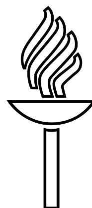
Kamil Janowski

CLOUD PLATFORM COMPARISON FOR MALWARE DEVELOPMENT



UNIVERSITY OF JYVÄSKYLÄ
FACULTY OF INFORMATION TECHNOLOGY
2019

ABSTRACT

Janowski, Kamil
Cloud Platform comparison for malware development
Jyväskylä: University of Jyväskylä, 2019, 64 pp.
Web intelligence and service engineering, Cyber Security, Master's Thesis
Supervisor(s): Khriyenko Oleksiy

The cloud platforms such as AWS, Google Cloud or Azure are designed to cover most popular cases in terms of web development. They provide services that make it easy to create a new user based on his email address, provide tools for inter-service communication, tools to manage the access rights of different users. Malware and botnet development however is more of a corner case, where the client application running on the victim's machine does not have an email address or a google account to authenticate itself and it does not run directly in the cloud, what can make it more difficult to manage the appropriate access rights. Also, the potential attacker may not want to write his own self-contained service, since, especially when managing a large number of clients, it might be much cheaper to run the backend serverlessly.

The big security companies always aim to lower the cost of development and maintenance of bots in order to provide their customers with their penetration expertise faster and cheaper.

The paper collects he data through the compilation of scientific publications regarding the botnet architecture and communication, as well as technical documentations regarding each of the cloud platforms discussed in the paper. Additionally proofs of concept are implemented for each of the proposed architecture in order to verify the validity of the approach, as well as measure the performance of the proposed solution and uncover hidden costs related to running the application in the cloud.

The following paper explores possible malware backend architectures for different cloud platforms, aiming to optimise the performance, minimize the development time while keeping the code easy to maintain and to minimize the execution cost.

After implementing proofs of concept for the standalone server-based CnC application as well as serverless running on GCP, AWS and Azure, it has been concluded that Azure is in fact the best platform for this sort of implementation due to simplicity of the architecture as well as ease of the implementation, while halving the execution costs compared to the standalone approach.

Keywords: malware, botnet, development, cloud, CnC, backend, serverless, cloud, Google Cloud, Azure

FIGURES

FIGURE 1: Gartner's Cloud Platform Market Shares in 2017 19
FIGURE 2: Standalone CnC - single instance design 22
FIGURE 3: Standalone CnC with load balancing 23
FIGURE 4: Resource consumption test sequence diagram 25
FIGURE 5: Standalone CnC memory consumption 25
FIGURE 6: Standalone CnC CPU usage 26
FIGURE 7: Standalone CnC client response times 27
FIGURE 8: AWS IAM 37
FIGURE 9: AWS IoT-based CnC design 43
FIGURE 10: AWS-based client response times 44
FIGURE 11 Azure-based CnC design 51

TABLES

TABLE 1 Single CnC instance costs 28
TABLE 2 Multi-instance CnC costs 28
TABLE 3 AWS IoT-based solution cost estimation 46
TABLE 4 Azure cost estimation 53
TABLE 5 Comparison of working solutions 55

TABLE OF CONTENTS

ABSTRACT
FIGURES
TABLES

1 INTRODUCTION ... 7
1.1 Research Problem ... 8
1.2 Research Objective ... 8
1.3 Research Question ... 9
1.4 Key Definition ... 9
1.4.1 Hacker ... 9
1.4.2 Botnet ... 10
1.4.3 Bot ... 10
1.4.4 Serverless computing ... 10
1.4.5 Cloud Computing ... 10
1.4.6 Malware ... 11
1.4.7 CnC server ... 11
1.4.8 DDoS attack ... 11
1.4.9 Serverless Framework ... 11
1.4.10 GCP ... 11
1.4.11 AWS ... 12
1.4.12 EC2 ... 12
1.5 Structure of the thesis ... 12

2 THEORETICAL BACKGROUND ... 13
2.1 Common botnet architectures ... 13
2.1.1 Centralised architecture ... 13
2.1.2 Peer to Peer (P2P) Architecture ... 14
2.1.3 Hybrid architecture ... 15
2.2 Common botnet use-cases ... 15
2.3 Command delivery methods ... 15
2.3.1 HTTP notifications ... 16
2.3.2 WebSocket notifications ... 16
2.3.3 IRC notifications ... 17
2.3.4 MQTT notifications ... 17

3 METHODOLOGY ... 17
3.1 Purpose of the study ... 17
3.2 Research approach ... 18
3.3 Research method ... 19
3.4 Data collection ... 20

4 FINDINGS – CASE STUDY ON 3 PLATFORMS 21
4.1 Standalone CnC server 21
4.1.1 Design 21
4.1.2 Resource consumption 24
4.1.3 Performance 27
4.1.4 Cost estimation 28
4.2 Google Cloud Platform-based approach 29
4.2.1 Serverless application engines 30
4.2.1.1 Google App Engine 30
4.2.1.2 Cloud functions 30
4.2.2 Authentication 30
4.2.3 Push notifications 32
4.2.4 Google Cloud Platform summary 34
4.3 AWS-based approach 35
4.3.1 Serverless applications 35
4.3.2 Authentication 35
4.3.3 Push Notifications 38
4.3.4 Design 42
4.3.5 Performance 44
4.3.6 Cost estimation 45
4.3.7 AWS Summary 46
4.4 Azure-based approach 47
4.4.1 Serverless applications 47
4.4.2 Push notifications and service-specific authentication and authorization 47
4.4.3 Design 50
4.4.4 Performance 51
4.4.5 Cost estimation 52
4.4.6 Development 53
4.4.7 Azure summary 54

5 CONCLUSION 54

1 INTRODUCTION

The popularity of computing clouds have increased drastically during the recent years. It is perfectly understandable, taken into account that renting the infrastructure from a cloud provider tends to be significantly cheaper than maintaining it inside the company. Things like the rental of the server room, electricity consumed by the servers, cooling of the server room and salaries of people responsible for the maintenance of the servers generate unnecessary overhead in terms of costs of maintenance, which can be drastically reduced when switching to the cloud, while in the same time providing higher availability and better monitoring of the hosted services. Furthermore the cloud providers constantly introduce new solutions allowing to reduce the maintenance costs even further. As we can read in "Serverless Computing: Economic and Architectural Impact" by Gajko Adzic and Robert Chatley (2017, p. 884):

> Amazon Web Services unveiled their 'Lambda' platform in late 2014. Since then, each of the major cloud computing infrastructure providers has released services supporting a similar style of deployment and operation, where rather than deploying and running monolithic services, or dedicated virtual machines, users are able to deploy individual functions, and pay only for the time that their code is actually executing. These technologies are gathered together under the marketing term 'serverless' and the providers suggest that they have the potential to significantly change how client/server applications are designed, developed and operated.

It is important to note however that those technologies are not only available to big corporations trying to lower their cost of server maintenance, but also to hobby software developers and black hat hackers.

A successful attacker may have thousands of devices under his control. In order to control such a large number of devices remotely a highly scalable Command-and-Control (CnC) server is required. Scaling up the virtual machines (VM) however can be costly, while having only a small number of administrators leads to a situation where most of the resources assigned to those VMs are seriously underutilized. While all the remote malware subscribes to the push notification service, it mostly just waits for a command to be generated

by an administrator. Effectively, while our CnC server has to be scalable in order to maintain the connection to numerous clients, it requires fairly low computing power until an administrator decides to generate certain load. This suggests that the serverless approach could be applied in this case, what could potentially not only save the attacker a lot of money, but also make such a large scale attack possible in the first place.

1.1 Research Problem

There are many various cloud providers out there. While they all provide services allowing to easily and quickly build secure web applications, the problem of building a CnC server is more of a corner case, that is not necessarily properly addressed by certain clouds. This might yield it impossible to implement such an application in a serverless manner at all, or require to make some compromises and implement workarounds for services that work in a different manner than desired.

The problem is important to address as those are not only the "black hat hackers" that seek to lower the cost of their attacks. There are various data security companies that are frequently requested to perform attacks on their customers in order to verify the security of their application or network infrastructure. Similarly, many "white hat hackers" work as freelancers. For those in particular lowering the cost of implementation and maintenance of the CnC server, might determine if they're going to make any income at all.

1.2 Research Objective

The main objective of the research is to find a way to use the cloud as a CnC server without implementing any application that requires a constantly running server in a Virtual Machine, as those are the main cost generators of the web applications. For this reason we are going to investigate the serverless solutions provided by various cloud platforms as well as other services that come with

specific clouds that could potentially allow us to set up the communication between the backend and the client application, enable the file transfer, make it easy to manage the access rights of different clients as well as enable the client management in as a whole. We are also going to take a closer look at how the continuous deployment can be solved in various cloud systems.

Each of the approaches will be backed up by a small Prove of Concept (POC) if possible at all. In order to optimise the development time and ensure multi-platform and multi-cloud support of at least parts of our code, all solutions will be implemented with Node.js.

1.3 Research Question

When focusing on various cloud platforms, such as Amazon Web Services (AWS), Google Cloud Platform (GCP) and Azure the approach to the problem of CnC application development might be completely different and the cost of execution may different significantly as well. The question in this case is, which one of the platforms is the best suited and the cheapest to run our CnC application.

1.4 Key Definition

1.4.1 Hacker

Hacker is an attacker attempting to access resources of a remote machine. In this thesis the term "hacker" will be used to describe the administrator of the CnC server and in the same time the administrator of the botnet.

There are 2 types of hackers, commonly referred to as:

- the "white hat hackers" – usually a hired penetration tester who rather than harming the victim, points out the security vulnerabilities his customer faces
- the "black hat hacker" – an attacker with malicious intent

1.4.2 Botnet

A botnet is a network of private computers infected with malicious software and controlled as a group without the owners' knowledge, e.g. to send spam.

1.4.3 Bot

A bot in this case is a single client application executing (and in some architectures issuing) the commands on the infected device.

1.4.4 Serverless computing

The “serverless” computing is a marketing term that relates to developing single functions, rather than a large monolithic application and then being charged only for the actual execution time of the function, rather than for the constantly running server that technically is still there, but is hidden from the service user. The concept was originally introduced by Amazon in their AWS cloud in 2014 under the name of Lambda. Since then all major cloud providers introduced various equivalents in their solutions. As many instances of lambda can be triggered in parallel, this solutions is not only cheaper to execute, but also potentially infinitely scalable. This is why it’s commonly used for a wide range of applications, starting with REST API call processing and ending with Big Data event handling.

1.4.5 Cloud Computing

As Amazon defines it[1]:

> Cloud computing is the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the internet with pay-as-you-go pricing.

[1] https://aws.amazon.com/what-is-cloud-computing (24-06-2018)

1.4.6 Malware

Malware, or malicious software, is any program or file that is harmful to a computer user. Malware includes computer viruses, worms, Trojan horses and spyware. These malicious programs can perform a variety of functions, including stealing, encrypting or deleting sensitive data, altering or hijacking core computing functions and monitoring users' computer activity without their permission.

1.4.7 CnC server

In "Survey on botnet: its architecture, detection, prevention and migration" by Ihsan Ullah et al. (2013) the CnC servers are defined as centralised servers allowing the malicious attacker to remotely control a number of clients applications that connect to it.

1.4.8 DDoS attack

DDoS stands for Distributed Denial of Service. It's one of the common use-cases of a botnet, where a number of bots are instructed to simultaneously send requests to a specific server, yielding the server inaccessible for other users.

1.4.9 Serverless Framework

A popular framework allowing to easily specify the configuration, deployment process and debugging process of serverless applications, while supporting a large variety of different cloud providers.

1.4.10 GCP

GCP is short for Google Cloud Platform. It's one of the platforms which will be discussed in this paper.

1.4.11 AWS

AWS is short for Amazon Web Services. It's one of the cloud platforms which will be discussed in this paper.

1.4.12 EC2

EC2 is a short for Elastic Compute Cloud. It's one of the services provided by the Amazon platform. It allows to create a number of Virtual Private Servers and/or Virtual Machines that can run the application of your choosing.

1.5 Structure of the thesis

In the beginning the thesis focuses on the theoretical background, allowing us to better understand how botnets are designed and how the communication between the Command & Control application and the bots is handled.

Next, in order to get a better understanding of the required implementation effort as well as related costs, we design the standalone platform-independent CnC application and run the performance measurements. Once that part is handled, we can easily compare this solution to some based on various cloud-based serverless solutions. In the next 3 chapters we investigate the serverless services, authentication methods and various command delivery methods provided by the Google Cloud Platform, AWS and Azure. We propose architectures for each of the platforms, implement proof of concepts, run performance measurements and estimate the costs.

Finally in the end we compile all the results in order to determine which of the cloud platforms appears the most suitable for the development of serverless Command & Control applications.

2 THEORETICAL BACKGROUND

2.1 Common botnet architectures

As we can read from "Survey on botnet: its architecture, detection, prevention and migration" by Ihsan Ullah et al. (2013, p. 661-662), as well as "Botnet Communication Patterns" by Gernot Vormayr et al. (2017, p. 2772) there's a number of different architectures that can be developed depending on the attacker's needs.

2.1.1 Centralised architecture

The architecture assumes that there's one CnC server that all the clients can connect to. It tends to use either Internet Relay Chat (IRC) or HTTP as the communication protocol. This solution tends to be the most commonly seen due to the ease of implementation as well as high efficiency. The main drawback of the approach is that it is fairly easy to detect. Each of the clients of the botnet needs to have a hard-coded address of the server that it is going to communicate with. Effectively simply editing the byte code of the application (or decompiling it, if possible) allow you to quickly read the address of the CnC server and then block all the traffic to it. The address can also be seen through network sniffing. This problem however can be mitigated through the use of Domain Generator Algorithms (DGA).

DGAs generate different domain names based on a changing input. For instance, a different domain could be used based on the current time. This then requires all clients to have synchronized time down to one hour. While relying on the system time might not necessarily be a good idea, as the system time largely depends on the user-specified settings, it can be easily achieved by polling popular websites that contain such information.

2.1.2 Peer to Peer (P2P) Architecture

The approach allows to hide most of the network traffic by introducing the supervisor-bot, who becomes responsible for delivering the command to other clients, who later on can forward the command even further. While the source of the command becomes fairly difficult to detect in this case, the actual delivery as well as the delivery of the result takes significantly more time than in the centralised architecture. This makes such botnet difficult for the attacker to manage. Also, it is important to note that the architecture is prone to the Sybil attack, where the attacker subverts the reputation system of a P2P network by creating a large number of pseudonymous identities, using them to gain a disproportionately large influence.

In a fully meshed botnet every client is linked to every client. This way it is possible to reduce the latency as well as ensure that the removal of any number of bots does not disrupt the communication. This solution however is not scalable due to the number of required connections in larger botnet. Additionally, the larger number of connections increases the visibility of the botnet. Also adding or removing a single client generates a significant network traffic as all other clients have to register the information about the new bot.

The topology unfortunately is difficult to implement due to the challenges of finding the initial peers and reliably distributing commands to every bot.

The list of peers can be hard-coded directly in the executable or provided by a cache server. The first solution however can work only in a very targeted attack and should the botnet be detected, the list can be easily extracted from the code. In the second case, the server is visible to the public internet and that brings back all the issues related to the centralised architecture.

Finally, depending on the NAT configuration, not every computer has direct access to the internet making it difficult to access from external network.

2.1.3 Hybrid architecture

Hybrid architecture combines both centralised architecture and the P2P one. Instead of bots connecting directly to the CnC server, an additional proxy layer consisting of bots connected in a P2P topology is added. Determining whether a certain bot should behave only as proxy or P2P accessed worker can be done based on the connectivity properties (such as when some of the infected devices don't have the direct access to the CnC server). In order to lower the probability of detection of the CnC server, additional layers of P2P connection can be added, although that comes with the cost of increased latency.

2.2 Common botnet use-cases

According to the definition of botnet provided by Norton[2] a botnet can be used for purposes like:

- Executing a DDoS attacks
- Emailing spam to millions of internet users
- Generating fake Internet traffic on a third-party website for financial gain.
- Replacing banner ads in your web browser specifically targeted at you.
- Pop-ups ads designed to get you to pay for the removal of the botnet through a phony anti-spyware package.

2.3 Command delivery methods

There's a number of different ways that a command can be delivered by CnC server to a bot. As already mentioned in the introduction, HTTP and IRC protocols are the most commonly used for this purpose, however those are not our only options. As mentioned by Inmaculada Ayala et al. in "An empirical study of power consumption of Web-based communications in mobile phones" (2017)

[2] https://us.norton.com/internetsecurity-malware-what-is-a-botnet.html (07-07-2018)

WebSockets are also a common option for the message delivery both in case of mobile applications as well as websites (and effectively botnet client). Also what is available in most clouds are IoT services that can enable the communication with a remote client over the MQTT protocol. Let's take a closer look at each one of these approaches now

2.3.1 HTTP notifications

As mentioned by Inmaculada Ayala et al. the command delivery over the HTTP protocol can be handled in two different ways: polling and long polling.
Inmaculada Ayala et al. defines the polling approach in the following way:

> The polling mechanism is the simplest way to receive asynchronous data. The client polls the server periodically (polling interval) for new content by sending HTTP requests, allowing the server to respond with an HTTP response if new data is available. Each request attempts to pull any available data. If no data is available, the server returns an empty response and the client waits for some time (polling interval) before sending another (poll) HTTP request.

Whereas the long polling is defined as follows:

> In order to alleviate client continuous polling, there exist different web models in which a longheld HTTP request allows a web server to push data to a browser only when new data is available. One of the most common server push mechanisms is HTTP "Long Polling", in which the server "holds open" (not immediately reply to) each HTTP request, responding only when there is new data to deliver. Then, there is always a pending request to which the server can reply for the purpose of sending data as it is available, thereby minimizing the latency in message delivery, and the use of processing/network resources.

2.3.2 WebSocket notifications

Inmaculada Ayala et al. describes also the WebSocket-based approach to the problem. With WebSocket protocol it is possible for the client to create a full-duplex persistent TCP connection to the server.

> Based on this connection, the Web server is able to actively send data to the client whenever it is available. Prior to data/message exchange, the WebSocket protocol requires an initial handshake and the message exchange. The initial handshake uses the HTTPUpgrade-request, which allows to switch from the HTTP to the WebSocket protocol. The message exchange is executed in form of frames, which contain either text or binary data.

2.3.3 IRC notifications

IRC protocol is a simple plain text protocol operating over a persistent TCP connection. Effectively, similarly to the WebSocket approach, the message is delivered to the client as soon as it is available on the server.

2.3.4 MQTT notifications

As Konglong Tang et al. Define the MQTT protocol in " Design and Implementation of Push Notification System Based on the MQTT Protocol" (2013), it's a protocol originally designed and developed by IBM, that allows the delivery of push messages. MQTT can work in one of three modes of message delivery:

- At most once – the actual delivery depends only on the TCP connection and as a result some messages can be lost on the way
- At least once – the server ensures that the message is delivered, but duplicates can happen
- Only once – the server ensures that the message is delivered exactly one time

It is a particularly interesting protocol in our case, as the MQTT push notification service is provided by every major cloud through IoT services.

3 METHODOLOGY

This part describes the methodology used for the study. It will go through the full approach of conducting the study, the data collection methods, the research methods and the purpose of the study.

3.1 Purpose of the study

The purpose is exploratory. The exploratory research, as the name already implies, aims to explore the research questions rather than provide the ultimate

solution to the problem. This is important in this case, as there is are hundreds of ways to implement a malware. It simply wouldn't be feasible to go through them all to find the one best solution.

The paper will compare various cloud platforms, services they provide and the cost of their usage to find various possible architectures for our Command and Control server and effectively the malware communicating with it. In the end we will also compare the cost of maintenance of different architectural approaches. After all the very reason why designing a Cloud Platform-specific CnC server makes sense is because it can drastically lower the execution costs.

It is also worth mentioning here that malware development is not among any of the target applications of those cloud platforms. While they provide a number of very convenient features useful for building robust web applications, management of IoT devices and AI data processing, services that might turn out to be essential to achieve our goals might simply not be in place. Should that happen, the only way to execute our solution is to spawn a virtual machine inside that cloud, running a standalone CnC server, what defeats the purpose of using that specific cloud.

3.2 Research approach

In the paper we will use the deductive approach. When utilizing the deductive research approach we want to start with a hypothesis and the through data collection we want to build a proven theory. In this case our hypothesis is that it is possible to build a CnC solution using only the serverless technologies provided by various cloud platforms and therefore minimize the cost of execution of the CnC application while keeping it scalable, what is necessary to manage a large number of clients.

3.3 Research method

We will use a combination of the qualitative case study as well as the exploratory research method. The qualitative case study method is used to collect the data through in-depth investigation of multiple cases within one context.
The study will focus on three cases of three different clouds:

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure

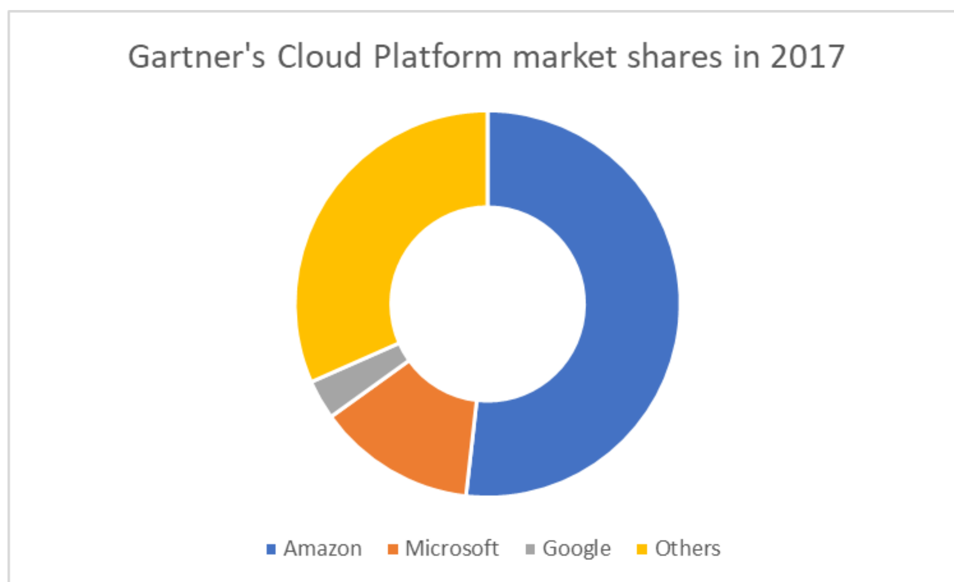


FIGURE 1: Gartner's Cloud Platform Market Shares in 2017

As we can see in the Gartner's report from the year 2017[3], these three have some of the largest market shares. Each one of these clouds support serverless computing in one way or another, whether those are lambdas, Google App Engine or other form of serverless logic executor and after all those are the services this study puts a lot of emphasis on. They all also provide various ways of message delivery through various custom push notification services to HTTP and MQTT-based IoT services. Some of them also provide other services that can allow us to make our malware more effective (like for instance the P2P services).

[3] https://www.gartner.com/newsroom/id/3884500, 26.12.2018

The second part of the study is exploratory. There has been very little research done related to building cloud-based serverless Command & Control applications. Most of the articles available on the topic focus on more traditional approaches where the standalone server is required. This is why we need to explore our options, propose completely new architectures and prove that they are feasible to implement. This is why minimalistic implementations of each of the proposed architectures will be provided, tested and discussed in more details.

3.4 Data collection

In exploratory case studies data often is collected through questionnaires, interviews and experiments. While the questionnaires and interviews make very little sense in terms of technology-related studies, the experiments do.

In the study we will collect the data through:

- Already existing research papers, official cloud documentations and blogs on related topics. Especially the blogs may prove to be very useful as most framework and technology providers as well as data security companies tend to describe on their blogs various approaches to various problems related to architecture, implementation and security threats.
- Empirical implementation, to validate that the approach is actually feasible. In the software development it is a very common case that a certain technology appears to solve the proposed problem, whereas during the implementation of the solution it turns out that the selected technology imposes certain limitations, yielding it inapplicable for the specific problem. Effectively the only way of ensuring that the solutions we will propose in this study are valid is to implement the proof

of concept for each one of them. Additionally the POC can give us information about performance of the proposed solution, point the hidden costs and show how much development effort is actually needed to implement the solution in the first place.

4 FINDINGS – CASE STUDY ON 3 PLATFORMS

4.1 Standalone CnC server

In order to better understand the complexity of CnC applications as well as evaluate the cost of their execution, let's first analyse the standalone approach where we try to create our own CnC application running on a server. Let us however not focus on any extreme examples just to prove the point on the thesis. Technically we could create a Java application running on a Tomcat server, but according to Oracle documentation[4] we would need 512 MB of memory just to run the server and then there are memory requirements of our application on top of that. For this reason we're going to build a small application in Node.js instead. One that can integrate the whole server in it, without relying on a third party one.

4.1.1 Design

While the list of common use cases of botnet is fairly long, most of them can be handled in a similar way:

[4] https://docs.oracle.com/cd/E13169_01/ales/docs22/installadmin/prepare.html, 07.08.2018

1. Client subscribes for the push notifications from the server over HTTP or IRC protocol (as already mentioned before)
2. A request is issued by the administrator to the server
3. The server dispatches appropriate commands to the client

Effectively the most trivial CnC application could be essentially just one server with all the clients connecting to it and waiting for the attacker to issue a command (FIGURE 2).

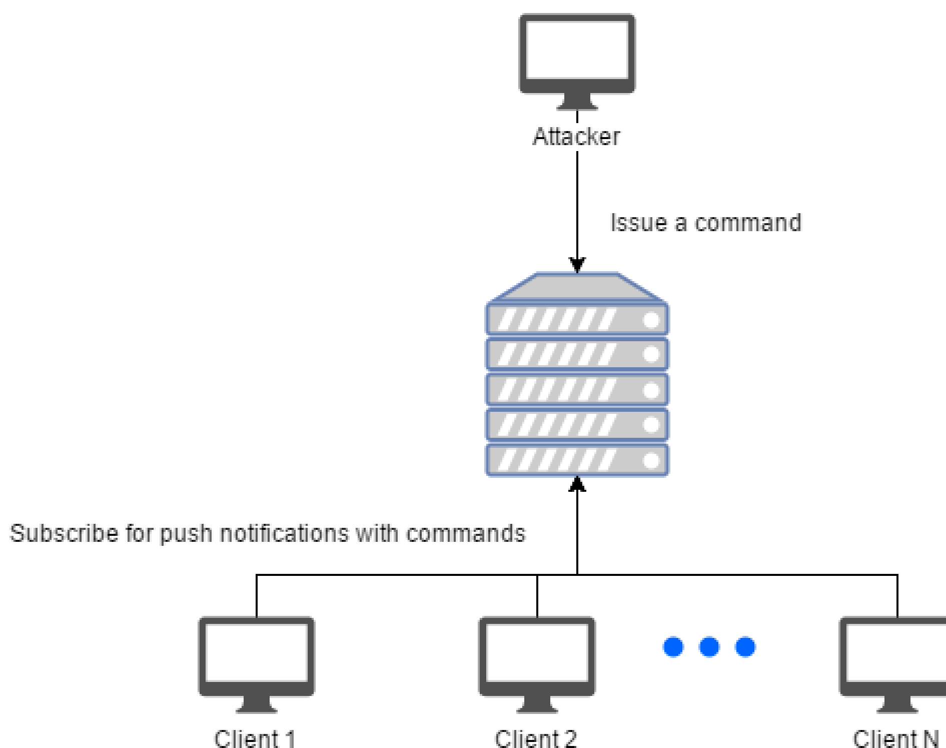


FIGURE 2: Standalone CnC - single instance design

This approach however has a drawback – there's only so many clients that can connect to the server in the same time. They all have to maintain an open TCP connection in order to be able to react to the command as soon as possible and once some data has to be transferred between the client and the server, there's also a limit imposed by the connection speed of the virtual machine running our server application. We can obviously always configure the virtual machine giving it higher bandwidth, but then we would end up paying for it at all times, even when we don't really use it. The same goes for all the other resources re-

quired to run the application. With just one server we cannot have green-blue deployments. Also, single server is more error-prone. Should anything happen to it, the entire CnC will go offline. For this reason it seems more reasonable to have a number of VMs with lower amount of resources, that can be spawned automatically by a load balancer when they're needed. This however introduces a difficulty. If there are multiple servers hidden behind a load balancer, then they need to be able to exchange the information about the connected clients between each other. Luckily there are multiple caching services out there that can be used for this purpose. One of the most popular ones and provided out of the box by most major cloud providers is Redis. Having that in mind, let's update the application design (FIGURE 3).

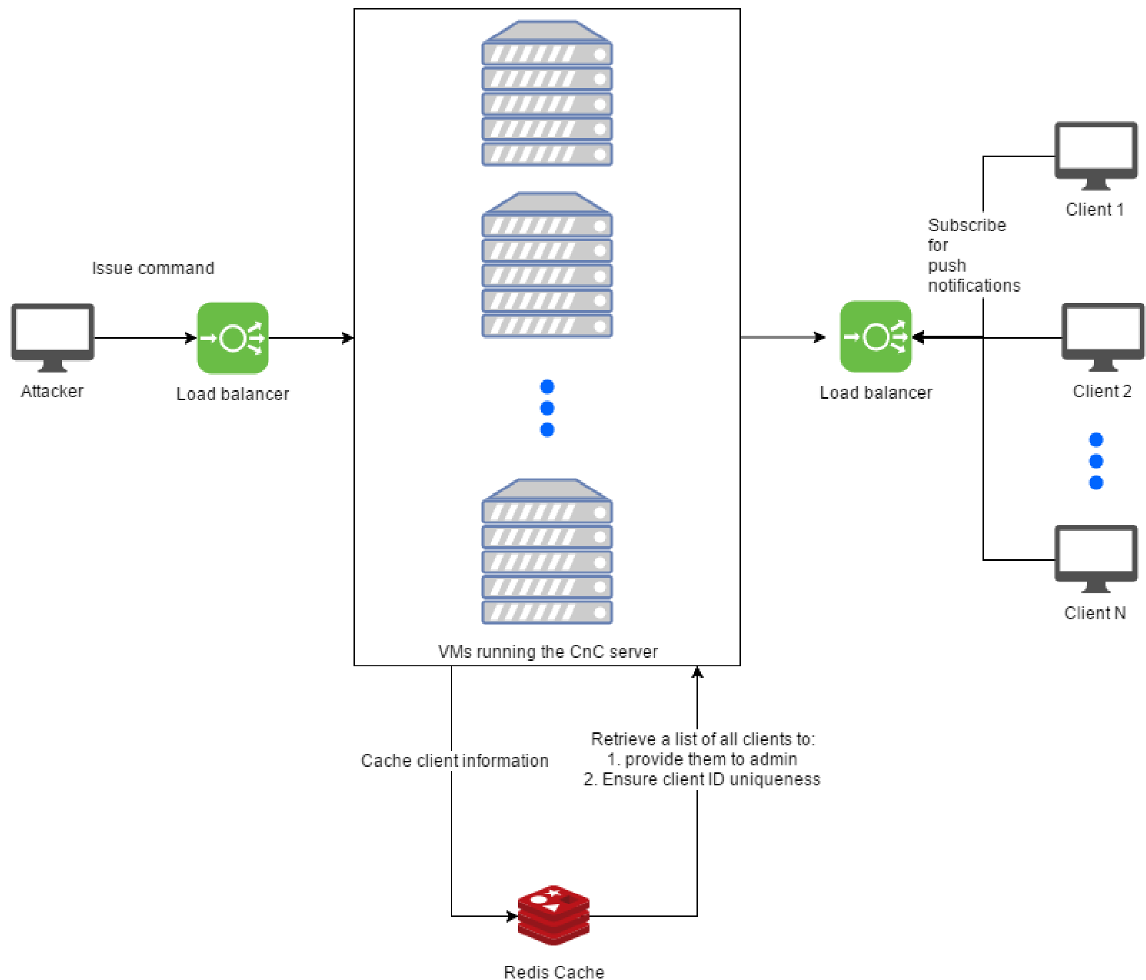


FIGURE 3: Standalone CnC with load balancing

This way we limited the cost of VMs required to run our CnC application, however in the same time we introduced the necessity of using the load balancer and the Redis cache, which do not come for free either.
In the next sections let's try to evaluate how much resources are needed in both approaches in order to calculate the approximate cost of execution of the server-based CnC application.

4.1.2 Resource consumption

In order to evaluate the resources actually needed to execute I wrote a minimalistic proof of concept in Node.js that can work either with or without the Redis support. The implementation details can be looked up from appendix 1.
In order to evaluate the required resources, I will simulate 10000 client connections, issue a command to every bot and measure the memory consumption and the processor usage of the CnC application process. The detailed description of how the test is executed is depicted in FIGURE 4, but the implementation details can be looked up from Appendix 2.

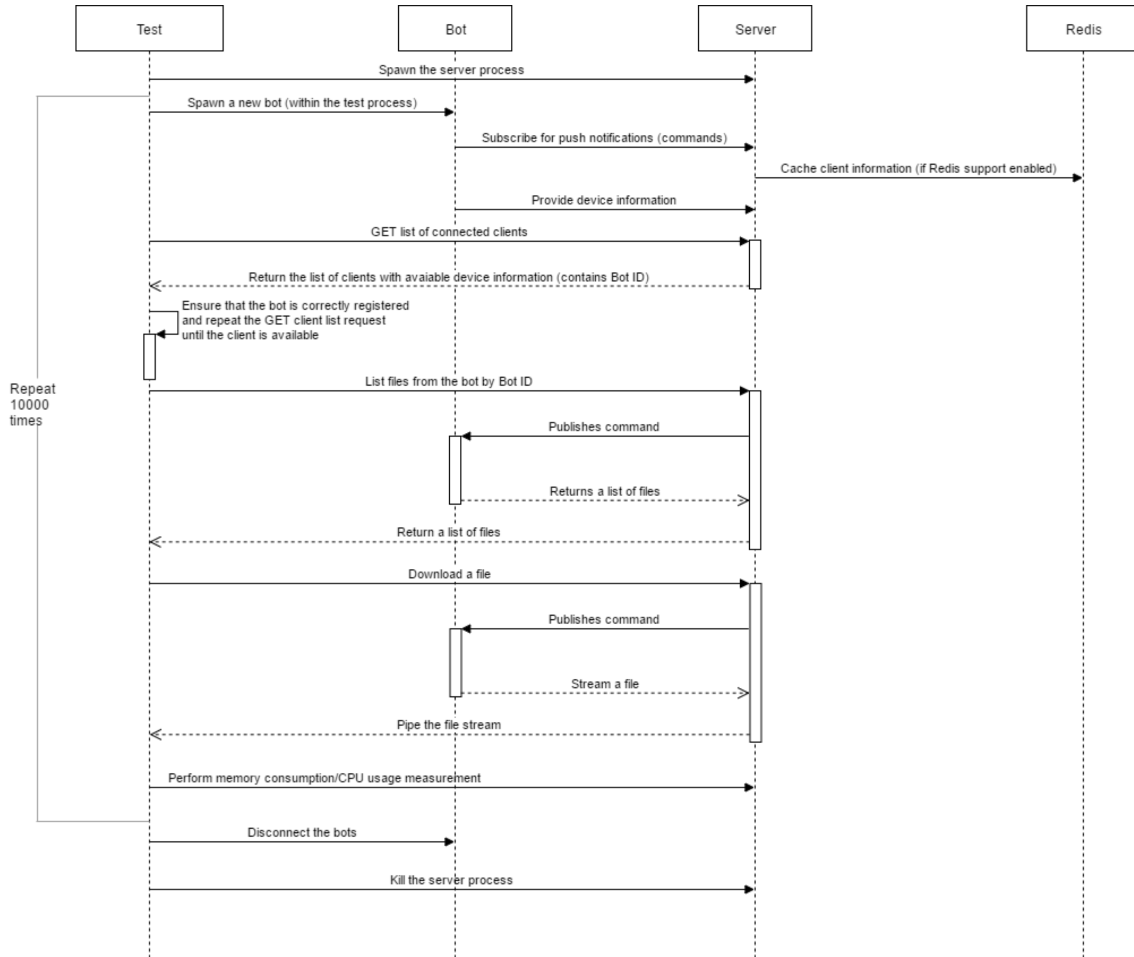


FIGURE 4: Resource consumption test sequence diagram

Following the testing method depicted in FIGURE 4 a number of results were retrieved. FIGURE 5 and FIGURE 6 depict the resources that were consumed by the server process during the measurement.

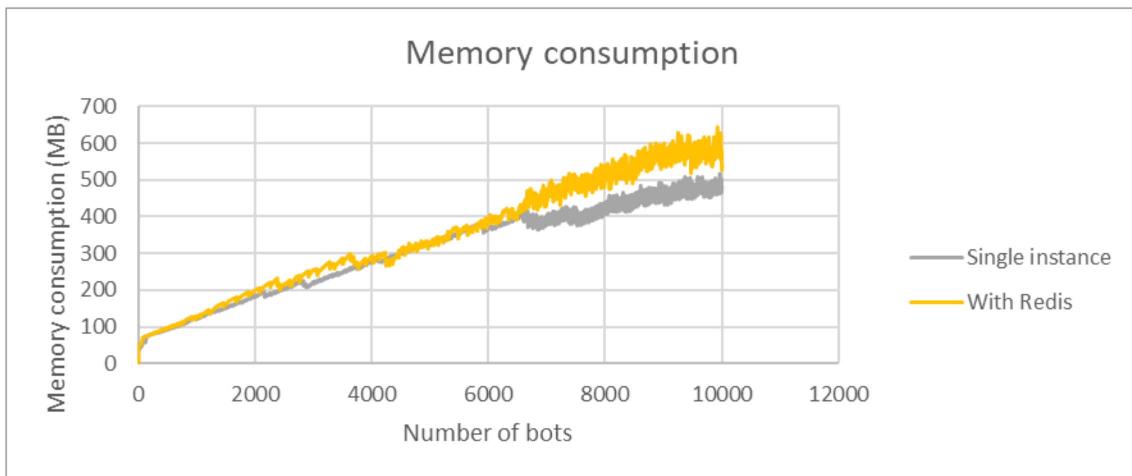


FIGURE 5: Standalone CnC memory consumption

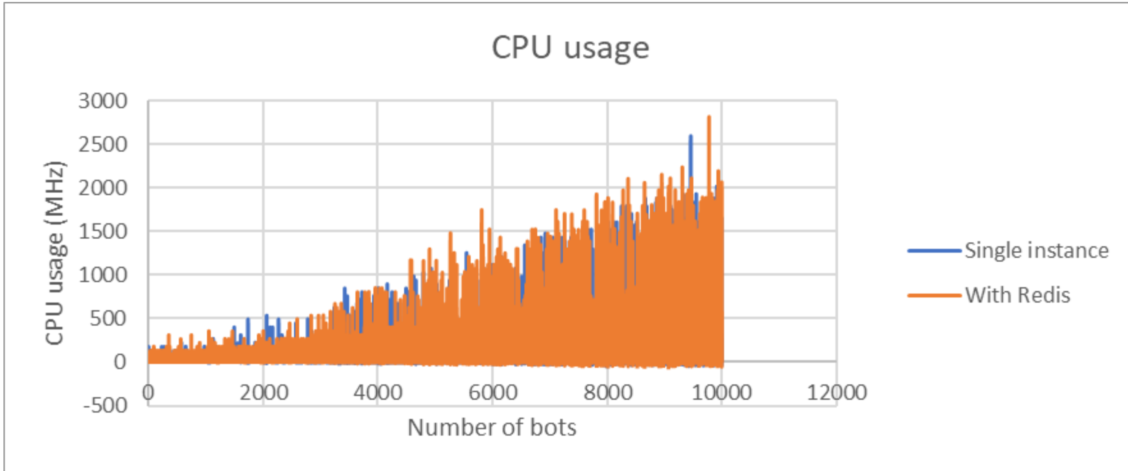


FIGURE 6: Standalone CnC CPU usage

As can be seen from FIGURE 5, the memory consumption when using the external caching system is slightly higher. This is understandable as in that case an additional library has to be initialized to enable the Redis support in the first place. Also the caching process itself requires a little bit of memory that is going to be released by the garbage collector only after a while. As a matter of fact we can see from the graph that the further we go, the more irregular measurements become, adding up to +/- 100MB delta between the lowest and highest measurement. This indicates that the garbage collector tries to free the memory from no longer necessary data. The detailed numbers of the measurement can be looked up from the Appendix 3.

The FIGURE 6 which depicts the CPU usage is more sparse. The built-in Node.js tool allowing to measure the resources used by a specific process returns the percentage of CPU that is used at a certain time. The measurement has been performed on a device with 2 core processor with 2.8GHz/core. As the CnC server is not performing any calculations at all times, many of the measurements return 0% CPU usage what in this case only indicates lower than ~28MHz usage. The detailed numerical results can be looked up from Appendix 3.

4.1.3 Performance

The standalone approach, as opposed to other ones that will be discussed later in this paper, does not require any internal network calls, apart from the one to Redis (if enabled). This means that by definition this approach should provide us with quicker response times. Let us however spend a moment to measure the response times between the CnC server and the client in order to see how much the latency changes from one approach to another.

In this test, in order to avoid the bias coming from the network latencies, we will actually deploy our server to a remote host. In this case we will use the an AWS EC2 server in eu-west-1 region. This means that the server is physically located in Ireland. We will simulate one client connecting to the CnC server and then measure the response time for issuing 1000 directory listing commands.

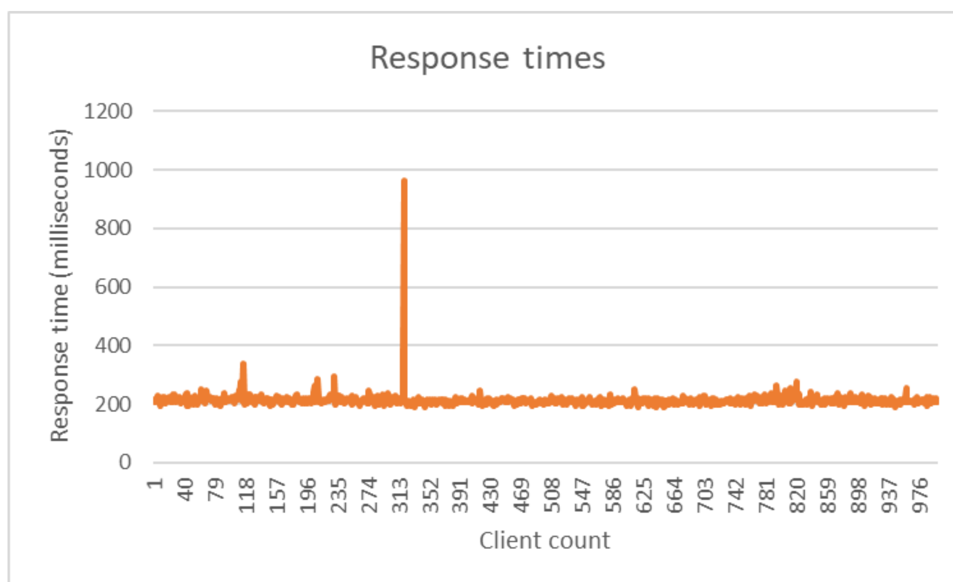


FIGURE 7: Standalone CnC client response times

As we can see in FIGURE 7, the responses, apart from a few exceptions tend to be fairly quick. The median response time is equal to 212 milliseconds.

4.1.4 Cost estimation

The cost of execution can vary slightly depending on the server provider that we would like to use. However, assuming that different providers use similar price lists in order to stay competitive, we'll perform the calculation based on the prices of only one of these providers – Amazon.

Amazon provides an easy to use price calculator that can be used to calculate the price of AWS services usage. During the price calculation we're going to focus only on the services that are specific to the standalone approach. So we're going to skip the cost of S3 bucket that could be used for providing the client updates, or Route 53 for generating the DNS domain, as these will also be needed in case of serverless approach and we're only interested in the cost difference between the standalone application and the serverless one in this case. Also, the prices can differ vastly between different regions. For the sake of clarity we're going to use the prices for the region eu-west-1 (Ireland).

TABLE 1 Single CnC instance costs

Single instance			
Service	Details	Why it is needed	Price (USD)
EC2	t2.small instance with external IP address	t2.small instance provides 2GB of memory. To provide for our 10000 bots this is all we need and a little bit more	$21.96
Data transfer IN	100GB	The data transferred from the admin to the server as well as the responses generated by bots and bot registration	$0.00
Data transfer OUT	100GB	The data transferred to bots and response for the administrator	$8.91
			$30.87

TABLE 2 Multi-instance CnC costs

Multi-instance			
Service	Details	Why it is needed	Price (USD)

EC2	t2.nano instance with external IP address	t2.nano instance provides 0.5GB of memory which is sufficient for lower number of clients	$8.28
Data transfer IN	100GB	The data transferred from the admin to the server as well as the responses generated by bots and bot registration	$0.00
Data transfer OUT	100GB	The data transferred to bots and response for the administrator	$8.91
Redis cache			$13.18
Load balancing			$22.10
			$52.47

It's important to notice that in the multi-instance approach the EC2 instance cost is somewhat variable. The presented cost is for one virtual machine, but more can be spawned by the load balancer at any time, should that be needed, so that all clients can be managed efficiently.

Additionally, the presented costs are per region. This means that should the administrator decide to deploy the application to multiple regions in hopes to minimise the latency, the final price should be multiplied by the number of regions in use.

4.2 Google Cloud Platform-based approach

Google Cloud Platform is a very convenient platform allowing the developers to easily manage their web applications in the cloud environment. As we can read in google documentation and marketing materials[5], there are basically 2 ways to approach the problem of serverless development on Google Cloud Platform:

- Applications running on App Engine
- Cloud Functions

[5] https://cloud.google.com/serverless, https://cloud.google.com/functions/docs/ and https://cloud.google.com/appengine/docs/, 17.03.2019

Let's take a closer look at them.

4.2.1 Serverless application engines

4.2.1.1 Google App Engine

Google App Engine is advertised as a fully managed serverless application platform, allowing you to deploy applications written in a number of popular programming languages including among many Go, Java, JavaScript and Python.
It comes with a number of monitoring features, requires close to zero configuration and allows easy deployments. The business logic execution on google app engine can be triggered by either an HTTP request or a CRON scheduler.
It also provides us with Memcache, which can be extremely useful for storing the state of a distributed application, as well as various permanent data stores.

4.2.1.2 Cloud functions

Cloud functions as of now are still in the beta version. Their support is greatly limited compared to Google App Engine, as they don't have very little monitoring or external service integrations that comes out of the box. They're designed to be triggered by any of the following:

- HTTP request
- Cloud Storage event
- Pub/Sub notification

What they however lack in supportability, they make up with portability. They're fully supported by the Serverless framework and that allows the developers to easily switch from their previous cloud provider to GCP without having to re-implement their application from scratch.

4.2.2 Authentication

When building a CnC application, in order to avoid a situation in which another hacker or a security engineer tries to access the data that is meant for another bot, or simply access the services provided by the CnC application despite not

being able to properly authenticate itself, it is important to use a proper way of authentication of bots. As a result each bot needs to have some form of unique credentials that will uniquely identify it in the botnet as well as ensure the explicit access to its own resources.

As we can read in Google Cloud documentation[6] in GCP there's a number of ways an application can authenticate itself, starting with acquiring webservice credentials, going through standard user authentication and ending with authentication functionalities provided by the IoT service.

1. Service authentication – a special account that represents an application as opposed to representing a user. You can use a service account by providing its private key to your application, or by using the built-in service accounts available when running on Google Cloud Functions, Google App Engine, Google Compute Engine, or Google Kubernetes Engine.
2. User accounts - you can authenticate users directly to your application, when the application needs to access resources on behalf of an end user. Example use cases include:
   - Your application needs to access Google BigQuery datasets that are in projects owned by users of your application.
   - Your application uses an API such as the Cloud Resource Manager API, which can create and manage projects owned by a specific user. The application would need to authenticate as a user to create projects on their behalf.
   - You plan to create development tools that create resources within projects.
3. An API key is a simple encrypted string that identifies a Google project for quota and billing purposes. API keys can be used when calling Google APIs that don't require authentication, and when using Google Cloud Endpoints.

[6] https://cloud.google.com/docs/authentication

After deeper investigation however it turns out that each one of these authentication methods have certain limitations that would make them difficult to use in case of our application. Service authentication credentials cannot be generated through provided SDK, but instead have to be manually delivered to the application. That would force us to either use the same credentials in all bots (what defeats the purpose of authentication in the first place) or manually create a set of credentials for each bot and then somehow deliver it remotely (not really feasible). The user authentication requires a real google account. This means that every bot would need to have a dedicated mailbox in order to be able to log into the system. And finally the API Key, although the easiest to use, is greatly limited in terms of what it can be used for. In particular, no push notification system provided by google can be accessed using the API Key.

What seems the most important to us are the push notifications though since only they can deliver a remote command that should be executed on victim's device and there are several different services in Google Cloud that allow us to deliver those. Some of them also introduce additional service-specific methods of authentication.

4.2.3 Push notifications

As mentioned before, Google Cloud Platform provides a number of different ways to deliver the remote command.

1. Pub/Sub service – the name suggests that this is specifically what we're looking for. After all we want our client to SUBscribe to a certain feed and then PUBlish the remote commands into it. Unfortunately, when trying to take it into use we find multiple issues with the service that yield it unsuitable for our use case:
   - It is originally designed to serve the noticiations to GCP-hosted applications. This can be worked around by providing the external application with a set of service credentials, but as mentioned in the previous

chapter, introducing the service credentials to the client is not really feasible.

- The undelivered messages are stored. The Pub/Sub service has a built-in message queue that persists each undelivered message for up to 7 days[7]. This is problematic, taken into account that many of the devices we issue a command to might be offline at the moment of the request. This means that once the device goes online, we might end up delivering a number of commands that we're no longer interested in and that can possibly cause us harm if executed when not wanted. Say, you want to start and then stop your DDOS attack, but one device starts it on its own two days later. This can possibly lead to the exposure of our botnet.

2. IoT Service – perhaps a somewhat unexpected ally in this sort of use case, IoT service is capable of generating push notifications to the remote clients connected to it. As a matter of fact it might be even better suited for the job than the Pub/Sub service taken into account that the clients of IoT Service are by design outside of the cloud. The IoT service introduces one more form of authentication that is specifically designed to be used with IoT – the client generates a key (any of the following formats: RS256, ES256, RS256_X509, ES256_X509) that is later on registered in the IoT service allowing the client to uniquely identify itself in the service. In this case unfortunately we also end up hitting the wall due to a number of incompatibilities with our use-case:

- The notification is only generated through device configuration change. All configurations are permanently stored in the cloud and versioned, leaving in the same time a clear trace of what we did to a certain device.

[7] https://cloud.google.com/pubsub/docs/subscriber, 23.12.2018

- We face a similar problem as we had with the Pub/Sub service – if the device is offline at the time of notification publishing, then it still gets delivered as soon as the device goes online again.
- Only one command can be delivered at a time. This makes it complicated to perform quickly multiple operations one after another. Chances are that only the last one will be delivered in this case.

3. Firebase Cloud Messaging – Firebase is a whole another service provided by Google that aims to provide a universal backend for android/web applications. It greatly extends and simplifies the use of the Google Cloud Platform, hiding some of the configuration complexity of GCP as well as providing several additional services that are commonly used in both android as well as in web applications. One of those services is the Google Cloud Messaging service. This one meets all of our requirements. The messages are not persisted. They are not getting delivered to the client if issued while the client was offline. It allows us to generate multiple notifications at once without waiting until the previous one generates a response. The authentication however is a problem again. Firebase uses multiple levels of authentication. First there's the general application authentication key, that can in fact be easily shared between all clients using the service. The issue is that in the end we want to authenticate the specific client and in order to do that, Firebase either requires Email/Password authentication, or a federated authentication from one of the popular social media services, Facebook, Google+ or Twitter. Generating such accounts separately for each of our clients doesn't quite feel right.

4.2.4 Google Cloud Platform summary

While Google Cloud Platform sounds very promising, it is still one of the youngest ones available on the market and it lacks crucial functionality in

the area of authentication as well as the delivery of the push notifications. Despite the best efforts of working around the limitations of the platform, it appears that GCP is not a suitable candidate for solving the problem of this thesis.

4.3 AWS-based approach

4.3.1 Serverless applications

In AWS, as opposed to GCP discussed in the previous chapter, there's only one ultimate way of introducing the serverless backend logic – lambda. As mentioned in "Serverless Computing: Economic and Architectural Impact" by Gajko Adzic and Robert Chatley (2017, p. 884), Amazon was the first company in 2014 to introduce an approach of deploying application logic without the need to spawn a dedicated server. Once the research proved that Lambdas allow the users to save 66%-95% of the costs by redesigning their architecture to the serverless approach (since the main idea is that you only pay for what you use, instead of paying for the server all the time just to keep it running), all other major platforms started introducing similar solutions.

Lambdas, similarly like the Cloud Functions from the Google Cloud Platform are essentially small functions aiming to accomplish one small pre-defined goal. They can be triggered by a number of various events, starting with simple HTTP requests and ending with batch operations on large data streams (AWS Kinesis). In fact nearly every service on AWS can generate some sort of events that can be used as Lambda triggers.

4.3.2 Authentication

With a large number of bots connecting to our CnC application we have to make sure that we can send a command to a very specific one. It is also important that the bot doesn't have a possibility to start listening to messages meant for a different client. This could potentially allow a security engineer to take the whole botnet down. This is why we have to introduce a form of au-

thentication that would allow us to uniquely identify a certain bot and assign him certain access rights, that can allow him to access a push notification service of a certain kind, but not wide enough to let him see messages that are not meant for him.

Amazon introduces a number of different authentication methods depending on what kind of application requires to get the access to certain services provided by the platform.

1. IAM – Identity and Access Management

   As we can read in the Amazon's official documentation[8]: IAM service forms the base of any other form of authentication in the AWS platform. The IAM service aggregates various principals (either a human user or an application) and upon every request to any of AWS services, it validates the requested action against a set of assigned policies, deciding whether the user should be allowed or denied a certain action (see FIGURE 8).

2. Cognito

   As we can read from AWS Cognito documentation[9], Cognito allows user-based authentication. As a matter of fact it can be considered as two separate services:

   - Cognito User Pool – essentially a database of users registered directly in the system that is being developed on AWS.
   - Cognito Identity Pool – a database of references to users that are physically stored in different systems. You will use the identity pool for instance to authenticate the federated identity from Facebook or Google. It also allows us to assign certain IAM roles and policies even to unauthenticated users, which is something that could be used in certain architectures of our CnC application.

[8] https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html and https://docs.aws.amazon.com/IAM/latest/UserGuide/intro-structure.html, 26.12.2018

[9] https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html, 26.12.2018

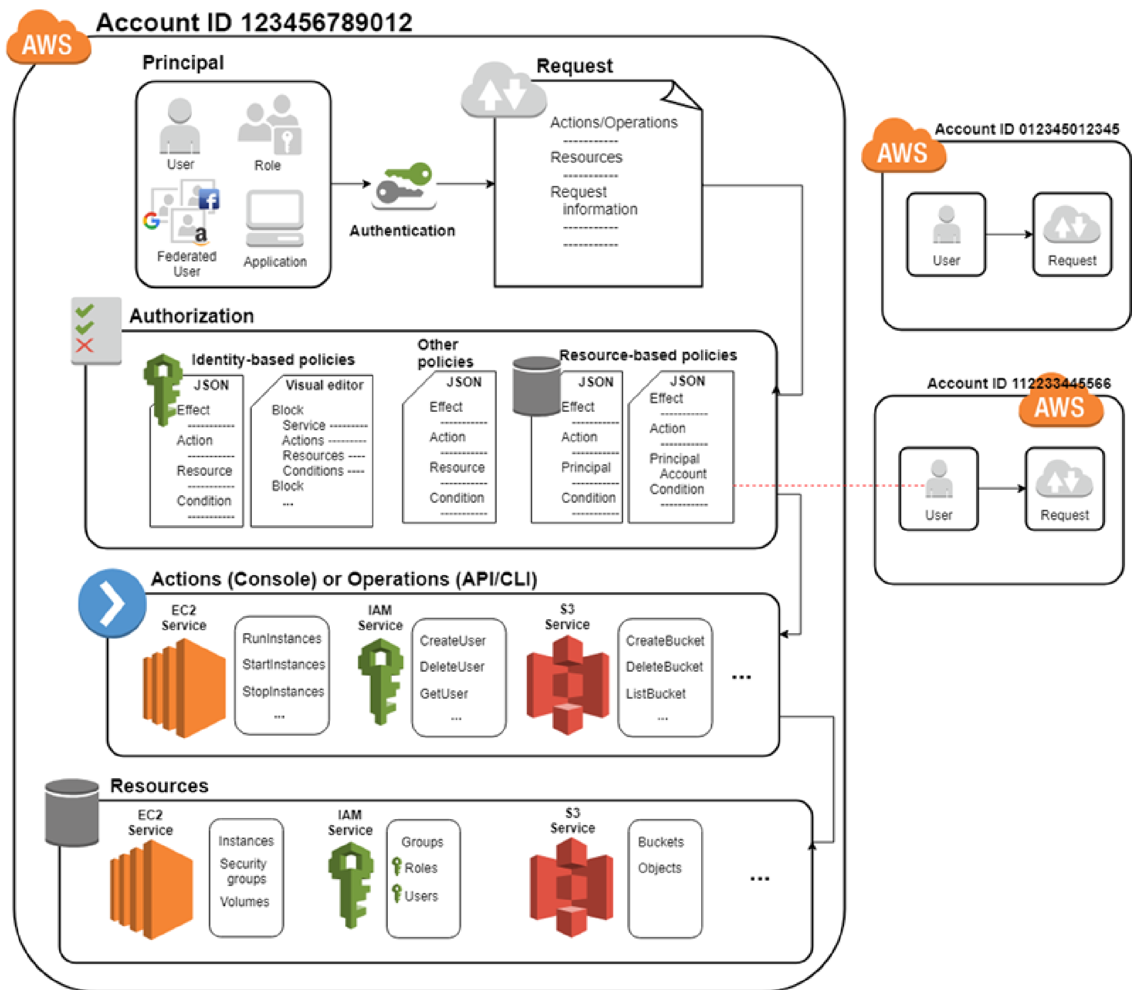


FIGURE 8: AWS IAM

Source: https://docs.aws.amazon.com/IAM/latest/UserGuide/intro-structure.html

3. IoT thing authentication

As we can find in the official AWS documentation[10] every Thing must have a pre-generated certificate that is linked to a specific Policy that defines what the device can do with the AWS account. Unlike the case of GCP, Amazon provides the full SDK allowing to generate certificates, defining policies and registering Things[11], so that the Thing registration can be easily automated through a Lambda. The Policy allows us in this case

[10] https://docs.aws.amazon.com/iot/latest/developerguide/iot-security-identity.html, https://docs.aws.amazon.com/iot/latest/developerguide/iot-security-identity.html, 26.12.2018

[11] https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/Iot.html, 26.12.2018

to specify what notification topics a certain Thing can register to, thus providing us with a possibility to make sure that different clients cannot start listening to messages that were not meant for them.

4.3.3 Push Notifications

There is a number of different ways to deliver a notification to an application in AWS. Let's take a closer look at them.

1. SNS – Simple Notification Service

   As we can read from the official AWS documentation[12], SNS is a service allowing the developers to embrace the concept of event-driven computing. It allows to publish notifications for other services, message queues, mobile applications and others. The very concept of the service suggests that this is something that could be easily used for delivering the remote commands to our bots.

   The message delivery can be configured with a number of different retry strategies[13] allowing us to make sure that the command we issue is properly delivered to designated recipients. Unfortunately, as soon as we try to configure SNS for our use-case, we find out that the service is primarily designed to deliver the messages to various services located within the AWS platform and while the service is advertised for being able to deliver the messages to external clients (in particular the mobile applications), it does so through integrations with external 3rd party platforms which in fact are designed specifically to provide the messages to mobile clients[14]. The integration with those however is fairly difficult without the specialized mobile SDK, which will not be available for our desktop clients. Additionally the security configuration of the service is fairly complex. We don't want different clients to be able to listen to messages

---

[12] https://aws.amazon.com/sns/features, 06.01.2019

[13] https://docs.aws.amazon.com/sns/latest/dg/DeliveryPolicies.html, 06.01.2019

[14] https://docs.aws.amazon.com/sns/latest/dg/sns-mobile-application-as-subscriber.html, 06.01.2019

meant for other clients. This means that each one of these clients will require a separate IAM Role and Policy. While the creation of these could be automated, it introduces a lot of mess in the system. Unfortunately AWS does not allow you to separate different applications into separate workspaces like the Google Cloud Platform does. This means that all applications hosted on AWS have to be placed in one shared account and as a result the IAM management becomes extremely messy, especially when one of the applications can dynamically generate thousands of entries.

In conclusion the SNS service, despite a very suggestive name and advertisements suggesting that this might be the right service for the job, is in fact not the right tool to deliver the commands to the remote clients.

2. AppSync

   AppSync, a very recently released (13.04.2018) new AWS service, is advertised as a solution allowing you to easily build, among others, chat applications[15]. As mentioned before, one of the most common protocols allowing the delivery of commands to bots is IRC which is in fact designed for online chat applications, hence this suggests that the service might actually be what we're looking for. As we can read in the AWS documentation of the service[16], the messages of AppSync are delivered via MQTT over web socket. This is quite convenient since MQTT additionally allows us to monitor in real time which of the clients are currently online and listening to new commands. The messages are delivered in the format of GraphQL objects and are triggered upon stored data mutation. This means that rather than explicitly generating a notification for the client, we should modify the value in the underlying data store and allow AppSync to generate the notification for us. While the AppSync wizard, that we can find in the AWS admin console, only allows to de-

[15] https://aws.amazon.com/appsync, 06.01.2019

[16] https://docs.aws.amazon.com/appsync/latest/devguide/real-time-data.html, 06.01.2019

fine a DynamoDB database as the underlying data store, there's still a number of other resolvers to choose from, that can be used instead when using command line tools, or CloudFormation template. One of the options is a simple AWS lambda. This means that we can in fact completely mock the data store however we want in order to achieve the wanted result. After all, we probably don't want to actually store every single command that we issue for a bot. That would be just unnecessary waste of disk space.

There are 4 ways of authenticating a client to the AppSync service[17]:

- API Key
- AWS IAM
- OpenID Connect provider
- Cognito user pools

As already discussed before, Cognito might be a somewhat uncomfortable form of authentication in this case due to the requirement of providing actual user information, such as email and password. This is not necessarily something that we want to generate for our bots. OpenID isn't any better considering that this service would have to be configured in a separate VPS, as it's not really a service provided by AWS. AWS IAM, as already mentioned before, could potentially generate a lot mess, making it difficult to manage the security as a whole in our AWS account. API Keys however are easily generatable by a lambda. The keys however have the maximum validity time of 365 days. This means that we have to explicitly introduce the functionality to periodically rotate the API Keys in thousands of clients while being able to identify them continuously as the same clients that just started using a different API Key. Such functionality would require careful investigation of all corner cases, like how do you do the rotation when the client is offline for a prolonged period of time and the key expires before the rotation was possible?

---

[17] https://docs.aws.amazon.com/appsync/latest/devguide/security.html, 06.01.2019

In conclusion, the approach appears possible to implement, although it feels a bit hacky. While the service appears to provide all the required features, it clearly isn't designed to deliver the remote commands. If we don't want to waste and pay for the disk space, we need to implement a custom mocked data store in the form of a lambda and then we have to create a mechanism allowing us to periodically rotate the API Keys.

3. IoT

   AWS IoT service, similarly as AppSync communicates with the remote clients via the MQTT protocol. This allows us to tell which of the clients are online at all times. The service registers the remote clients as Things. Each one of these can be easily assigned to a Thing Group, limiting the mess within the AWS account. Thing Groups also allow us to easily issue messages to a number of clients at once. The service provides 3 different forms of communicating with Things:

- Shadows
- Jobs
- Simple push notifications

As we can read from the AWS IoT documentation[18] shadows essentially represent the configuration of the Thing. They're represented by a simple JSON document that stores the information of the requested configuration as well as the last acknowledged by the client configuration. Every time a configuration is changed, a notification is delivered to the client that needs to explicitly confirm the receival of the new configuration. In that sense, the Thing Shadows work in a similar manner as the IoT Configuration in Google Cloud Platform which we discussed before and concluded that it's not really appropriate for delivering the remote commands to our clients.

Jobs are way closer to our desired effect[19]. A job can be represented by any form of JSON document. It can be created in a way that it is delivered to any

[18] https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html, 06.01.2019

[19] https://docs.aws.amazon.com/iot/latest/developerguide/iot-jobs.html, 06.01.2019

number of Things and the execution progress can be tracked in real time, as every Thing has to explicitly confirm the receival and execution of the Job. In fact, in case of longer jobs, a Thing can report the exact progress of the job execution. As the progress of the jobs is trackable, they have to be stored, but since they are stored directly in the IoT service, the user is not required to set up any additional database or pay for the storage of such data.

The simple push notifications are also an option in AWS IoT service. In that case the client has to subscribe for a specific topic that only he will be able to access. This means that a specific IoT Policies have to be created for each Thing separately, to make sure that they cannot listen to each-other's communication channels. The command delivered this way doesn't leave any trace on AWS account of what we issued, what arguably might make it the best option to deliver our messages.

In conclusion, AWS IoT service appears to be perfect for the use-case of delivering the remote commands in a serverless manner. The Jobs and Push Notifications allow us to handle the communication between a remote client and the backend in a number of different ways.

4.3.4 Design

In the previous section we determined that the best way to deliver the remote commands to the client in the AWS is through the usage of the Push Notifications generated by the IoT service. Let us now design how the whole application could behave in such situation.

IoT service requires that the communication with the outside world is handled through the SSH certificates. This means that our client should start by generating one and uploading the public key to the cloud, where it will be registered within the IoT service. In order to handle it in a secure manner, we can build a lambda triggerable by HTTP events that will receive the public key, register it within the IoT service, create the Thing, and generate the IoT Policy that specifies what Push Notification topics the thing is allowed to listen to.

Once the client is successfully registered, it can immediately subscribe to his IoT topic directly within the IoT service. Then as soon as the command is issued by an attacker, it gets delivered directly to the client, who in response can generate a response back to the IoT service, that may again be delivered back to the attacker.

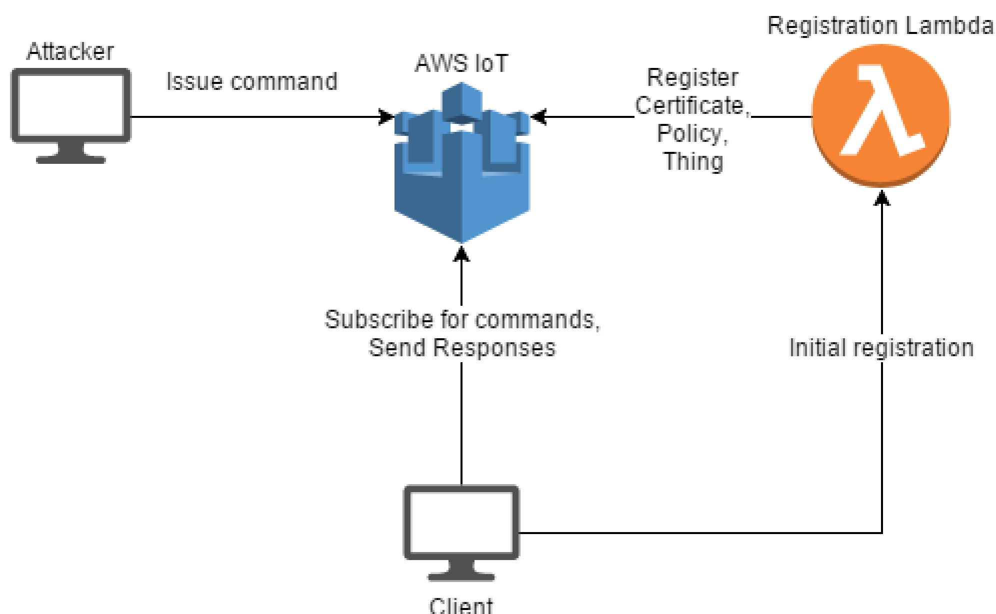


FIGURE 9: AWS IoT-based CnC design

The Appendix 8 contains the backend implementation of the design from FIGURE 9. As we can see from the comparison with the standalone CnC server (implementation provided in Appendix 1), the amount of required code is incomparably smaller and yet, thanks to the AWS cloud, it provides much wider area of applications. Right now, we're using only a small subset of functionality of the IoT service, but introducing for instance video/audio streaming wouldn't require any additional work on the backend side, whereas in the standalone solution it quite likely would require quite extensive changes, should we ever decide to introduce it.

The proposed solution however is not necessarily very clean. It requires the attacker to be directly connected to the IoT service in order to receive the instant response. This means that should there be more than one administrator

of the botnet, there is a requirement of introducing separate IoT topics for each one of them, to make sure that they don't receive responses for requests they didn't send personally. The IoT Jobs make it much easier to track who exactly issued a certain command, but they also leave a trace of what happened, which is something we don't necessarily want. Yet, it is necessary to create an IoT Job in order to easily identify what response was issues for what request.

4.3.5 Performance

We measure the performance of the application by deploying it to Amazon's eu-west-1 region (just like we did in case of the standalone approach before). The client will first register to the IoT service via lambda and then we will issue 1000 directory listing commands to estimate the time the client will need to produce a response.

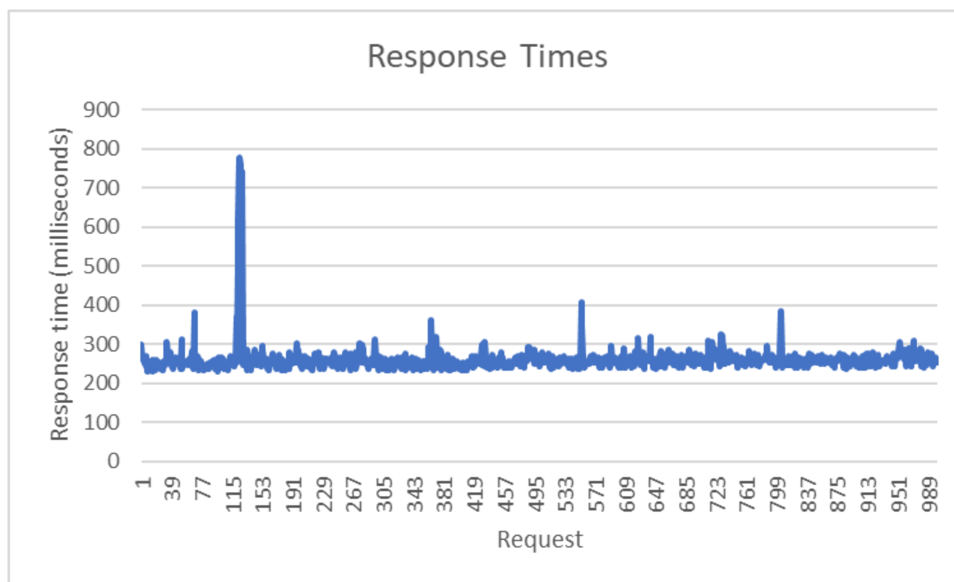


FIGURE 10: AWS-based client response times

As we can see from FIGURE 10 and FIGURE 7 the AWS-based approach is just a little bit slower than the standalone approach despite the fact that the IoT service that we make requests to still needs to internally consult the IAM service to figure out if the client has access rights to subscribe and publish to certain topics. Those internal requests however are performed within the same physical

data center, hence the latency is greatly limited. As a result median for the response time in the AWS-based serverless approach for the CnC application is only 256 milliseconds. Compared to the original 212 ms from the standalone CnC the difference is nearly unnoticeable, except that in this approach we no longer have the need for a VPS that would constantly run in the background to maintain the connection with the bots.

4.3.6 Cost estimation

Similarly, like we previously did with the standalone approach, let's try to estimate the cost of maintenance of the AWS IoT-oriented solution in order to determine if the serverless approach actually proved to be as cheap as advertised.

Just like in the case of the standalone application, we're going to use the AWS price calculator[20] in eu-west-1 region (Ireland) and we're going to skip the costs of S3 bucket as well as Route53 (DNS management) as those are only optional for the CnC application. We're going to estimate 1000 client registrations through a lambda running on 128MB of memory, where a single registration takes 2000ms of the lambda execution (during the tests the maximum execution time observed was 1100ms). Unfortunately AWS doesn't provide the price calculator for the IoT service, but it does publish a price list[21].

IoT service has 2 different kind of charges that we're facing: the cost for maintaining the connection to the service and the cost of actually issuing the messages.

[20] https://s3.amazonaws.com/lambda-tools/pricing-calculator.html, https://aws.amazon.com/lambda/pricing/, 06.01.2019

[21] https://aws.amazon.com/iot-core/pricing/ (06.01.2019)

TABLE 3 AWS IoT-based solution cost estimation

Cost per 1000 executions			
Service	Details	Reasoning	Price (USD)
Lambda	128MB, 2000ms timeout	Lambda is responsible for performing the registration in the IoT. The first 1M executions each month however are for free	$0.00
IoT	Maintaining the connection with all the clients	1000 clients connected for a month 24/7, each costing 0.08USD per million minutes of connectivity	$3.46
IoT	Issuing the messages	Up to 1 billion messages costs 1USD	$1.00
Api Gateway		HTTP events that trigger the lambda are originally generated by the Api Gateway - 3.50 USD per million requests	$3.50
			$7.96

As can be seen from TABLE 3, the cost is drastically decreased compared to the original standalone approach. In fact we have been able to lower the costs by 84.8%. This is because we no longer need to pay for a number of virtual machines that have to run at all times despite the fact that they're heavily underutilized. Instead we only end up paying for the resources we actually use. These numbers all in fact still rounded up. In our experiments the lambda didn't take 2000ms to register a new Thing. The computers running our client will not be online 24/7 as many users tend to turn their computers off for the night. As a result the more realistic price would be even lower.

4.3.7 AWS Summary

AWS proves to be a really powerful platform with a large number of interchangeable services allowing to minimize both the maintenance as well as the development cost. The vast amount of services allows the end-developer to write much less code while achieving the same or better result.

The AWS IoT service feels most intuitive and easy to configure. It also provides the perfect base for the CnC applications by providing simple inter-

face for Thing provisioning and configuration as well as management of the remote operations.

4.4 Azure-based approach

4.4.1 Serverless applications

Just like every other cloud platform discussed in this thesis, Azure also provides its own serverless platform allowing the developer to specify the backend logic without the necessity of implementing a full blown standalone application – the Azure Functions. The platform however introduces a significantly clearer manner of managing its functions in comparison to AWS and GCP, as it introduces the concept of Function Apps, each of which can contain a number of Azure Functions, while the application can be monitored as a whole.

Azure functions, similarly as AWS Lambdas, can be triggered by a wide range of different triggers. While in our case HTTP request is the only one that we need to initially issue a request, the full list includes 19 different kinds of triggers[22].

4.4.2 Push notifications and service-specific authentication and authorization

It appears that Azure cloud has fewer services that seem interchangeable at the first glance, compared to AWS. There are however two different services that allow us to publish a notification to a remote application. Also each one of them uses different form of authentication. It appears that Azure chooses a very different approach to authorization compared to other cloud platforms researched in this paper. Instead of introducing the centralized IAM service, it follows the concept of connection strings. Before a service can be used, the developer has to define a specific configuration of the instance that is required. This results in the

[22] https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings, 10.02.2019

generation of a new endpoint that is identified by a unique connection string, which is necessary for the client to use the service.

1. SignalR – as we can see from the marketing page of the service[23], SignalR is a WebSocket-based message broker designed to build any applications that require real-time updates, such as chats or games. As we can read in the Azure documentation[24] the service is well designed to be error-proof. The network connection should never be considered as particularly stable and SignalR takes that fact very seriously. This is why whenever the connection is dropped between the client and the service, the connection data on the server side is not immediately disposed of. The client still has all the information required to re-establish the connection and once it does, it receives all the messages that were sent for him during the time that the connection was down.
   There are numerous tutorials showing how to easily create a group chat using SignalR using just a few lines of code. The issue is that the group chat messages are received by all clients that are connected to SignaR endpoint. We don't want that to happen with the remote bots. It looks like the SignalR does not provide any authentication form on it's own though. Rather than that, it requires the user to specify an Azure Function that will allow the user to specify who he is and who he wants to send the message to. Unfortunately for us however, only a limited number of authentication choices is supported in this case. As the official documentation specifies[25] the client has to be authenticated with one of the following:
   a. Azure Cloud Directory
   b. Facebook
   c. Twitter

[23] https://dotnet.microsoft.com/apps/aspnet/real-time, 17.02.2019
[24] https://docs.microsoft.com/en-us/aspnet/signalr/overview/guide-to-the-api/handling-connection-lifetime-events, 17.02.2019
[25] https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-authenticate-azure-functions#log-into-azure-with-vs-code, 17.02.2019

d. Microsoft Account
e. Google

Unfortunately none of these area feasible option in case of a client that is generated automatically and never truly gets registered by a human user.

2. IoT Hub – another IoT service. Just like the Goggle and AWS ones, it supports HTTP and MQTT protocols, allowing either long polling or persistent connection. It uses just a slightly different approach for authentication and authorization than SignalR. While the IoT Hub is represented by its own connection string, this one is meant to be used only for administrative operations, such as registering a new device. Every device however is represented by a separate unique connection string. While this thing perhaps makes it a bit easier to implement the client side, as you no longer need to generate any certificates, like you normally would on AWS, on the server side it adds a new difficulty – you have to provide the connection string to the Azure Function somehow. In other clouds this is not an issue thanks to external IAM Service with which the applications don't necessarily have to be informed of what they are allowed to do.

   The IoT Hub provides a number of different ways of communication with the external client[26]:

   a. Cloud-to-device messages – meant as one way communication, although various telemetry operations from the client side are possible, allowing the user to build some form of bi-directional communication.
   b. Twin's desired properties – similar to Shadows in AWS IoT service. It's meant to store and provide the configuration of the device. The twin is represented by a JSON document specifying the current as well as the requested client configuration. Once the user submits a

[26] https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-c2d-guidance, 17.02.2019

new configuration change, the twin document is updated with the last requested configuration. Then the notification about the configuration change is sent to the device. Once the device confirms the receival of the new configuration the twin document is updated again to reflect that fact.

c. Direct methods – option completely unavailable in other cloud platforms. It is actually possible to expose a function on the client side in such a way that it feels similar as calling any other function in the local application. This introduces an extremely simple and intuitive way of building the communication between the backend and the client, where it's the backend that has to initiate the communication.

4.4.3 Design

As established in the previous chapter, the most convenient way of handling the communication with the remote clients is through the direct method calls that are possible thanks to the Azure IoT Hub. Before that can happen however, we have to register the client in the hub. That can be easily done inside an Azure function that is triggered by an HTTP request. Once the function registers the client, it returns to the caller a unique connection string that can be used to establish the connection with the hub and expose functions to the cloud. Those can then be easily accessed by other Azure Functions which are meant to be used by an attacker.

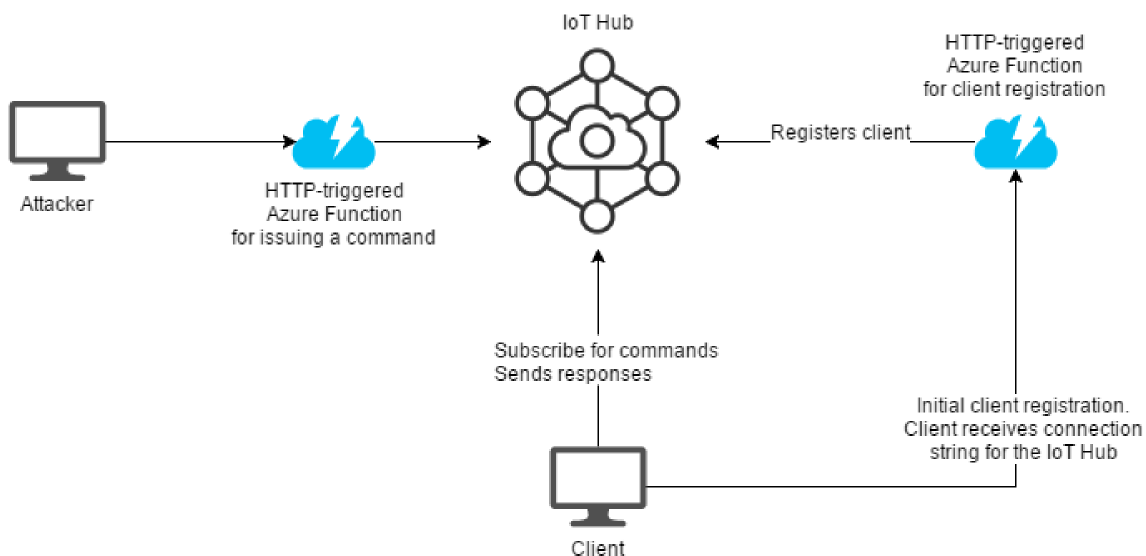


FIGURE 11 Azure-based CnC design

Appendix 11 contains the implementation of the design depicted in the FIGURE 11. As can be seen from the code, the implementation of the Azure-based approach is even shorter than the one for AWS. This is because AWS requires the user to explicitly create a certificate, Thing and correctly link them with the IAM Policy. In Azure however the Thing can only ever upload the telemetries to its own topic. It can upload a file, but also to a very specific place. The Thing can never interact with any non-IoT services hosted in the cloud, hence there is no need to specify any policy. There is no certificate, hence there is no need to upload it anywhere. All this makes the Azure implementation so much simpler. In addition to that the remote function invocation is significantly easier to use and maintain than anything AWS has to offer. As a result once the administrator calls the Azure Function to issue a command, the function can actually wait until the client responds and then forward the response directly back to the administrator as if it was just a simple single HTTP request.

4.4.4 Performance

We measure the performance of the Azure-based approach using the North Europe region which is physically located in Ireland, similarly as AWS's eu-west-

1. This allows us to limit the bias caused by the distance between the data centres and the location of test execution.

The client will first register to the IoT Hub via Azure Function and then we will issue 1000 directory listing commands to estimate the time the client will need to produce a response.

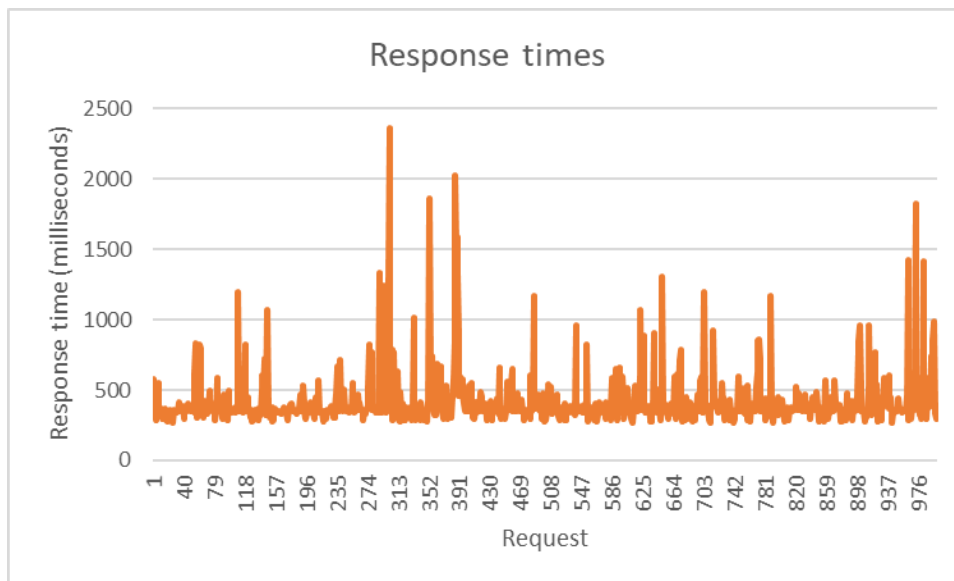


FIGURE 12 Response time for Azure-based CnC

As can be seen from FIGURE 12, the Azure IoT Hub appears to be significantly less performant than the IoT service provided by AWS. While the AWS requests take around 256 ms, the median request to Azure takes 360 ms. While for most use-cases it doesn't make much of a difference, it is a clear indication that the method is not necessarily the most efficient for heavier network operations, such as file transfer. Having said that, the cloud provides a number of other services specifically designed for that.

4.4.5 Cost estimation

Similarly, as with the standalone and the AWS approaches, let's try to estimate the cost of maintenance of the Azure IoT Hub oriented solution in order to determine if the serverless approach actually proved to be as cheap as advertised.

In order to perform the estimation, we're going to use the Azure price calculator[27] in the West Europe region (Ireland). We're going to estimate 1000 client registrations through an Azure Function running on 128MB of memory, where a single registration takes 2000ms of the lambda execution.

Cost per 1000 executions			
Service	Details	Reasoning	Price (USD)
Azure function	128MB, 2000ms timeout	Function is responsible for performing the registration in the IoT Hub. You start paying for the function only after the first million executions, when it costs $0.2	$0.00
IoT	Unlimited number of devices with up to 400 000 messages per day		$25
			$25

TABLE 4 Azure cost estimation

As can be seen from TABLE 3 and TABLE 4, it is clear that the Azure approach is significantly more expensive than the one built on top of AWS, despite the fact that the solution appears less performant. While the AWS solution costs us only $7.96, Azure asks for more than 3 times that amount for a similar service. It is still a half of the price of what the standalone CnC would cost us, but it does introduce grater latencies.

4.4.6 Development

The technical documentation and articles related to Azure cloud make it very clear that the solution has been built by Microsoft. It's in many cases very challenging to find a decent tutorial or even an official documentation that is not fully based on C# (a programming language developed by Microsoft). Then C# unfortunately requires Microsoft Visual Studio, which does not come for free and therefore is not necessarily easily accessible to many developers. The POC for the Azure-based CnC has still been written in JavaScript, but has been based greatly on GitHub examples that have been uploaded by Microsoft. The JS SDK

[27] https://azure.microsoft.com/en-us/pricing/calculator/, 16.03.2019

online documentation however is fairly limited and difficult to understand on its own.

Having said all that, it comes as a surprise that Azure implementation indeed is the least demanding. The platform services as such are extremely well designed and easy to understand and as a result the amount of code required to manage them, is absolutely minimal. In fact the combined number of lines of code for both the POC client and the backend in the Azure approach was under 100.

4.4.7 Azure summary

As mentioned before, the Azure CnC application seems the slowest of all that have been tested in this thesis. Also, while it is significantly cheaper to run than the standalone CnC application, it is definitely not among the cheapest options, due to the significant cost of the IoT Hub. The documentation regarding JavaScript also lacks behind, making it challenging to develop anything in this particular language.

Having said all that, the Azure solution still produces substantial cost savings and 100ms higher latency is not anything problematic in many use cases. Additionally, not everyone really likes programming JavaScript anyway, hence as long as C# remains an option, Azure remains the platform that requires the lest implementation. An experienced developer could implement a bot and the appropriate backend using this platform in only 1 day which is a significant improvement compared to the standalone CnC or the one specifically designed for AWS and that completely justifies the execution cost.

5 CONCLUSION

In this thesis we compared 4 different approaches to the development of Command & Control applications:

- A standalone approach that requires the developer to implement an entire service constantly running in the backend that is responsible for bot registration and issuing the remote commands through the REST interface
- A Google Cloud-based serverless approach, that in the end proved to be extremely challenging to implement due to limited capabilities of the platform
- An AWS-based serverless approach with the use of Lambdas and the AWS IoT service
- An Azure-based approach with the use of Azure Functions and the IoT Hub

We looked at the performance, costs related to the development and cost of running the solution on each of the platforms.

	Standalone	AWS	Azure
Performance	212 ms	256 ms	360 ms
Cost	$52.47	$7.96	$25
Difficulty/cost of development where 1 indicates very easy/cheap and 10 very difficult/expensive	10	5	2

TABLE 5 Comparison of working solutions

It has become clear that the standalone CnC, although the most performant, is also the most expensive to develop in the first place and the most expensive to run in the backend. We concluded that the Google Cloud-based CnC is not really feasible without a number of badly looking workarounds, due to the platform limitations. The AWS-based serverless CnC, although slightly slower than

the standalone one, proved to be pretty efficient in terms of command delivery and response. It is fairly simple to implement while keeping the project secure, and is definitely the cheapest option when it comes to costs of the platform services. The Azure-based CnC proved to be the least performant of all approaches tested in this paper, but the amount of time to receive the response from the client was still fairly reasonable. The cost of keeping the project running on Azure is higher than on AWS, but it still halves the cost of the standalone approach, while keeping the cost of the development at minimum. In fact the Azure implementation is the simplest of them all and the easiest to maintain for an already experienced developer. The possibility of remote execution of locally exposed functions and automatically receiving the returned value greatly simplifies the architecture and the implementation of the solution. It almost feels like the service was originally designed for CnC application development.

Therefore I conclude that despite the original assumption that AWS might be the most appropriate platform for the Command & Control application development for managing the botnet, the benefits of clear architecture and simplicity of code on Azure cannot be overlooked and therefore it is the most appropriate platform to build the serverless CnC application for.

DISCUSSION

The architecture and POCs proposed in the thesis are not the only ways of implementing the serverless CnC application on each of the platforms. In fact the proposed architectures were greatly simplified in order to minimize the development time and in real life additional components would be added, such as a database storing the details of each bot as well as the routing information, should the bot use some form of P2P communication, or a lambda/azure function that would pre-process responses from bots.

Additionally, the proposed cloud platforms are not the only ones available on the market, but only the most popular ones. This does not mean that other cloud providers don't provide services that could be cheaper or better suited for the job. In fact even in case of the discussed platforms the cost and performance of the provided services can vary depending on the region of hosting as no two data centres ever use exactly the same physical infrastructure.

"Botnet Communication Patterns" by Gernot Vormayr et al. (2017, p. 2772) mentions that moving from the start architecture to the hybrid one, where bots can receive the CnC-issued commands over a number of P2P nodes, can be more secure by lowering the chances of full botnet detection. This paper however does not cover this form of communication which might quite easily complicate the implementation and require more complex architecture.

In that regard, more research might be needed to optimize and simplify the application even further.

REFERENCES

Journal articles with doi:

Ihsan Ullah, Naveed Khan, Hatim A. Aboalsamh (2013). Survey on botnet: Its architecture, detection, prevention and mitigation. *10th IEEE International conference on networking, sensing and control (ICNSC)*, 660-665. doi: 10.1109/ICNSC.2013.6548817

Vormayr, G., Zseby, T., & Fabini, J. (2017). Botnet communication patterns. IEEE *Communications surveys and tutorials*, 19(4), 2768-2796. doi: 10.1109/COMST.2017.2749442

Gajko Adzic, Robert Chatley (2017). Serverless computing: economic and architectural impact. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 884-889. doi: 10.1145/3106237.3117767

Ayala, I., Amor, M., Fuentes, L., & Muñoz, D. (2017, November). An empirical study of power consumption of Web-based communications in mobile phones. In Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence & Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2017 IEEE 15th Intl (pp. 861-866). IEEE. doi: 10.1109/DASC-PICom-DataCom-CyberSciTec.2017.144

Journal articles without doi:

Tang, K., Wang, Y., Liu, H., Sheng, Y., Wang, X., & Wei, Z. (2013, September). Design and implementation of push notification system based on the MQTT protocol. In International Conference on Information Science and Computer Applications (ISCA 2013) (pp. 116-119).

Websites and technical documentation:

What is cloud computing. Retrieved on 24.06.2018 from source: https://aws.amazon.com/what-is-cloud-computing

What is botnet. Retrieved on 07.07.2018 from source: https://us.norton.com/internetsecurity-malware-what-is-a-botnet.html

Google. *Serverless computing.* Retrieved on 17.03.2019 from source: https://cloud.google.com/serverless

Google. *Google app engine documentation.* Retrieved on 17.03.2019 from source: https://cloud.google.com/appengine/docs/

Google. *Google cloud function documentation.* Retrieved on 17.03.2019 from source: https://cloud.google.com/functions/docs/

Google. *Authentication overview.* Retrieved on 17.03.2019 from source: https://cloud.google.com/docs/authentication

Google. *Subscriber overview.* Retrieved on 23.12.2018 from source: https://cloud.google.com/pubsub/docs/subscriber

Amazon. *What is IAM?* Retrieved on 26.12.2018 from source: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

Amazon. *Understanding how IAM works.* Retrieved on 26.12.2018 from source: https://docs.aws.amazon.com/IAM/latest/UserGuide/intro-structure.html

Amazon. *What is Amazon Cognito?* Retrieved on 26.12.2018 from source: https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html

Amazon. *Security and Identity for AWS IoT.* Retrieved on 26.12.2018 from source: https://docs.aws.amazon.com/iot/latest/developerguide/iot-security-identity.html

Amazon. *Class: AWS.Iot.* Retrieved on 26.12.2018 from source: https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/Iot.html

Amazon. *Amazon SNS features.* Retrieved on 06.01.2019 from source: https://aws.amazon.com/sns/features

Amazon. *Setting Amazon SNS Delivery Retry Policies for HTTP/HTTPS Endpoints.* Retrieved on 06.01.2019 from source: https://docs.aws.amazon.com/sns/latest/dg/DeliveryPolicies.html

Amazon. *Using Amazon SNS for User Notifications with a Mobile Application as a Subscriber (Mobile Push).* Retrieved on 06.01.2019 from source: https://docs.aws.amazon.com/sns/latest/dg/sns-mobile-application-as-subscriber.html

Amazon. *AWS AppSync.* Retrieved on 06.01.2019 from source: https://aws.amazon.com/appsync

Amazon. *Security.* Retrieved on 06.01.2019 from source: https://docs.aws.amazon.com/appsync/latest/devguide/security.html

Amazon. *What is AWS IoT?* Retrieved on 06.01.2019 from source: https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html

Amazon. *Jobs.* Retrieved on 06.01.2019 from source: https://docs.aws.amazon.com/iot/latest/developerguide/iot-jobs.html

Amazon. *AWS Lambda Pricing Calculator.* Retrieved on 06.01.2019 from source: https://s3.amazonaws.com/lambda-tools/pricing-calculator.html

Amazon. *AWS Lambda Pricing.* Retrieved on 06.01.2019 from source: https://aws.amazon.com/lambda/pricing/

Amazon. *AWS IoT Core Pricing.* Retrieved on 06.01.2019 from source: https://aws.amazon.com/iot-core/pricing/

Microsoft. *Azure functions triggers and binding concepts.* Retrieved on 10.02.2019 from source: https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings

Microsoft. *Real time ASP.NET with SignalR.* Retrieved on 17.02.2019 from source: https://dotnet.microsoft.com/apps/aspnet/real-time

Microsoft. *Understanding and Handling Connection Lifetime Events in SignalR*. Retrieved on 17.02.2019 from source: https://docs.microsoft.com/en-us/aspnet/signalr/overview/guide-to-the-api/handling-connection-lifetime-events

Microsoft. *Tutorial: Azure SignalR Service authentication with Azure Functions.* Retrieved on 17.02.2019 from source https://docs.microsoft.com/en-us/azure/azure-signalr/signalr-authenticate-azure-functions

Microsoft. *Cloud-to-device communication guidance.* Retrieved on 17.02.2019 from source: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-c2d-guidance

Microsoft. *Pricing calculator.* Retrieved on 16.03.2019 from source: https://azure.microsoft.com/en-us/pricing/calculator/

APPENDICES

Appendix 1 Standalone CnC proof of concept

Description: In order to validate that the theorised standalone architecture is a viable option, a proof of concept has been implemented and the implementation details can be looked up from the public Git repository.

URL: https://github.com/kamiljano/CloudDoorThesis/tree/master/poc/standalone

Appendix 2 Standalone CnC resource consumption measurement application

Description: In order to evaluate the resource consumption of the standalone CnC application, an application needed to be implemented that would spawn a the server and the number of clients that would connect to it and then measure the server resource consumption. The implementation of the application can be looked up from the public Git repository.

URL: https://github.com/kamiljano/CloudDoorThesis/tree/master/poc/standalone/e2e

Appendix 3 Standalone CnC resource consumption measurements

Description: The exact measurements generated by the application described in Appendix 2 have been saved to an excel file and can be looked up from the public Git repository

URL:
https://github.com/kamiljano/CloudDoorThesis/blob/master/generatedStats/standalone/resourceComparison.xlsx

Appendix 4 Standalone CnC performance measurements

Description: The exact performance measurements for Standalone CnC response times, depicted in FIGURE 7.

URL:
https://github.com/kamiljano/CloudDoorThesis/blob/master/generatedStats/standalone/performance.csv

Appendix 5 Standalone CnC performance measurement application

Description: Code of the application that enabled to perform the measurements described in Appendix 4.

URL:
https://github.com/kamiljano/CloudDoorThesis/blob/master/poc/standalone/e2e/performanceTest.js

Appendix 6 AWS-based CnC performance measurements

Description: The exact performance measurements for the AWS-based CnC response times, depicted in FIGURE 10.

URL:
https://github.com/kamiljano/CloudDoorThesis/blob/master/generatedStats/aws/performance.csv

Appendix 7 AWS-based CnC performance measurement application

Description: Code of the application that enabled to perform the measurements described in Appendix 6.

URL:
https://github.com/kamiljano/CloudDoorThesis/blob/master/poc/aws/e2e/performanceTest.js

Appendix 8 AWS-based CnC backend

Description: Code of the serverless backend of the AWS IoT-based CnC.

URL:
https://github.com/kamiljano/CloudDoorThesis/tree/master/poc/aws/CloudDoorBackend

Appendix 9 Azure-based CnC performance measurements

Description: The exact performance measurements for the Azure-based CnC response times.

URL:
https://github.com/kamiljano/CloudDoorThesis/blob/master/generatedStats/azure/performance.csv

Appendix 10 Azure-based CnC performance measurement application

Description: Code of the application that enabled to perform the measurements described in Appendix 9.

URL:
https://github.com/kamiljano/CloudDoorThesis/blob/master/poc/azure/e2e/performanceTest.js

Appendix 11 Azure-based CnC backend

Description: Code of the serverless backend of the AWS IoT-based CnC.

URL: https://github.com/kamiljano/CloudDoorThesis/tree/master/poc/azure/backend