Cloud Computing & Big Data

Joseph Eastoe - ur19476

University of Bristol

Abstract. Cloud Computing has many advantages, one of them is performing a task in parallel, which can be scaled up with low cost (due to it being easy to horizontal scale). To take full advantage of this I chose to do a embarrassingly parallelisable task which leverages cloud capabilities of scalability and on-demand. My task was to brute-force url paths for a given website, this is an appropriate task for cloud computing as it is parallelisable and benefits from increasing the number of threads hence cheap scalability which cloud computing offers, is a advantage for this task. The heart of my architecture is RabbitMQ queue where all the workers get their work from. Them being both inside a kubernetes cluster and the results are posted to DynamoDB database.

1 Introduction

My embarrassingly parallelisable task was to brute-force url paths for a given website. I used Gobuster[1], which is a brute-force url paths tool. Written in Go it uses concurrency using multiple concurrent threads under the hood.
Penetration testers are used to test the security of a system by taking the role of the attacker. Scanning websites for hidden url paths is a common tool used by penetration testers. A swifter tool could scan higher number of possible url paths in same amount of time, than a slower tool. Thus increasing the chances of finding a hidden path and improving the security of the website. For example, the website could unknowingly allow the external internet to access an internal API, which could be a security risk, thus quickly finding detects the issue and allowing it to be migrated, therefore increases security.

This task is appropriate for cloud computing as it is parallelisable and benefits from increasing the number of threads thus the cheap scalability that cloud computing offers is a advantage for this task. Cloud computing also allows for the url path requests to come from different IP addresses thus making it more difficult for the website to block the requests.
CAP theorem[3] states that any distributed data store can provide at most two of the following : Consistency (All nodes should see the same data at the same time); Availability (Node failures do not prevent survivors from continuing to operate); Partition tolerance (The system continues to operate despite network partitions). Clouding Computing, being a type of distributed data store, is governed by this theorem, thus the key for a successfully cloud computing system is to get trade off between the three properties, which weaken some definitions

of them.
One way to achieve high availability and consistency is by implementing fault tolerances. This is explain in the below section.

2 Application Architecture

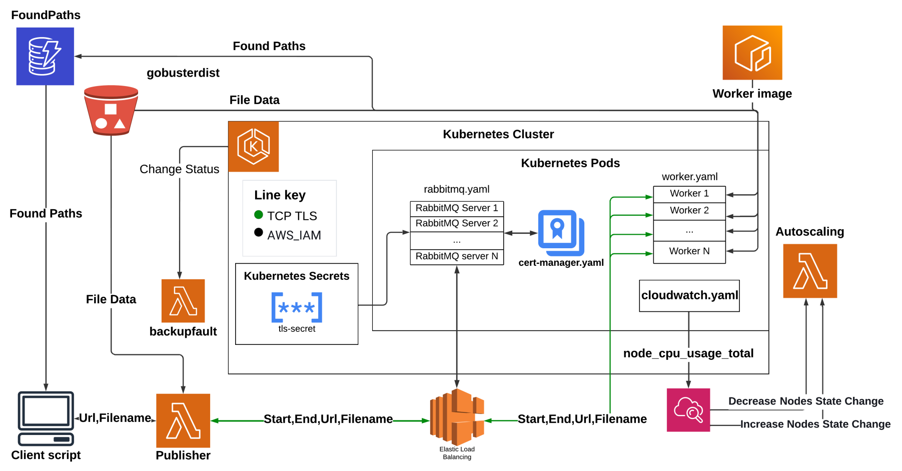

Fig. 1. Diagram of Architecture

2.1 Description

The glue to my architecture was kubernetes, I used AWS EKS to set up high availability kubernetes cluster. EKS from the box actively monitors the load on control plane instances (responsible for managing the worker nodes and the Pods) and automatically scales, and replaces them if they are unhealthy instances. Kubernetes offers some self healing properties, for example detecting and responding to cluster events starting up a new pod when a deployment's replicas field is unsatisfied, thus doing fault tolerance.
I used RabbitMQ to create a message queue, where one publisher would queue messages and multiple consumers would consume them allowing for easy scalability, by increasing the number of consumers. I chose to only use one publisher as for this task the publisher does not do any hard computation. RabbitMQ has excellent fault tolerance and I wanted to take full advantaged of this. I Implemented publisher confirms, when the publisher lambda function pushes a message to the queue, the queue acknowledges that the message have been be received which ensure guaranteed delivery of said message. If publisher does not

get "basic.ack" after a certain length of time, then it tries again. The language I used for the publisher lambda function was java because RabbitMQ Java client allows for publisher confirms, unlike the equivalent in python (pika). Thus to achieve the same result in python I would have to use RCP calls which would be costly as the lambda function would need to be live, until the workers finish and callback. I also implemented consumer acknowledgement, which along with heartbeats ensures that if a worker goes down, the messages that they consumed are re queued. I used RabbitMQ quorum queue which uses multiple pods to host the queue. Doing all the above ensure that all published confirmed messages are not lost as long as at least a majority of RabbitMQ nodes hosting the quorum queue are not permanently made unavailable [2]; thus achieving strong fault tolerance. To strengthen this I implemented lambda function backupfault, which is triggered when EKS cluster failed node count is more than 5. It deletes the cluster and nodegroup if presented, making a new cluster and deploying the whole application again and invokes the publisher function with the URL and filename that client inputted before the fault. This is to act as last resort if the whole EKS cluster itself fails.
Apart from using kubernetes and DynamoDB (which is autoscalling by default), I did autoscalling by creating a lambda function called Autoscalling, which gets triggered when the pod_cpu_utilization metric for the default namespace (one used by the worker) is too high or too low, it then increase the amount of nodes in the node group or decreases accordingly. The cloudwatch.yaml sends data back every 60 seconds to ensure fast autoscalling. I also implemented a limit for the amount of nodes, ensuring that vcpu limit is not reached (causes the nodegroup to become deranged) if an adversary tries to maliciously harm the system.
I initially wanted to use EKS build in autoscalling feature however was unable to so due to the restriction on aws academy lab account. However I glade I implement autoscalling myself, has it allowed me to set the maximum and minimum limits for the node count in the nodegroup ensuring that work can always be done and limiting malicious requests.
Above is the flow of system. To note : I implemented TLS for the RabbitMQ server as I wanted to ensure security and longevity, for the system. I purpose chose to make the DynamoDB server "FoundPaths' partition key to be the path found itself (for example "robot.txt"). As if a worker fails in between, inserting the found paths into "FoundPaths" and the consumer acknowledgement (this show in the diagram to be the thick red line), there could be repeat entries as the unacknowledged message would be re-queued ; But because the partition key is the path itself DynamoDB solves this issue (as no repeats with the same key are allowed).
Some other potential alternative architecture and there reasoning why I did not choose them : Using Apache Kafka instead of RabbitMQ, my reasoning here is that rabbitmq offers better fault tolerances, is designed to be a message queue unlike Kafka and Kafka is more tailored for big data, consumer and publishers processing large batch of messages at once which is not needed for this cloud

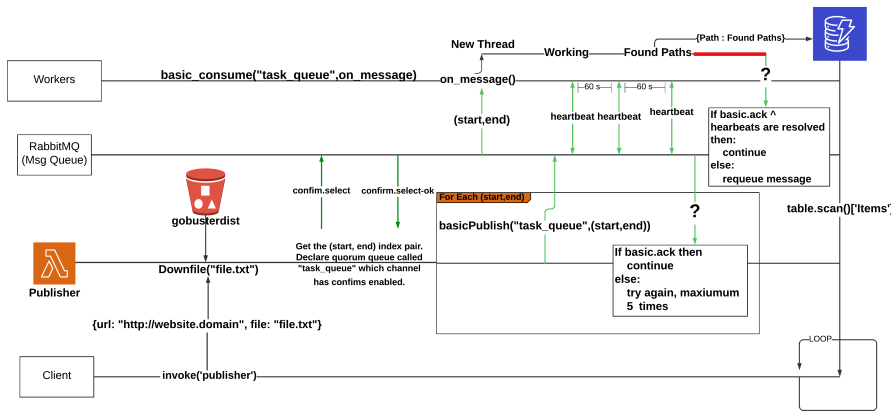

Fig. 2. Diagram of showing the flow of the system

system.
Running the publisher lambda function code on the client side instead. The reasoning is that I wanted to have a light weight client and all the processing to be done in the cloud.

2.2 Performance & Cost

The main two advantages of auto scalling achieves are better performance and cost management. If the system needs extra resources then scale up, and if it has too many resources, it can scale down to save cost.
As mentioned earlier, I invoked the autoscalling lambda function based on a metric. I determined this metric by creating alarms for the following list of metrics: node_cpu_usage_total, pod_cpu_utilization, node_filesystem_utilization, node_memory_utilization, node_cpu_utilization and node_network_total_bytes.
I then set up a simple Node.js server that would log all the paths tried on the website in a file called "triedpaths.txt" listening on my localhost. I started ngrok to expose this to the internet, forwarding requests to a unique URL to my localhost server.
Using the client script, I set the url to look at and a filename of file that was stored in my S3 bucket (using a 5418 KB file). I recorded the time started and the time ended. Of the list I picked pod_cpu_utilization, node_cpu_usage_total and node_filesystem_utilization. Then for each one, I made it the metric for the lambda function autoscalling, I chose the pod_cpu_utilization metric because it made the system finish the fastest but also scaled down at the end. I also compared them to no auto scaling, which all performed quicker as expected.

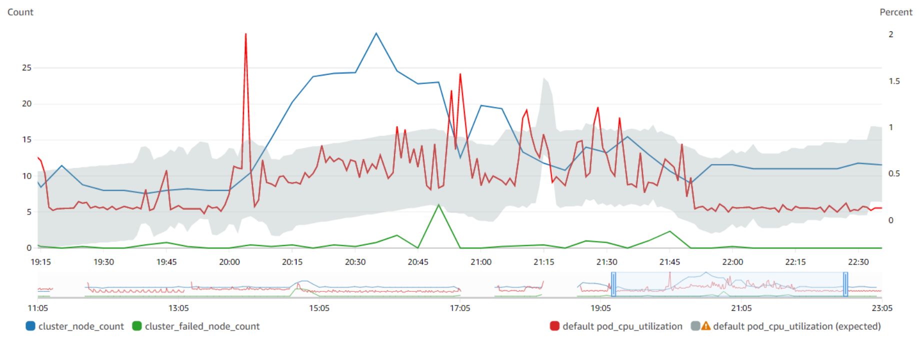

Fig. 3. Cloudwatch of Autoscalling

Using the simple Nodejs server in the same way as stated above, I did the following:

Table 1. Costs

Cloud	Time (hours)	Cost
Full system	1:49	$1.931
No auto scalling	2:32	$1.742
Single t3.medium	15:85	$0.6625

The reason I used a single t3.medium instance on AWS instead of a local machine is that I wanted to accurately represent an attack machine located far away from the target, rather than being on the same network. To calculate the cost, I first looked at the breakdown in AWS Cost Explorer to see which services were spending money. This is because some AWS services only start charging after a certain threshold of usage is reached. It turned out that all services except for the Lambda functions were charging. Since they were running all the time, I calculated the cost per hour for each, multiplied it by the number used and the time, all added up to get the total cost.
For autoscaling due to the changing number of nodes, I had to calculate the area under the blue line in Figure 3 between 20:03 and 21:52. To do this, I exported the CSV data of the blue line and imported it into Excel, then calculated the area under the graph. I then multiplied this area by the rate, and added the cost of the other services to get the total cost.
EKS : $0.10 hr. one EC2 node : $0.0416 hr. cloudwatch, used AWS cost explore depends on metrics sent : $0.21 hr. ELB : $0.025 hr
Looking at the information above, it is clear that this system benefits from being in the cloud, completing tasks about 10 times faster while only being 3 times more expensive. This is important because it allows for more potentially vulnerable URL paths to be discovered in the same time. Another thing to note is how autoscaling speeds up the system without a drastic increase in cost. This

makes sense when looking at Figure 3, which shows that the system scales down quickly when pod_cpu_utilization starts to drop, allowing for extra resources not to be wasted and lowering cost.

2.3 Fault Injection

Fault tolerance is important for my cloud application, as missing messages would cause the workers to miss part of the word list file. If the part they missed included a valid path (for example, one that returns a 200 status code), then the application would miss this path.
There are three places where I tested the fault tolerance of my system. The first was to randomly crash the Python worker scripts, to see if the messages they received would be requeued. The second was to force shutdown the Kubernetes pods of the RabbitMQ server. The last was to crash five or more nodes, invoking the backup fault lambda function, which would recover the system with no data loss.
To test that the shutdown of worker scripts works, I first implemented os.exit(0) in the script itself, which would happen one-tenth of the time. I then removed this and created a script that would kubectl delete a random worker pod. The results were the same.

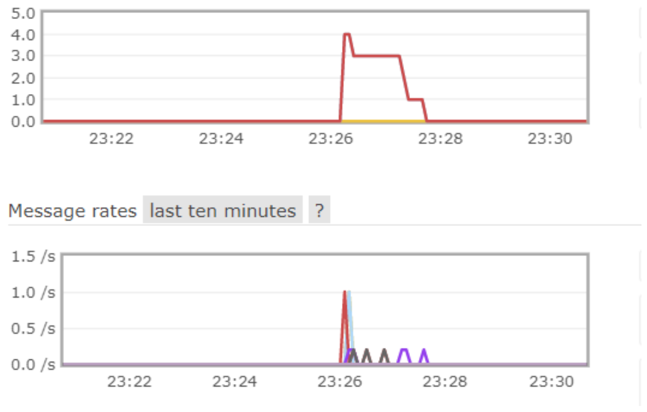

Fig. 4. RabbitMQ Dashboard

Looking at the first graph above, the red lines represent the total number of messages (unacked plus those in the queue), and the yellow line is the number of messages in the queue.
Looking at the second graph above, the red lines represent messages being published to the queue, the light blue lines are publisher confirms for those messages, the purple lines are consumer acknowledgements, and the black line is redelivered messages.

As you can see, when RabbitMQ doesn't receive a consumer acknowledgement and the heartbeat is broken (which will happen if the worker crashes), it redelivers the unacknowledged message, which another worker can pick up or the same worker can pick up once Kubernetes reboots it.
I had some issues testing the worker script shutdown, as for some reason shutting down too many workers would cause the Amazon EBS CSI Driver to become "deranged," which in turn would cause the RabbitMQ server to crash, causing the workers to crash as well. Given more time, I would like to solve this issue as this would be a single point of failure for the cloud application.
The second way I tested the fault tolerance of the system was to force shutdown the RabbitMQ server pods and see if the RabbitMQ dashboard was still up, suggesting that the server was still running normally. Provided that I didn't terminate the majority of the servers, the dashboard was still up, which is expected according to the RabbitMQ docs.
The third way I tested the fault tolerance of the system was to abruptly shut down six EC2 instances that were nodes in the Node group used by the Cluster. This triggered the backupfault lambda function, which deletes everything and rebuilds the whole application. The system saves the inputted URL and filename from the client script, so if the backupfault function is triggered, it can invoke the publisher lambda function and no information is lost.
Initially, I wanted the backupfault function to delete and rebuild everything. However, due to the time limit on lambda functions and the time needed for the cluster/nodegroup to be live, this wasn't possible. So, I made it delete the nodegroup and create a new one without interfering with the cluster, if no EKS cluster exists it will create a new one. The EKS cluster is fault tolerant anyway, so I found this to be a good compromise.

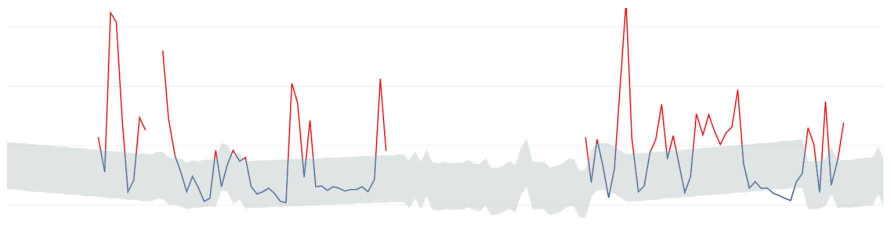

Fig. 5. Cloudwatch of backupfault

Above is the backupfault function working (where the line is pod_cpu_utilization), you can see there a period of shutdown then continue with the job

2.4 Conclusion & Limitations

In this paper, I presented a full fault tolerance and autoscaling cloud system for. My system leveraged the power and scalability of cloud services such as Amazon

EKS, EC2, and Lambda to create a highly available and cost-effective solution. I demonstrated how our system could handle various types of failures, such as worker script crashes, RabbitMQ server shutdowns, and node crashes, without losing any data or affecting the performance of the application.

Through extensive testing and evaluation, I showed that my system was able to complete tasks up to 10 times faster than a local machine, while only being 3 times more expensive. I also demonstrated how our system could dynamically scale up and down according to the workload, allowing for efficient resource utilization and cost savings.
One other possible improvement that was not mentioned above is the use of a more robust web server for testing. During testing, I used a the simple Nodejs server, but due to using ngrok, some requests did not go through because ngrok was not able to handle all of them. In the future, I would set up an EC2 instance to host the simple web server and ensure that it can handle many requests at once by using gateways and load balancing

References

1. Gobuster, https://github.com/OJ/gobuster. Last accessed 14 December 2022
2. Quorum Queues—RabbitMQ, https://www.rabbitmq.com/quorum-queues.html. Last accessed 7 December 2022
3. Fox, A., & Brewer, E. (1999). Harvest, yield, and scalable tolerant systems. In Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (pp. 174-178).

Appendix

The source code is found https://github.com/ccdb-uob/CW22-45 including a README.md to recreate cloud system on a new aws account.