Elastic Akka Cluster with Websockets

Utkarsh Upadhyay, Aviral Jain

Abstract*: *In this paper we will discuss an automated cost-effective solution to induce elasticity in a system dealing with the persistence of websocket connection. We are going to make use of the concept of Akka Clustering to ensure there is no single point of failure making the system resilient. The system is deployed using automation to the AWS cloud where we will make use of Cloudwatch and Lambda functions to scale up or down the cluster as traditional methods to scaling based on memory and CPU utilization will not suffice. The elastic nature of the system will make it flexible to deal with variable loads in terms of connected clients and scale based on the number of connected clients. The use of AWS auto-scaling will also ensure the system is highly available.

***Keywords*: Websocket, Elasticity, Akka cluster, Cloud Automation.**

I. INTRODUCTION

Websockets are a communications protocol which provide a two-way communication channel with a remote host over a single TCP channel. The Websocket (ws:// or wss://) protocol is a great solution to get real-time data between a server and a client and of-fers much greater performance compared to Ajax polling and Comet solutions that are used to simulated two-way communication. Websockets can be used in multiplayer gaming, multimedia chats, social feeds, etc. The persistent nature of websockets makes the scaling of websocket servers challenging. There is a hardware limitation of each server in terms of the number of connections which can be handled. While running multiple servers can be a workaround there are situations where the number of servers running would be more than which are required making it an expensive solution.

Akka is a set of open-source libraries which can be used to design scalable, fault-tolerant applications which can span across processor cores and networks. Akka employs the concept of Actors. Actors are like Objects in Java which are containers for State, Behavior, a Mailbox, Child Actors, and a Supervisor Strategy. Actors can only communicate with each other by sending messages and this is the primary differ-ence between Actors and Objects, where objects communicate by the invocation of methods. When an Actor receives a message, it can send a message to a finite set of actors it is aware of, create a finite number of actors and change the behavior to be ap-plied when the next message is received. Actors form the fundamental processing of computation units called Actor Systems which can lie across various nodes forming a distributed system and Actors within these distributed systems can communicate with each other. Akka clustering enables building of high-performance distributed systems, where an application can span across multiple distributed nodes with no single point of failure.

In this paper, we will present a solution which induced the behavior of automated scale up and scale down (elasticity) in an Akka clustered application residing within the AWS cloud to which clients connect to by means of websocket connection. The elastic behavior will be created using concepts like auto-scaling, Cloudwatch alerts and Serverless Lambda functions.

II. SYSTEM DESIGN

A. CI/CD Pipeline

With the popularity of cloud hosted web services, the concept of Continuous Integration and Continuous Deployment or CI/CD has also become very prominent. The idea behind CI/CD is that any change made to the service at code-level or infrastruc-ture-level is automatically deployed to the cloud where the service is hosted. To create the Elastic Akka cluster we are also going to make use of CI/CD with Gitlab. Gitlab provides a solution for maintaining code repository as well as a way to define the var-ious stages from code-build to deployment using the ".gitlab-ci.yml" file. The pipeline is run using "runners" which are basically docker containers which run the commands specified in the .gitlab-ci.yml file. The process of deployment can be split into parts depending on the purpose, these are called stages. All the components used to create the system will use a CI/CD pipeline of their own.

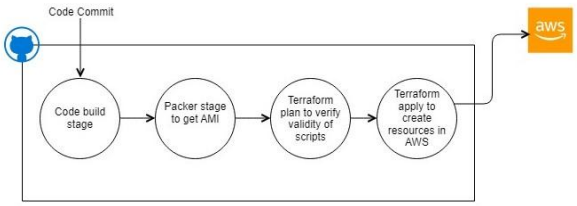

Fig. 1.Gitlab pipeline design with stages from code build to deployment.

B. Leader Nodes

When making use of Akka clustering, one of the crucial components is a "leader node", the leader node is responsible for managing cluster convergence and change in state of membership of any node. In this system we create 2 leader nodes, the nodes will be hosted in 2 separate EC2 instances in separate availability zones. We will also assign individual domain names to each instance to handle any IP address changes in case of redeployments/termination.

Revised Manuscript Received on June 30, 2020.
***** Correspondence Author
Utkarsh Upadhyay*, Advanta Development, Siemens, Bangalore, India Email: utkarsh.upadhyay241@gmail.com
Aviral Jain, Advanta Development, Siemens, Gurugram, India Email: aviraljain0286@yahoo.com

To create the instances, we build the AMI (Amazon Machine Image) using Packer, which is a tool created by Hashicorp to build machine images. The images are generated by specifying details like base image, files to be copied and the target directory in the instance and any commands to be run. The instances are deployed using terraform scripts with the AMI created, the same scripts also take care of assignment and creation of the DNS addresses for both instances. Both instances are private and can be accessed only within the VPC (Virtual Private Cloud).

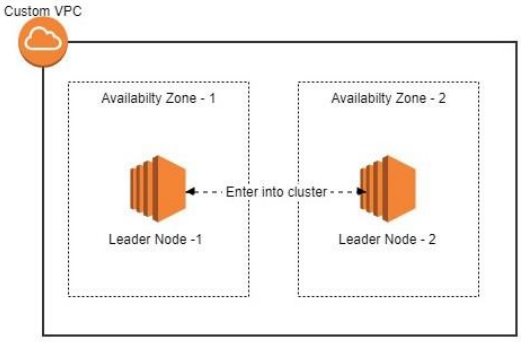

Fig. 2.Deployment view of Leader nodes within AWS VPC.

C. Launch Configuration and Worker Nodes

The worker nodes are the nodes which will join the cluster and provide the interface for clients to connect to the system. The worker nodes are what will be scaled to pro-vide elasticity to the system as they will be part of an autoscaling group. Within AWS, when we create autoscaling groups, a template is required for instances and is launched as a part of it, this information comes from the launch configuration. For the worker nodes, we will also use Packer to create the AMI however the base image used by the packer will be different from that used in the creation of the leader nodes as we will make some kernel level to increase the number of websockets which can be han-dled. The AMI will then be added to a launch configuration which will be associated with an auto-scaling group. The creation of the launch configuration and auto-scaling group will be done using terraform scripts.

D. Elastic Load Balancer and Target Group

The worker node auto-scaling group created can have ‘n’ number of instances at any given instant therefore we will employ AWS Elastic Load Balancer (ELB) to route and balance traffic across the worker nodes. The advantages of using ELB are:

- The scaling aspect of the Load Balancer is handled by AWS.
- The AWS ELB is very highly available, and it is unlikely to be any outage for it.
- Easy integration of SSL/TLS encryption to secure http/ws traffic by making use of ACM (Amazon Certificate Manager) certificates.
- Easy integration with instances in an auto-scaling group by adding them to a target group.

A target group is a group of resources to which the ELB will route traffic based on the evaluation of a rule. In the system we have implemented the auto-scaling group that will be associated with a target group which will be created using a terraform script. The target group is then associated with the ELB which will also be created using terraform, the ELB will route traffic to the instances as part of the auto-scaling group in a round-robin fashion to ensure equitable balance of traffic across all instances. AWS also provides various metrics at the load balancer level like the Active Connection Count which we will make use of to scale up/down the cluster.

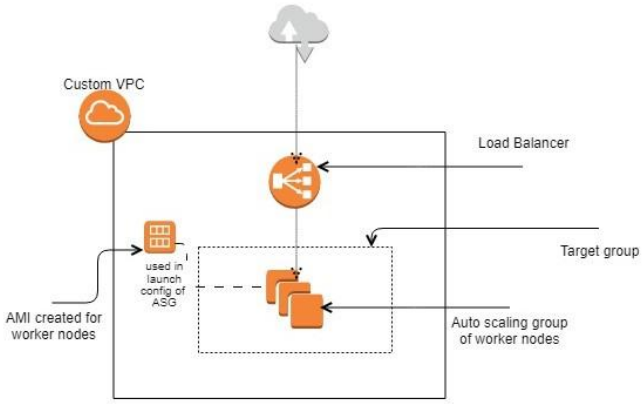

Fig. 3. **Deployment view of Worker Nodes**

E. CloudWatch and Lambda

Cloudwatch is the solution offered by AWS to monitor applications, respond to system-wide performance changes, optimize resource utilization, and get a unified view of operational health. Cloudwatch collects data in the form of logs, metrics, and events. Cloudwatch also provides alarms, which can be used to detect behavioral changes in resources and take automated actions. We will make use of 2 separate Cloudwatch alarms, one which will be triggered when the active connection count goes beyond a threshold while the second one will be triggered when the active connection count drops below a threshold.

AWS Lambda functions enable running code without the overhead of provisioning and managing servers, the other benefit of lambda functions is the fact that they can be triggered from AWS services like Cloudwatch alarms which is what we will be leveraging. We will employ 2 lambda functions; one will be responsible for increasing the number of worker node instances and the other will reduce the number of worker nodes. The reason behind using 2 separate functions is to reduce the complexity and duration of execution, it is also a logical separation of the job they are intended to do.

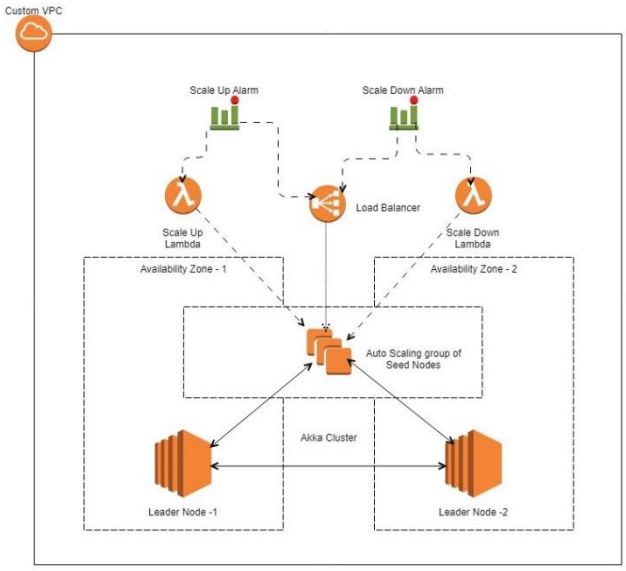

Fig. 4. **Deployment view of Elastic Akka Cluster with Websockets in AWS**

III. IMPLEMENTATION

This section will discuss the implementation details of the system and how the integration takes place.

A. Creation of Base AMI for Seed Nodes

AMI is basically a template for an EC2 instance. AWS also provides a feature where you can create AMIs from running instances by taking a snapshot of the instance, the snapshot can be converted into an AMI and can be used to deploy new instances. To begin with, we will create an instance with the basic Amazon Linux 2 AMI and make some kernel tweaks to raise the number of connections the instance will be able to handle. Every incoming or outgoing connection needs to open a socket which in Linux is a file, therefore to support more number of connections which in this case will be websockets we will first increase the limit on number of files which can be open on the machine. This is done by adding the highlighted lines of code in the below image to /etc/security/limits.conf file.

```
#*               soft    core            0
#*               hard    rss             10000
#@student        hard    nproc           20
#@faculty        soft    nproc           20
#@faculty        hard    nproc           50
#ftp             hard    nproc           0
#@student        -       maxlogins       4
*                soft    nofile          100000
*                hard    nofile          100000
root             soft    nofile          100000
root             soft    nofile          100000

# End of file
```

Fig.5. /etc/security/limits.conf with modification to increase the limit of number of files.

The change will take effect once the machine is restarted and can be verified by running the command **ulimit -n** which should give the output as 100000. We will now create the image of this instance and use it for our worker nodes.

B. Cluster Formation

To create the cluster we first deploy our leader nodes, to enter into a cluster the actor system uses “seed-node”, seed node is the node which a new member in the cluster will join after which the seed node will inform all other nodes in the system about the availability of the new member. For Leader Node -1 we will use the DNS of Leader Node - 2 as the seed node and vice-versa. We are using 2 nodes so that in-case one of the nodes goes down the ability to form the cluster will not be disrupted i.e. removal of single-point of failure. We are also keeping the leader nodes private and accessible only within the VPC as they are only going to take part in the cluster creation clients will not be connected to them. Leader nodes play a crucial role in the stability and scalability of the cluster therefore the above steps are taken to minimize the risk of them going down due to external interference.

To deploy our worker nodes, we will set the minimum number of instances in the auto-scaling group as two, the instances will have both DNS values for the leader nodes as part of the seed-node configuration, the worker nodes will try to join a cluster by sending a message to both the seed-nodes, and whichever node it gets a response from first, it will send the “Join” command to and become a part of the cluster.

Akka clustering makes the above process very simple by defining set fields to be used in the configuration file and making use of the Cluster class. In the case of the leader nodes, there may be a disparity in the start times so to prevent failure of the service Akka also provides retry options which can again be configured by providing pre-defined options in the configuration file.

```
akka {
  actor {
    provider = "cluster"
  }
  remote.artery {
    canonical {
      hostname = "127.0.0.1"
      port = 2551
    }
  }

  cluster {
    seed-nodes = [
      "akka://ClusterSystem@<DNS_of_Leader-2>:2551"]

    #Time after which join is attempted in case of failure
    seed-node-timeout = 10s
    #Value to shutdown system in case of unsuccessful join for 100s
    shutdown-after-unsuccessful-join-seed-nodes = 100s
  }
 #Takes care of shutting down JVM in case of unsuccessful join
 coordinated-shutdown.terminate-actor-system = on
}
```

Fig.6. Configuration file for Leader Nodes.

```
akka {
  actor {
    provider = "cluster"
  }
  remote.artery {
    canonical {
      hostname = "127.0.0.1"
      port = 2551
    }
  }

  cluster {
    seed-nodes = [
      "akka://ClusterSystem@<DNS_of_Leader-1>:2551",
      "akka://ClusterSystem@<DNS_of_Leader-2>:2551"]

    #Time after which join is attempted in case of failure
    seed-node-timeout = 10s
    #Value to shutdown system in case of unsuccessful join for 100s
    shutdown-after-unsuccessful-join-seed-nodes = 100s
  }
 #Takes care of shutting down JVM in case of unsuccessful join
 coordinated-shutdown.terminate-actor-system = on
}
```

Fig.7. Configuration file for Worker Nodes.

Due to the elastic nature worker nodes, will leave the cluster at the time of scaling down and the cluster needs to be made aware that the node has left, to do this we will make use of the “addShutdownHook” method of the “sys” object in Scala which will execute the code to leave the cluster by sending the leave command.

```
sys.addShutdownHook {
  val checkTerminationEvery = 500L

  Cluster(system).leave(Cluster(system).selfAddress)
  while (!Cluster(system).isTerminated) {
    Thread.sleep(checkTerminationEvery)
  }
  Await.result(system.terminate(), Duration.Inf)
  ()
}
()
```

Fig.8. Code for a node to leave cluster on shutdown.

C. Health Check and Authentication

For every instance behind an AWS load balancer it needs to be "healthy" for it to receive traffic, to check if an instance is healthy, the load balancer hits a defined HTTP endpoint and based on the response code marks instance as healthy or unhealthy. In our case we are hosting websocket (WS) servers behind the ALB so to configure the health check seemed like a challenge initially however, websocket requests are basically HTTP GET requests with headers "Upgrade: websocket" and "Connection: upgrade". We are rejecting all requests at the server level without those headers barring any requests with "/info" endpoint, any call to this will return an empty response with a HTTP 200 status code and this will serve as health check. While the data is encrypted during transmission by using SSL/TLS we also ensure that the client is also authenticated at the server using the "Authentication" header which should contain a valid JWT token issued by our Cloud Foundry UAA server.

```
HttpRequest(HttpMethod(GET),ws://<DNS_Of_Load_Balancer>/wsserver
List(Timeout-Access: <function1>, UpgradeToWebSocket: , X-Forwarded-For: <>, X-Forwarded-Proto: https,
X-Forwarded-Port: 443, Host: wss.eu1.crspng.mindsphere.io, X-Amzn-Trace-Id: Root=1-5e8d7a4a-c37622b9a5d7ca0e41ee33f0,
Upgrade: websocket, Connection: upgrade, Authentication: <>, Correlation-ID: 665, origin: localhost, Authorization:<> ,
sG2xEOaeJyCP26hJuDkW3A==, sec-websocket-origin: https://<DNS_Of_Load_Balancer>
```

Fig.9. Websocket request received at worker nodes.

```
private def handleHttp(ctx: ChannelHandlerContext, req: FullHttpRequest): Unit = {
  if (req.uri().contains("info")) {
    sendHttpResponse(ctx, req, HttpResponseStatus.OK, "Success")
  }
  else if(req.method != HttpMethod.GET) {
    sendHttpResponse(ctx, req, HttpResponseStatus.FORBIDDEN, "Unhandled HTTP method")
  }
  else if (!req.headers().contains(HttpHeaderNames.UPGRADE, HttpHeaderValues.WEBSOCKET, true)) {
    log.debug("Not a WebSocket Request")
    sendHttpResponse(ctx, req, HttpResponseStatus.BAD_REQUEST,
      s"No websocket ${HttpHeaderNames.UPGRADE} header")
  }
  else if(!req.headers().contains("Authentication") && req.headers().get("Authentication")){
    if(!validate(req.headers().get("Authentication"))){
      sendHttpResponse(ctx, req, HttpResponseStatus.BAD_REQUEST, "Invalid Token")
    }
  }
}
```

Fig.10. Code to define health check and validation for Websocket requests.

D. Scaling Action of Lambda

As described in 2.5 we will be using Cloudwatch and Lambda to perform the scaling actions, this section will describe the implementation details of the two. At this point we have set up the leader nodes, load balancer and the auto-scaling group under it with 2 worker nodes as the minimum i.e. we can have 200,000 clients connected to this cluster. However, we will set the threshold for the scaling up alarm at 50,000, this is done to avoid the overloading of any instance in the cluster as shown in the table below (the calculations below are done based on the round-robin routing of the load balancer). During scale down, auto scaling will remove the node which was created last therefore, the number of clients impacted will be the lowest across all nodes.

Table- II: Traffic distribution across worker nodes at different levels of load.

Connections	~ Connection across Nodes * 1000			
	Node 1	*Node 2*	*Node 3*	*Node 4*
Upto 50000	25	25	N/A	N/A
Upto 100000	42	42	16	N/A
Upto 150000	54.5	54.5	28.5	12.5

[a] System Scales to 1 million concurrent connections in this way

In our set up we are taking the maximum capacity of the auto scaling group as ten i.e. we are defining the threshold of the system at 1 million concurrent clients. For the scale down alarm we will set the initial value as 1 million and it will be adjusted to 50,000 after the first scale up i.e. it will then trigger scale down lambda when the number of active connections drops under 50,000, this action will be performed by the scale up lambda using a Boolean flag "FIRST_INVOCATION". Before performing scale up or scale down the lambda functions will check that the scaling action does not reduce the worker nodes below 2 or increase them beyond 10. The lambda functions on being invoked will be responsible for adjusting the "Desired Capacity" value of the auto scaling worker node group, within the bounds of "Minimum Capacity" i.e. 2 and "Maximum Capacity" i.e. 10. The lambda functions will also update the threshold val-ues for their respective alarms by a factor of 50,000 i.e. Scale Up lambda will increase the threshold of Scale Up alarm by 50,000 and the Scale Down lambda will decrease the threshold of the Scale Down alarm by 50,000.

To minimize the iterations in searching for the alarms and auto scaling group we will use terraform outputs to write the values for the alarm ARN (Amazon Resource Name) and Name to a file, these values will then be utilized in the environment variables of the Lambda functions. Another detail to note is the fact that while using the SDK we will not be configuring any credentials, instead we will be using an IAM role with per-mission for Auto Scaling and CloudWatch which will be attached to the lambda functions.

```
AmazonAutoScaling asgClient = AmazonAutoScalingClientBuilder.standard().build();
DescribeAutoScalingGroupsResult asgResult = new DescribeAutoScalingGroupsResult().
        withAutoScalingGroups(new AutoScalingGroup().
        withAutoScalingGroupName(System.getenv( name: "WS_ASG_NAME")));
List<AutoScalingGroup> asg = asgResult.getAutoScalingGroups();
Integer currentDc = asg.get(0).getDesiredCapacity();
SetDesiredCapacityRequest dcRequest = new SetDesiredCapacityRequest()
        .withAutoScalingGroupName(System.getenv( name: "WS_ASG_NAME")).withDesiredCapacity(currentDc + 1);
SetDesiredCapacityResult dcResult = asgClient.setDesiredCapacity(dcRequest);
```

Fig.11. Code to manipulate Auto Scaling Capacity.

```
AmazonCloudWatch cwClient = AmazonCloudWatchClientBuilder.standard().build();
DescribeAlarmsResult alarmResult = new DescribeAlarmsResult().withMetricAlarms(new MetricAlarm().
        withAlarmName(System.getenv( name: "SCALE_UP_ALARM")));
List<MetricAlarm> alarm = alarmResult.getMetricAlarms();
Double currentThreshold = alarm.get(0).getThreshold();
PutMetricAlarmRequest updateAlarm = new PutMetricAlarmRequest().withAlarmName(System.getenv( name: "SCALE_UP_ALARM"))
        .withThreshold(currentThreshold + 50000);
PutMetricAlarmResult result = cwClient.putMetricAlarm(updateAlarm);
```

Fig.12. Code to manipulate CloudWatch alarm threshold manipulation.

```
if(System.getenv( name: "FIRST_INVOCATION").equals("false")) {
    PutMetricAlarmRequest updateScaleDownAlarm = new PutMetricAlarmRequest().
            withAlarmName(System.getenv( name: "SCALE_DOWN_ALARM"))
            .withThreshold(50000.0);
    PutMetricAlarmResult scaleDonwResult = cwClient.putMetricAlarm(updateAlarm);
}
```

Fig.13. Code to handle setting Scale Down Alarm at first Scale Up Action.

IV. RESULT AND DISCUSSION

The architecture developed creates a fault-tolerant ad highly available system which scales in an automated fashion to tackle the problem of persistence which comes when working with websockets. Below are some points to note:

1) Each component of the system is deployed to the cloud using full automation in the CI/CD discussed in II.A.
2) There is a significant cost advantage, which we will discuss here using two use cases:

- Case 1: The workload to be managed is of 50,000 clients connected at the same time for a period of 3 hours. If a bare Amazon Linux AMI is used we will require 13 instances, let us assume the instances are of t2.medium type below is the cost calculation.

Cost = (No. of instances * No. of hours * Cost of t2.medium per hour) + (Cost of Cloudwatch Alarm invocation * No of invocations) (1)

Cost without Elastic Akka Cluster = 13 * 3 * $0.0464 = $1.8096

Cost with Elastic Akka Cluster = 4 * 3 * $0.0464 = 0.5568

In case 1 we can see almost a 4 times reduction in the cost, this difference stems from the AMI prepared in II.C which raise the connection limit of each instance from 4000 to 10,000. We are not charged for Cloudwatch and Lambda as they are charged per invocation.

- Case 2: We have a workload of 100,000 clients where 50,000 clients are connected for 2 hours and 50,000 clients are connected for 6 more hours i.e there will be 100,000 clients connected for 2 hours and 50,000 for 6 hours.

Using equation (1)

Cost without Elastic Akka Cluster = 25 * 6 * $0.0464 = $6.96

Cost with Elastic Akka Cluster = (5 * 2 * $0.0.464) + (4 * 4 * $0.0464) + (2*0.10) = $1.4064

In case 2, we can again see the difference in the cost even though we use 2 invocations of the Cloudwatch Alarm (1 Scale Up and 1 Scale Down).

3) With the introduction of Akka Clustering any load of processing the data pushed to and from clients can be easily distributed across all the nodes of the cluster. Akka framework abstracts the details of how the load is distributed and all Actors across the nodes communicat e seamlessly with each other like they are deployed on a single machine

V. CONCLUSION

Websocket protocol is a great solution for continuous real-time data transmission their persistent nature can make it very difficult and expensive to create a system to deal with many concurrent clients, by inducing elasticity into the system we have created a way to bring down infrastructure cost as well as automate the task of scaling infrastructure where traditional methods of memory and CPU utilization based scaling are not applicable. While we have only presented the use of Akka to create a resilient and fault-tolerant cluster, the elements of the Actor model along with the other modules of Akka framework like Streams can be used across the system to perform high-performance computation and manipulations being pumped into the system via the websockets. Furthermore, using Gitlab, CI/CD and infrastructure as code with Terraform the system also adheres with the latest practices around Cloud based development and deployments.

VI. ACKNOWLEDGMENT

We would like to thank Ms. Sukriti Sinha (Graduate, University of Arizona), Mr. Amod Paranjape (Network Service Delivery Engineer, Schneider Electric) and Ms. Aastha Sinha (Student, Purdue University) for taking the time to review this paper and provide valuable insights.

REFERENCES

1. Qigang, Liu & Sun, Xiangyang. (2012). *"Research of Web Real-Time Communication Based on Web Socket"*. International Journal of Communications, Network and System Sciences. 05. 797-801. 10.4236/ijcns.2012.512083.
2. Akka Actors https://doc.akka.io/docs/akka/current/typed/actors.html#akka-actors.
3. Akka Cluster Usage. https://doc.akka.io/docs/akka/current/typed/cluster.html.
4. Amazon Web Services (2019). AWS Well-Architected Framework.
5. Amazon Web Services (2018). Operational Excellence Pillar.
6. Raymond Roestenburg, Rob Bakker & Rob Williams (2016). *"Akka in Action"*.

AUTHORS PROFILE

Utkarsh Upadhyay, B.Tech in Computer Science from SRM University (2017). Presented "Secure File Network for Cloud Based Organizations" at ICASET'17. Currently part of R&D team for Siemens Mindsphere.

Aviral Jain, B.Tech in Electrical and Electronics Engineering from VIT University (2017). Currently part of R&D for Siemens Digital Industries.