) offers the following advantages:

a. *Robust Infrastructure*: Offers high availability by bringing up nodes if some nodes crash or fail.
b. *Scalable Infrastructure:* In case the workload of a node increase, we could spawn additional containers or nodes and add them to the cluster [14].

4.3 Amazon Web Services (AWS)

Amazon hosts a large number of web services. In our project, we focus on the Amazon ECS (EC2 Container Services). Amazon EC2 (Elastic Compute Cloud) is a web service that provides secure, resizable compute capacity in the cloud. In our project, the Kubernetes clusters are hosted on AWS EKS(Elastic Kubernetes Service).

4.4 Implementation Flow

- A role is created for the services in AWS.
- An empty stack is created using cloud formation. The output of the stack are:
  VPC ID of the cluster, SubnetID's, Control plane security group.
- AWS - IAM (Identity and Access Management) authenticator to connect to the AWS console and *kubectl* which is the kubernetes cluster manager are installed.
- AWS - EKS (Elastic Kubernetes Service) cluster is provisioned with the details from the empty stack previously created.
- A stack for the worker nodes is created using cloud formation. For the purpose of the project, a single instance is enough to hold multiple docker containers. The steps are visualized in Figure 3.

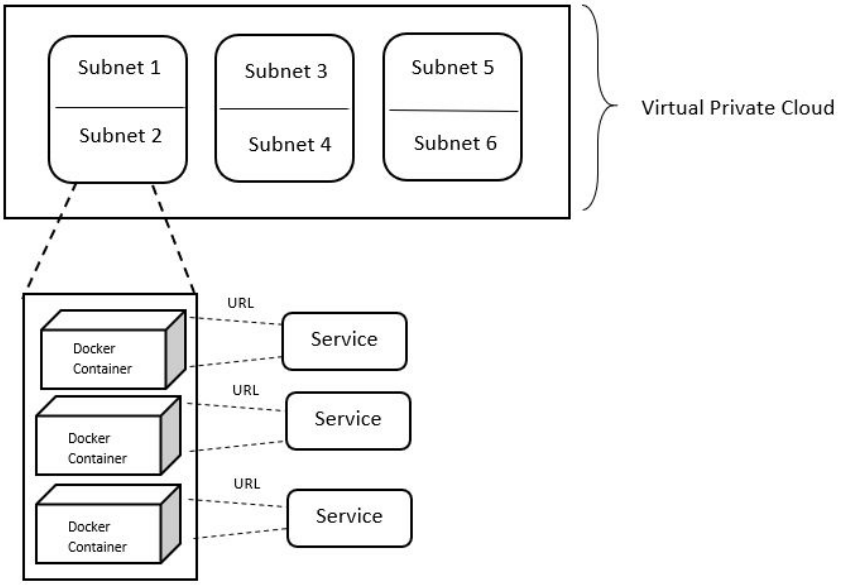


Fig 3. Exposing Docker Containers as Service through URLs

5. Experiment

This section explains the experimental setup for the implementation scenario described in the above section. Broker nodes as Docker containers are deployed as a replica set of three. In order for us to access the docker containers, the same must be extended as a service. The service is the load balancer, which will balance the load on the docker containers depending on the user access. The load balancer can be extended to various options of Route53 [15] in AWS such as latency, weighted, geographical routing, etc. With limitations on the AWS account access, this has not experimented.

5.1 Experiment Design

- Using one of the AWS - EC2 instances the docker images are built. We chose to utilize the Google Cloud Platform (Docker container registry) to house our docker images. The platform is authenticated by an SDK and the docker images are pushed as and when they are built.
- The deployments and services are executed as .yaml files which include the specifications such as replica set, docker image URL, the port on which the docker container should run, port exposed to the outside world, type of service, and node selector.

5.2 Experiment Results

The snippet of the deployments and services

It demonstrates the implementation of BCS as the Load Balancer and the port on which it is running along with the protocol used. The next line illustrates that the brokers are managed as a Kubernetes cluster with the IP 10.100.0.1: 443.

```
C:\Users\shara\Desktop\project>kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
brokerdeployment1   3         3         3            3           1h

C:\Users\shara\Desktop\project>kubectl get services
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP                                                                PORT(S)          AGE
brokerservice   LoadBalancer   10.100.159.25   a04761e3e8f0211e99c280257bc7e18a-1994244408.us-west-2.elb.amazonaws.com   5000:30537/TCP   33m
kubernetes      ClusterIP      10.100.0.1      <none>                                                                     443/TCP          140d
```

The snippet of the deployed Broker Nodes

It shows the status of the deployed broker nodes that are running as Docker Containers.

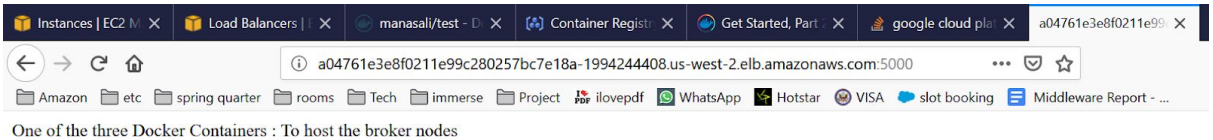
```
C:\Users\shara\Desktop\project>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
brokerdeployment1-5ddc79bdf7-q4lqq   1/1     Running   0          49m
brokerdeployment1-5ddc79bdf7-vkvh4   1/1     Running   0          49m
brokerdeployment1-5ddc79bdf7-wpl5g   1/1     Running   0          52m
```

The snippet of the running replica set of three docker containers

It shows details of each running replica set of the 3 docker containers with the names of the containers, their creation time, their status, and on which port are they running, etc.

```
[root@ip-192-168-83-12 ec2-user]# docker ps
CONTAINER ID        IMAGE                                                                COMMAND                  CREATED
     STATUS              PORTS               NAMES
17d11e581516        b9e1c43c93da                                                         "/bin/sh -c 'pytho..."   45 minutes ago
     Up 45 minutes                           k8s_brokerdeployment1_brokerdeployment1-5ddc79bdf7-q4lqq_default_f43fbb53-8f01-11e
9-9c28-0257bc7e18ac_0
0abb896bd12b        602401143452.dkr.ecr.us-west-2.amazonaws.com/eks/pause-amd64:3.1   "/pause"                 45 minutes ago
     Up 45 minutes                           k8s_POD_brokerdeployment1-5ddc79bdf7-q4lqq_default_f43fbb53-8f01-11e9-9c28-0257bc7
e18ac_27
75e8d6062674        b9e1c43c93da                                                         "/bin/sh -c 'pytho..."   About an hour a
go   Up About an hour                        k8s_brokerdeployment1_brokerdeployment1-5ddc79bdf7-vkvh4_default_f13940cc-8f01-11e
9-9c28-0257bc7e18ac_0
cd7c5802bc10        602401143452.dkr.ecr.us-west-2.amazonaws.com/eks/pause-amd64:3.1   "/pause"                 About an hour a
go   Up About an hour                        k8s_POD_brokerdeployment1-5ddc79bdf7-vkvh4_default_f13940cc-8f01-11e9-9c28-0257bc7
e18ac_0
118b607d8324        gcr.io/middleware-243721/images                                      "/bin/sh -c 'pytho..."   About an hour a
go   Up About an hour                        k8s_brokerdeployment1_brokerdeployment1-5ddc79bdf7-wpl5g_default_7d58e11b-8f01-11e
9-9c28-0257bc7e18ac_0
9b434d67bdb5        602401143452.dkr.ecr.us-west-2.amazonaws.com/eks/pause-amd64:3.1   "/pause"                 About an hour a
go   Up About an hour                        k8s_POD_brokerdeployment1-5ddc79bdf7-wpl5g_default_7d58e11b-8f01-11e9-9c28-0257bc7
e18ac_19
```

Accessing the load balancer will direct the traffic to one of the docker containers provisioned.



One of the three Docker Containers : To host the broker nodes

Fault-Tolerance

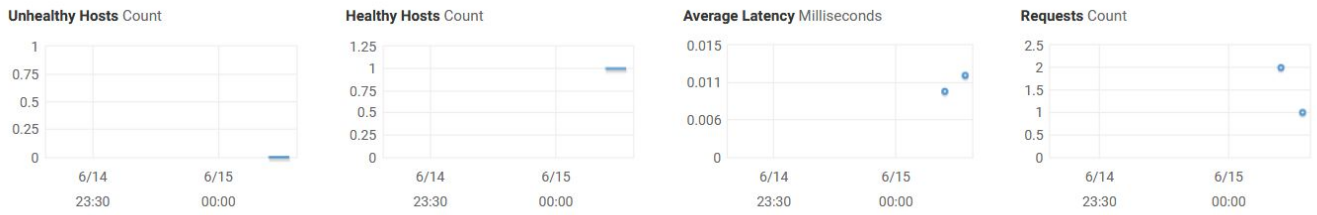
Killing one of the docker containers in a kubernetes cluster will automatically initiate another docker container with zero downtime supported from the elastic load balancer as well. The following snippet demonstrates this where the terminating node (52m) immediately leads to the spawning of the new broker deployment (15s).

```
C:\Users\shara\Desktop\project>kubectl delete pod brokerdeployment1-5ddc79bdf7-q4lqq
pod "brokerdeployment1-5ddc79bdf7-q4lqq" deleted

C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>
C:\Users\shara\Desktop\project>kubectl get pods
NAME                                READY   STATUS        RESTARTS   AGE
brokerdeployment1-5ddc79bdf7-q4lqq   1/1     Terminating   0          52m
brokerdeployment1-5ddc79bdf7-rcqqt   1/1     Running       0          15s
brokerdeployment1-5ddc79bdf7-vkvh4   1/1     Running       0          53m
brokerdeployment1-5ddc79bdf7-wpl5g   1/1     Running       0          56m
```

Load Balancer metrics:

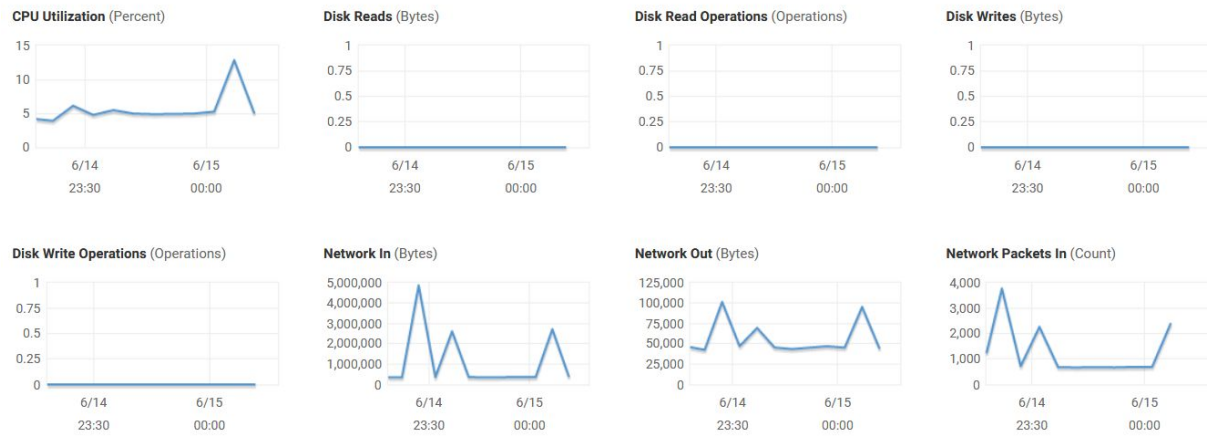
The following snippet shows the load balancer with the number of brokers and their states. This information is critical in the event of any failure since it will be required to spawn another broker.



a. *Unhealthy Hosts*: The number of unhealthy instances registered with the load balancer. An instance is considered unhealthy after it exceeds the unhealthy threshold configured for health checks.
b. *Healthy Hosts*: The number of healthy instances registered with the load balancer. (passing the timely health check)
c. *Average Latency*: The total time elapsed, in seconds, from the time the load balancer sent the request to a registered instance until the instance started to send the response headers.
d. *Requests*: The number of requests completed or connections made during the specified interval (1 or 5 minutes).

EC2 Instance metrics:

The figure shows a snippet of the performance metrics such as CPU utilization, disk operations, and network utilization of the chosen EC2 instance for deployment.



6. Conclusion and Future work

In this project, we explored how a distributed broker node can make use of cloud services to become more available and scalable. Our project focused on two parts: conversion of the broker nodes into docker containers and how they can be deployed and managed as Kubernetes clusters. This project allowed us to explore the various important metrics of distributed systems such as availability, scalability, and load-balancing. We consider the following as a possible future work:

a. Migrate Asterix Database from its traditional deployment into AWS.
b. Explore other functions such as failover handling, migration handling, and handoff mechanisms among brokers.
c. Implement a decentralized intelligent handling mechanism that could predict failure and take necessary migrations and failover handling.

References

[1] Dr. Dothang Truong, “How Cloud Computing Enhances Competitive Advantages: A Research Model for Small Businesses”, The Business Review, Cambridge, 2010.
[2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI’04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
[3] D. DeWitt and J. Gray. Parallel database systems: The future of high-performance database systems. Commun. ACM, 35(6):85–98, June 1992.
[4] V. Borkar, M. J. Carey, and C. Li. Inside ”Big Data Management”: Ogres, onions, or parfaits? In Proceedings of the 15th International Conference on Extending Database Technology, EDBT ’12, pages 3–14, New York, NY, USA, 2012. ACM.

[5] S. Goel and R. Buyya. Data replication strategies in wide area distributed systems, enterprise service computing: From concept to deployment. In Robin G. Qiu (ed), ISBN 1-599044181-2, Idea Group Inc, pages 211–241, 2006.
[6] H.-I. Hsiao and D. J. DeWitt. A performance study of three high availability data replication strategies. Distrib. Parallel Databases, 1(1):53–80, Jan. 1993.
[7] S. A. Moiz, S. P., V. G., and S. N. Pal. Article: “Database replication: A survey of open source and commercial tools”. *International Journal of Computer Applications*, 13(6):1–8, January 2011. Full text available.
[8] Al Hubail, M. M. (2016). Data Replication and Fault Tolerance in AsterixDB. *UC Irvine*. ProQuest ID: AlHubail_uci_0030M_13967. Merritt ID: ark:/13030/m57m4wrw. Retrieved from https://escholarship.org/uc/item/6pn1f8nj.
[9] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil. The log-structured merge-tree (LSMtree). Acta Informatica, 33(4):351–385.
[10] A. Javed, Container-based IoT Sensor Node on Raspberry Pi and the Kubernetes Cluster Framework, 2016.
[11] Jacobs, S. (2018). A BAD Thesis: The Vision, Creation, and Evaluation of a Big Active Data Platform. *UC Riverside*. ProQuest ID: Jacobs_ucr_0032D_13462. Merritt ID: ark:/13030/m55q9t2d. Retrieved from https://escholarship.org/uc/item/47g680h1.
[12] Przemyslaw Pawluk, Bradley Simmons, Michael Smit, Marin Litoiu, “Introducing STRATOS: A Cloud Broker Service”, *IEEE Fifth International Conference on Cloud Computing* (2012):891-898.
[13] “Kubernetes Homepage” [Online]. Available: https://kubernetes.io/ .[Accessed: 09-June-2019].
[14] “Kubernetes vs Docker: A Primer” [Online]. Available: https://containerjournal.com/2019/01/14/kubernetes-vs-docker-a-primer/ . [Accessed: 09-June-2019].
[15] “Amazon Route 53: Highly Available and Scalable Domain Name Server”. Available: https://aws.amazon.com/route53/ . Accessed: 14-June-2019