NETWORK PROGRAMMING AND DISTRIBUTED APPLICATIONS - LAB 4

Brikelda Sema
Lulea Technical University
Skelleftea, Sweden
semabrikelda7@gmail.com

1 Agent-Based Attacks on a TCP Serverr

For Part 1 of the assignement we are required to build an Architect agent with GUI, which will be responsible for creating several Agent Smiths (attacker agents) which will attack a multi-threaded TCP Server (that we need to build as well). The agent smiths have a ticker behaviour which runs periodically, for the Part 1 just to open a TCP socket. The attack should vary in terms of number of agent smiths and ticker interval and to analyse CPU metrics of the TCP Server.

To build this architecture, I used 2 AWS EC2 instances: one for the agents and one for the TCP Server. I first created the EC2 instance for the agents, where I initially installed java 17, jade and vncserver (to display the GUI locally). These are the commands I used to install the respectively:

```
#Install JAVA
sudo apt install openjdk-17-jdk -y
#Install JADE
wget https://jade.tilab.com/dl.php?file=JADE-bin-4.6.0.zip -O jade.zip
#Install VNC Server
sudo apt install tightvncserver -y
```

After installing Jade, I had to unzip it, take the jade.jar file path, but it in CLASSPATH and put the CLASSPATH as an environment variable in ~/.bashrc, whereas for the VNC Server I had to start it, set a value for DISPLAY (like :1) and connect it to my local TightVNC Viewer (client).

Since I am supposed to run 10000 agent smiths, I picked t3.large instance type (with 8 GB of memory) and also attached an EBS Volume for each of the instances. I mounted the volumes with the instances and saved everything in the volumes. For the TCP Server I picked t3.medium, because we need it at some point to get overloaded. I decided to go for a single instance for the agents since the agent smiths are automatically created by the Architect agent, and since the number of these agent smiths varies depending on the users input. It is more cost efficient to scale vertically in this case rather than horizontally, because there might be instances that run without any process on them (if we create a small number of agent smiths) and it is also easier to manage. So instead, I decided to group these agents into separate containers when they are created, to better manage them.

After setting up the environments, I create the script for ArchitectAgent.java, which is responsible of displaying a GUI with 3 fields, where the user can enter the number of agent smiths, ticker interval and attack duration, and a button start. Then, this ArchitectAgent creates the agent smiths by sending a request with the given parameters to AgentSmith.java. Agent-

Smith.java takes the parametres, and sets the ticker behaviour for a certain time to open a TCP socket at the hard-coded DNS name (in my case the DNS Name of the TCP Server instance is: ec2-13-49-222-43.eu-north-1.compute.amazonaws.com) and port 1234 and give some console outputs ("Connected to server:", "Error: Unable to connect to server" etc). On the other EC2 instance, I created the script for the TCP Server, which listens to port 1234 and handles multiple connections (multi-thread) with this line : "new ClientHandler(clientSocket).start();" and communicates with them. It creates several instances of agent smiths by grouping them into containers of 1000 agents.

Having created everything neccessary, the attack can be launched. However, before doing that I set an alarm on CloudWatch that if the CPU Utilization is above 80%, the alarm should be triggered, but no particular action should be taken, just for demonstartion purposes. So, I SSH into my two EC2 instances and start the TCP Server and an instance of ArchietectAgent:

```
#From EC2 Instance "Agents Brikelda"
java jade.Boot -gui
java jade.Boot -container ArchietcAgent:ArchitectAgent

#From EC2 Instance "TCP Server Brikelda"
java TCPServer
```

1.1 No Overload Scenarios

After starting them, I conduct several experiments and give as parameters through the GUI to my ArchitectAgent to create 5000 and 10000 agents, for intervals of 1000 ms and 4000 ms and for duration of attack 60000 ms. Fig.1 shown the communication between the agents and TCP server for creating 10000 agents, with ticker interval 0f 1000 ms and attack duration of 60000 ms. In this case 10 containers were created (Conatiner0-Conatiner9), since there are 10K agents and each container holds 1K agents.

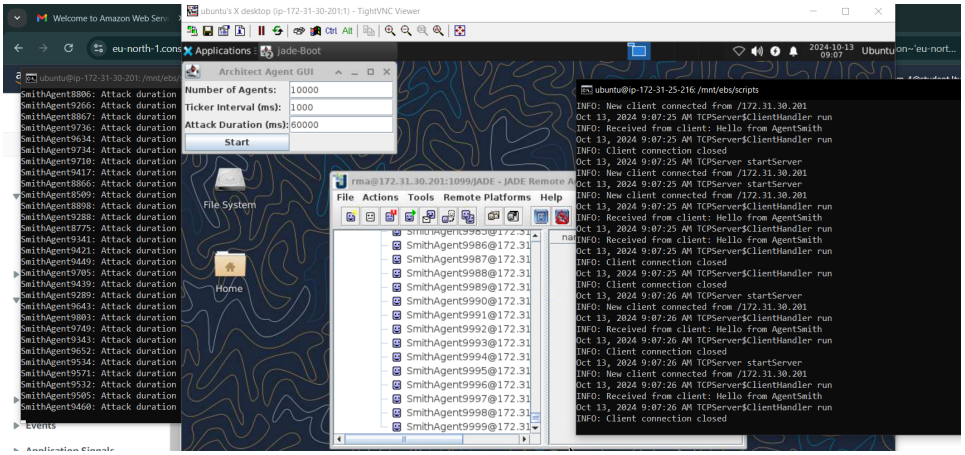

Figure 1: Attacking with 10000 agents with ticker interval 1s and attack duration 1 min

To check the CPU Utilization for the TCP Server, I would take the instance_id of the TCP Server, which is "i-080ba87c1066271ea", go to CloudWatch, metrics, browse the instance_id, select the CPUUtilization metric, check the graph and download the .csv format for that experiment. I conducted 3 scenarios for this part:

- **Scenario 1:** 10000 agents, ticker interval of 4 seconds
- **Scenario 2:** 10000 agents, ticker interval of 1 second
- **Scenario 3:** 5000 agents, ticker interval of 1 second

The table 1 shows the peak CPU Utilization value for each scenario (the data were aggregated from the CloudWatch .csv files).

	Scenario 1	**Scenario 2**	**Scenario 3**
CPU Utilization	35.983%	47.376%	45.206%

Table 1: CPU Utilization for the 3 scenarios

It is evident that the CPU Utilization for 10K agents with 4s interval is lower than 5K agents with 1s interval. At first we might thing that there's something wrong, but it actually makes sense. For Scenario 1, in any given second, only 1/4th of the agents are active because of the 4-second interval. So, at any moment, approximately $\frac{10000}{4} = 2500$ agents are active. However, for Scenario 3, all 5,000 agents are active every second. Since the CPU usage generally scales with the number of active agents performing operations, the Scenario 3 (handling 5,000 agents with a 1-second interval) would require more the CPU load per second compared to Scenario 1, this is why this last one has less CPU Utilization, even though it's being attacked by more agents. And obviously, Scenario 2 has the most CPU usage. This cahnge in resource utilization can also be proven using NetworkIn and NetworkOut (number of bytes entering and leaving the system), as shown in Fig.2. First spike is more Scenario 1, second spike for Scebario 2 and thrird spike for Scenario 3. More bytes travel when more agents are active.

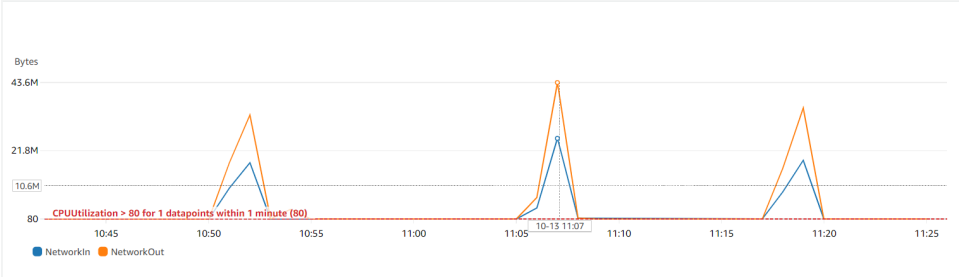

Figure 2: NetIn & NetOut graph for the three Scenarios

However, we can see that in the experiments, the TCP Server doesn't really get overload; actually the CPU Utilization doesn't even reach 50%, even though we are attacking with 10K agents. This happens because these agents don't really do anything apart from opening a TCP socket. Thsi command doesn't overload the server, because the server is doing nothing else for them. In the next subsection we are introducing new scenarios where the server does actual job for the agents.

1.2 Overload Scenarios

To overload the server, the agents have to ask it to do a certain job. I took the case of computing the Fibonacci value of a certain integer, which I hard-coded to be 30. So, I edited the AgentSmith.java by giving it an int fibonacciNumber = 30; and then it send it to the TCP Server when it tries to create the connection like: out.println(fibonacciNumber); and from the TCPServer.java, I added the function to calculate the Fibonacci value of the integer sent by the agent smith. The function is set to be not efficient on purpose, to overload the server:

```
// Inefficient recursive Fibonacci calculation
    private long inefficientFibonacci(int n) {
        if (n <= 1) {
            return n;
            }
        return inefficientFibonacci(n - 1) + inefficientFibonacci(n - 2);
        }
```

I compiled the scripts, and conducted attacks with the purpose to overload the server. The three experiments I did using Fibonacci were:

- **Scenario 1:** 1000 agents, ticker interval of 1 second
- **Scenario 2:** 5000 agents, ticker interval of 1 second
- **Scenario 3:** 10000 agents, ticker interval of 1 second

Same as for the other scenarios, I went to CloudWatch to download the .csv files for each scenario and analyse them. Table 2 shows the peak CPU Utilization value for each scenario.

	Scenario 1	**Scenario 2**	**Scenario 3**
CPU Utilization	65.201%	93.887%	94.187%

Table 2: CPU Utilization for the 3 scenarios using Fibonacci

Finally, our server got overloaded when attacked with 5K and 10K agents that where asking it every second to compute the Fibonacci value of 30. We can see that there's not a huge difference between Scenario 2 and Scenario 3 results, which is due to the overloading of the server and the fact that it starts to not respond back or accept connections (because it went down). Fig3,4 and 5 show graphical representations of the CPU Utilization for each scenario.

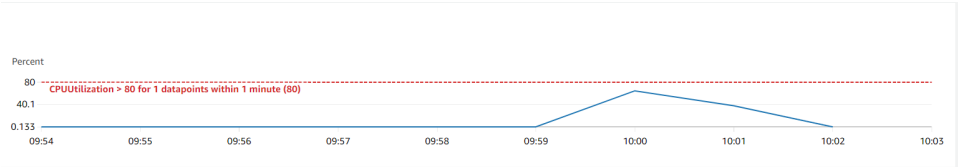

Figure 3: CPU Utilization for 1K Agents attack using Fibonacci

Figure 4: CPU Utilization for 5K Agents attack using Fibonacci

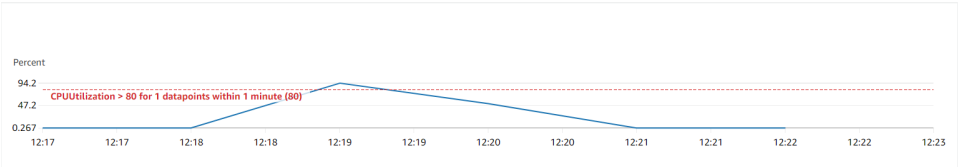

Figure 5: CPU Utilization for 10K Agents attack using Fibonacci

By looking at the graphs we can see the effect of overloading the server. The attacks I conducted had a duration of 1 minute. For example, at Fig5, I started the attack at 12:18 and it ended at 12:19. However, we can see that the server still works and delays some responses even 2 minutes after the attack has ended (12:21) because it was overloaded before. The same situation happens with the other 2 scenarios, the server delays the responses when it is overloaded.

2 AWS Extended Architecture

In this section we are required to extend the archicture by incorporating a load balancer and auto scaling for our instances. To do this, I firstly created the load balancer, whose role is to distribute evenly the incoming TCP traffic across multiple EC2 instances running my TCP server.re the steps I followed to create the load balancer:

1. Select the right Load Balancer: AWS offer three options of load balancers (Application LB, Network LB and Gateway LB). Since we are going to work with incoming TCP traffic, the Network Load Balancer is the best option, ensuring high performance and low latency. I selected the NLB and named it NBLBrikelda.
2. Basic Configuration: For the NLB Scheme I selected *Internet-facing*, supposing the clients will be connecting from the internet, and the IP address type as IPV4.
3. Network mapping: Here we have to select the VPC were the LB will exist and scale. Since my EC2 instances are already part of a VPC, I selected that one, which is karan_test. This VPC hosts addresses in the range 172.31.0.0/16, and my Agents Brikelda and TCP Server Brikelda instances have private IP addresses 172.31.30.201 and 172.31.25.216 respectively. They belong to Availability Zone eu-north-1a (eun1-az1), this is why I selected this one, and the corresponding subnet within the VPC (subnet-2a759743).
4. Security groups: For security groups, I selected the one I had attached to my previous instances (launch-wizard-75), which allows inbound TCP traffic at port 1234.
5. Listeners and Target Group Creation: In this part I initially created a target group (named it TCPServersTrafficBrikelda), which will route traffic to my EC2 instances, so it's very important to define the traffic type as TCP and the target type as Instances within karan_test VPC. Then, after creating the target group, I selected it when creating the load balancer and defined the listener (determines how the load balancer routes requests to its registered targets) to route the TCP traffic coming from port 1234 to the target group (so my instances).

Finally, the NLB was created (Fig.6) and ready to be attached to the Auto Scaling Group.

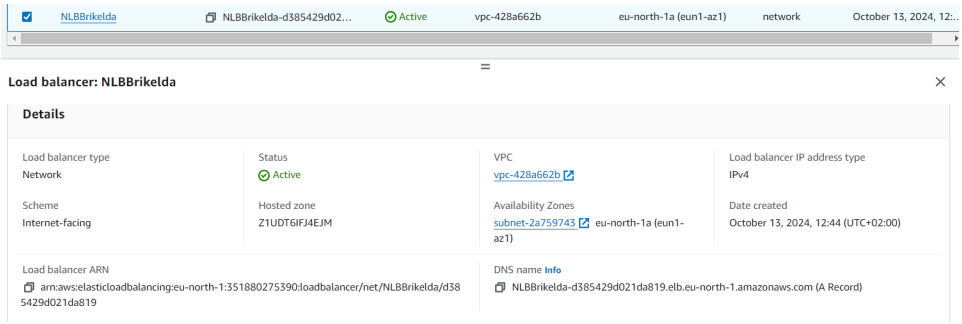

Figure 6: Network Load Balancer Dashboard

The next step is to create an Auto Scaling Group, which will scale-out (increase the number of TCP Server instances) or scale-in (decrease the number of TCP Server instances) based on some traffic rules that we will define. The steps I followed to create my ASG were:

1. Configuration of Launch Template: The main idea of ASG is to launch or terminate instances based on the traffic load or a specified type of metric, but to do so we have to define the template of these newly created instances. In my case, I took the initial TCP Server Brikelda instace that I had created and I created an AMI out of it, with the purpose to use this AMI for the launch template of my ASG. So, all the new created instances due to auto scaling will have the same configurations as my initial TCP server instance (they will listen to port 1234, create multiple TCP sockets and compute Fibonacci).
2. Conguration of ASG: After selecting the launch template (that I created with the AMI), I named my ASG as ASG_Briikelda. Then I selected the VPC karan_test and the subnet subnet-2a759743 for AZ eu-north-1a. On the next step, I attached the load balancer I already createed (TCPServersTrafficBrikelda), I enabled the Health Checks, to check the state of my instances and when defined as unhealthy, to terminate and replace them. I enabled Monitoring because I want to see the data on CloudWatch and

I defined the size of my ASG. As desired and minimum capacity I put 1, because I want that always, at least one TCP server instance to be active to handle the traffic, and as maximum capacity 5, so that maximum 5 TCP server instances can be created during auto-scaling. For the Automatic scaling I selected Target tracking scaling policy, because I want to select a metric to define the behaviour of my ASG. This metric was CPU Utilization and according to this policy, if the CPU Utilization exceeds 60% with 3 datapoints for 3 minutes, the alarm will be triggered and the ASG will add 1 additional instance (TCP server). I set the instance warmup to 30s, meaning that the newly added instance will need 30 seconds to "warmup" before it can handle traffic. I put the value this low because I would need the new instance to take action asap. Whereas for the opposite situation, where the architecture needs to scale-in, the policy is that if the CPU Utilization is below 42% for 15 datapoints for 15 minutes, remove 1 instance.

I created the ASG_Brikelda and it automatically creates 1 instance from the Launch Template I provided, because I defined that my desired capacity is 1 (Fig.7).

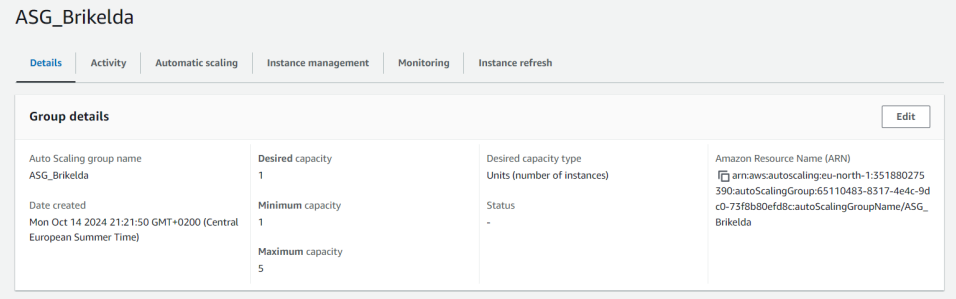

Figure 7: ASG_Brikelda Dashboard

However, to make things more interesting, and because I need low latency when my ASG scales-out, I decided to create a warm pool. Warm Pools decrease latency for applications that have long boot times and improve application performance (Source). In other words, by creating a warm pool, I can have instances to be in like "ready-mode", so when the ASG scales-out, it takes less time for the additional instance to be ready to handle traffic. There are 3 types of states that the EC2 instances can be in the warm pool: Stopped, Hibernated and Running. I selected them to be Stopped to minimize extra costs. I set my pool size to be 2. Fig.8 shows the warm pool I created. I also configured that after an instance is being eliminated from the ASG because of scaling-in, to go to the warm pool.

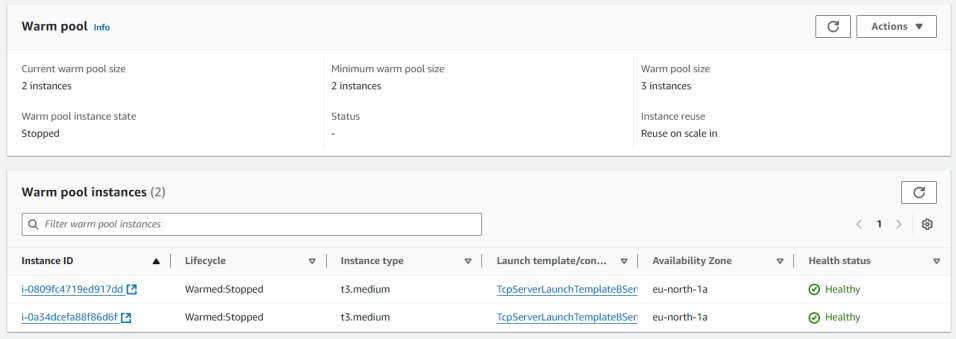

Figure 8: Warm Pool of ASG_Brikelda

Fig9 represents the diagram of the current AWS Extended Architecture, where 10K agents attack the systems, the traffic goes through the NLB, which it later distributes the load across the instances inside the ASG.

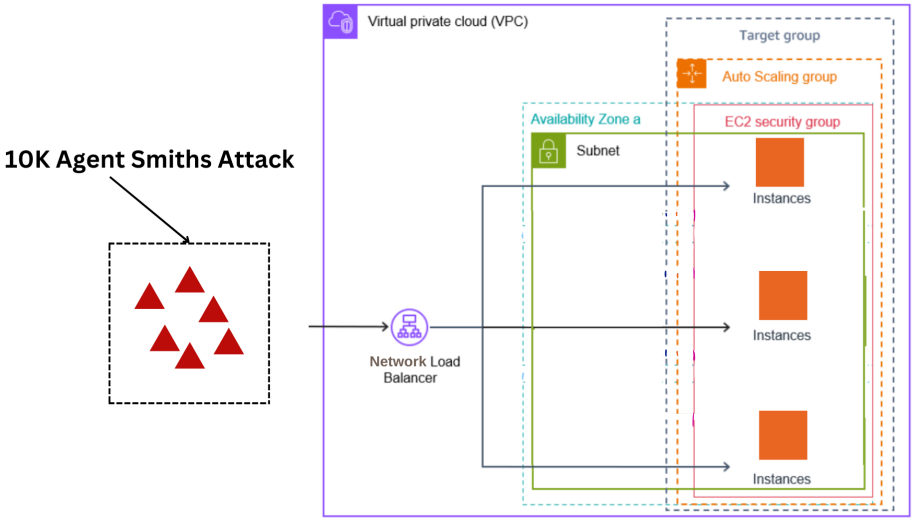

Figure 9: AWS Extended Architecture Diagram

The next step now is to execute this architecture. For this, I had first to upadte the script of AgentSmith.java and change the DNS name where the agents will attack, from the initial TCP Server Brikelda instance, to the DNS name of the NLB (NLBBrikelda-d385429d021da819.elb.eu-north-1.amazonaws.com). This is due to the fact that now the agents have to pass through the NLB first to go to the server(s). The NLB is the one that will balance the traffic across the servers. So, I edited that and started the attack with 10K agents, ticker interval of 1 second and attack duration 6 minutes (Fig.10). This time the attack should last longer, since it takes 3 minutes for the alarm to be triggered and to scale-out the ASG.

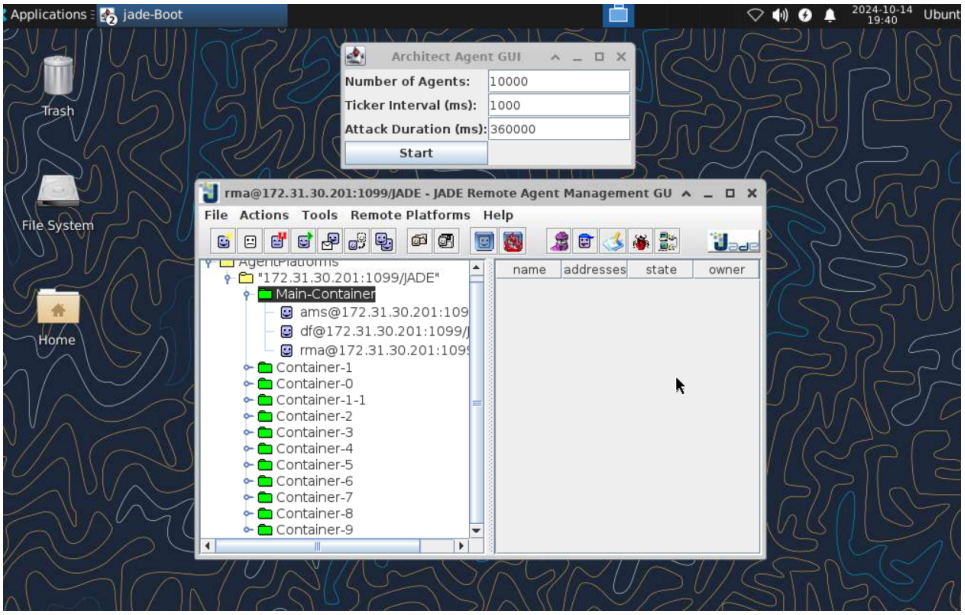

Figure 10: Attacking the Extended Architecture (NLB + ASG)

I start the attack at 21:37, and it is expected to increase the CPU Utilization of the ASG which initially contains only 1 server instance. Fig. shows the graph of the CPU Utilization for the ASG, triggered by the CloudWatch alarm (that was created when creating the ASG.)

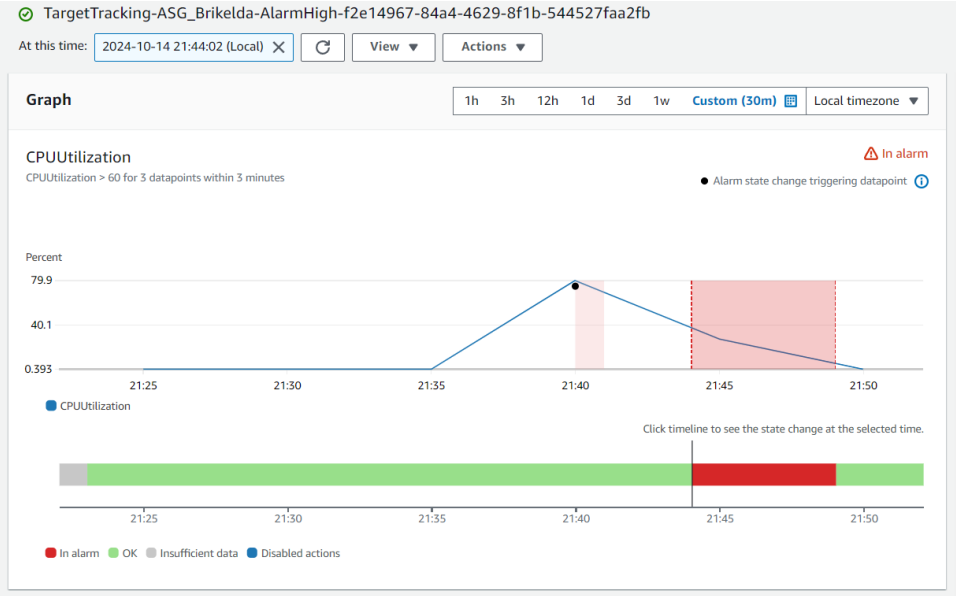

Figure 11: ASG CPU Utilization during and after the attack

Based on the graph, the alarm was triggered after 3 minutes, since the CPU Utilization should be above 60% for 3 minutes for the alarm to be triggered. After the alarm was triggered, the ASG took action and launched a new EC2 instance (TCP Server) from the warm pool. Because the CPU Utilization kept being above 60%, more instances were launched, since the attack was long and the servers needed more power. The curve starts to decrease as more instances are added to the ASG, since now all of them are handling the attack, distributed evenly by the NLB. The attack ended after 6 minutes, so at 21:53, where the load is already below 60%. Something that needs to be pointed out is that "The alarming datapoints can appear different to the metric line because of aggregation when displaying at a higher period or because of delayed data that arrived after the alarm was evaluated"-information point from CloudWatch. This means that CloudWatch may aggregate data into larger time periods (such as every minute or every 5 minutes), but alarms are evaluated based on more granular periods (per second or per minute). So, a spike that triggered the alarm might be "smoothed out" or not as obvious on the graph because of aggregation. Also, data might not always arrive at CloudWatch in real-time. Some data points might be delayed, meaning they arrive after the alarm has already been evaluated. This can lead to the graph displaying additional data after the fact that wasn't available at the time of alarm evaluation. For this reason we should not rely 100% on this graph, because as it can be seen, according to it, the alarm was triggered at around 21:44 (red line), which is not true, because the CPU Utilization by that time is already less than 60%. However, it was proved that as the ASG scales-out, the CPU Utilization decreases. Fig.12 shows the 5 instances created to support the traffic on the ASG and Fig.13 shows some monitoring data regarding the scaling-out process.

After the CPU Utilization goes below 42% for at least 15 minutes, the ASG starts scaling-in, which means it starts terminating instances that are no more needed. Fig.14 shows some activity notifications that the size of the ASG is going to its desired capacity.

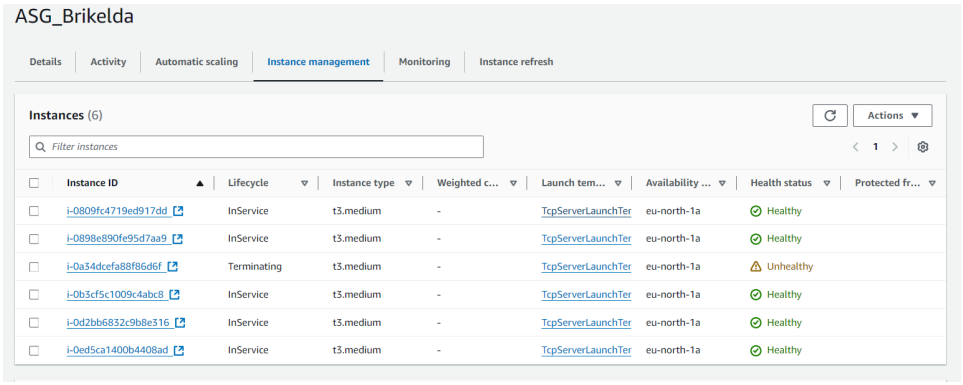

Figure 12: ASG Instances after scaling-out

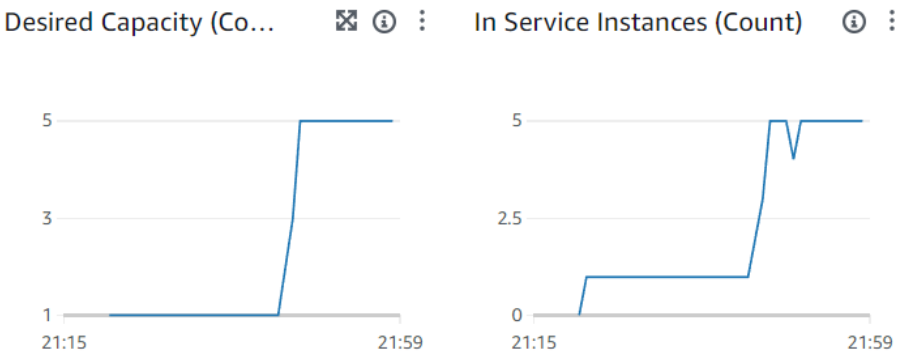

Figure 13: ASG Monitoring

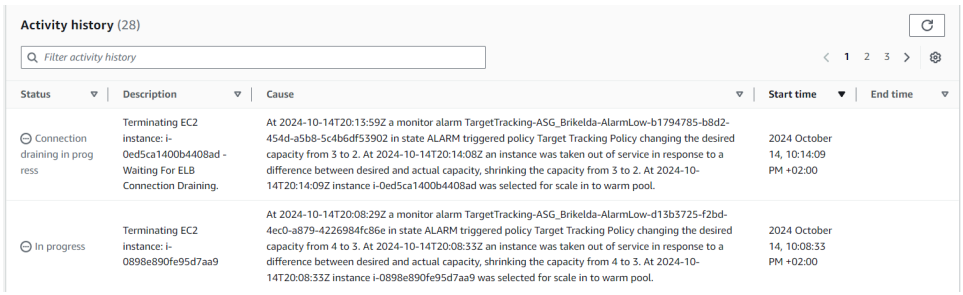

Figure 14: ASG scaling-in

Overall, we can say that the Extended Architecture helped managing the agent attacks by balancing the load to the TCP servers and decreasing CPU Utilization. The maximum value of CPU Utilization reached from the Extended Architecture was 79.9% and then it directly decreased, whereas for the scenarios without Load Balancer or Auto Scaling, it was above 90%, which made the server go down and refuse connections. It is definitely better to use Auto Scaling and Load Balancing when handling high traffic loads, as they improve traffic management and ensure continuous connectivity.

3 Appendix

The .csv files and Java scripts used for this report can be found on this GDrive link: files