An information-gathering botnet for private cloud environments

Elsabé Ros

10053663

Supervisor: Professor Hein Venter

Abstract

Organisations implementing a private cloud can run into problems related to misuse of resources, as well as various other issues. These problems are an issue because it is hard to gather information from the cloud environments. To mitigate these problems, the organisation must be able to monitor cloud usage closely, without interfering with employees' productivity. Using a botnet, traditionally considered malware, to gather data and report it to a central point for analysis, means that employees need not even be aware of the tool. Using a botnet instead of traditional information gathering tools has several advantages, but comes with its own set of disadvantages too. A botnet with a central server that collects data allows for big data analysis and easy maintenance of the botnet.

DECLARATION OF ORIGINALITY

UNIVERSITY OF PRETORIA

The Department of Computer Science places great emphasis upon integrity and ethical conduct in the preparation of all written work submitted for academic evaluation.

While academic staff teach you about referencing techniques and how to avoid plagiarism, you too have a responsibility in this regard. If you are at any stage uncertain as to what is required, you should speak to your lecturer before any written work is submitted.

You are guilty of plagiarism if you copy something from another author's work (eg a book, an article or a website) without acknowledging the source and pass it off as your own. In effect you are stealing something that belongs to someone else. This is not only the case when you copy work word-for-word (verbatim), but also when you submit someone else's work in a slightly altered form (paraphrase) or use a line of argument without acknowledging it. You are not allowed to use work previously produced by another student. You are also not allowed to let anybody copy your work with the intention of passing if off as his/her work.

Students who commit plagiarism will not be given any credit for plagiarised work. The matter may also be referred to the Disciplinary Committee (Students) for a ruling. Plagiarism is regarded as a serious contravention of the University's rules and can lead to expulsion from the University.

The declaration which follows must accompany all written work submitted while you are a student of the Department of Computer Science No written work will be accepted unless the declaration has been completed and attached.

Full names of student: Elsabé Ros

Student number: 10053663

Topic of work: An information gathering botnet for private cloud environments

Declaration

1. I understand what plagiarism is and am aware of the University's policy in this regard.

2. I declare that this mini-dissertation (eg essay, report, project, assignment, dissertation, thesis, etc) is my own original work. Where other people's work has been used (either from a printed source, Internet or any other source), this has been properly acknowledged and referenced in accordance with departmental requirements.

3. I have not used work previously produced by another student or any other person to hand in as my own.

4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

SIGNATURE 

Contents

List of Figures

List of Tables

1. Introduction

1.1 Introduction

Cybersecurity and the idea of "keeping honest employees honest" has always been a concern for large enterprises. The companies invest a large amount of money to guard corporate secrets and prevent misuse of company equipment. This includes software such as firewalls and proxy servers and access control mechanisms (both on systems and physical). Often, employees are also required to give their permission to have their communications monitored.

As the idea of cloud computing becomes more popular, many companies invest in their own "private clouds" for various reasons. These clouds are harder to regulate and control than a normal network, since it is possible to create and destroy virtual machines easily. Employees can thus use the cloud resources for personal (or illegal) purposes and easily get rid of the evidence once they are done.

To discourage people from abusing the cloud, companies need some sort of tool to monitor virtual machines in the cloud. This tool needs to report its findings to a central point, since a virtual machine can be destroyed at any moment.

One of the most efficient information gathering structures is a botnet, often used to steal personal information from unsuspecting victims. Botnets propagate using exploits and social engineering attacks and runs silently on a victim's computer, executing commands received from some central point.

The rest of this chapter examines the problem statement of this dissertation and its motivation. It also lists the tasks performed to reach the solution and provides an overview of the layout of the dissertation.

1.2 Problem statement

The problem statement of this dissertation is that companies can not easily monitor private cloud environments without changing the underlying architecture of the cloud.

The architecture of the cloud cannot be changed significantly for two reasons. The first reason is that cloud solutions are often proprietary, which means that the company will not be able to change the cloud's functionality, architecture or infrastructure. The second has to do with virtual machines that has already been created. These existing machine images must be monitored as well.

1.3 Motivation

Monitoring user activities in the cloud is a hard problem, even for large companies like Google and Amazon. Smaller organisations do not have the resources to implement and maintain complex solutions. This leads to a lack of accountability, which leaves the organisation open to a number of problems.

Some of these problems are:

- Misuse of resources
- Security of personal information
- Damage to company reputation

Each of these problems is discussed briefly below.

1.3.1 Misuse of resources

Lack of checks and balances on a cloud system can lead to employees and, in extreme cases, outside parties exploiting the cloud resources for personal gain. In some cases this can be legal (for example, mining for bitcoins) and in other cases illegal (launching distributed denial of service attacks).

1.3.2 Security of information

A lack of accountability on a cloud system can lead to employees (or external parties) gaining unauthorized access to personal or sensitive information they are not authorized to view. This can happen because of incorrectly configured security or by exploiting a vulnerability in the cloud computing environment.

1.3.3 Damage to company reputation

The problems named above, as well as problems not discussed, can damage the good reputation of a company if it should become public. Because many companies have a policy to publicise security breaches, some companies prefer not to use cloud computing at all.

1.4 Objectives

To achieve a solution to the aforementioned problem, the author performed a number of tasks:

- Literature study on botnets and cloud computing
- Set up a high-level model of the solution
- Determine the architecture of the solution
- Implement a prototype
- Evaluate the solution and prototype

1.5 Layout

This dissertation is divided into six chapters. Chapter 1 introduces the concepts of a private cloud and an information gathering botnet. It expands on the problem faced by corporations with private clouds.

Chapters 2 and 3 do a literature study on botnets and cloud computing. Chapter 2 gives an overview of botnets, including typical topologies, the lifecycle of the botnet, and propagation mechanisms. This chapter also goes into more depth on the two main classes of botnets, namely Command-and-Control and Peer-to-Peer. Finally, it gives some suggestions on detecting and defending against botnets.

Chapter 3 presents the definition of cloud computing as set up by the National Institute of Standards and Technology(NIST), going into detail on the essential characteristics, deployment and service models of the cloud as described in the NIST definition document. This chapter also briefly discusses the problems cloud computing is facing and gives a brief overview of cloud architecture.

Chapter 4 introduces a high level model of a suggested solution. It describes the three components of the solution and specifies how they will interact with each other. It also shows a high-level overview of the system.

Chapter 5 introduces an architecture for the solution, going into more detail for each component. This chapter also describes program flow and some specific architecture decisions of the prototype.

Chapter 6 critically evaluates the proposed solution, looking at the advantages, disadvantages and potential usages. It also briefly discusses privacy concerns and proposes some solutions to the problem of user privacy.

Chapter 7 summarises the work done in this dissertation and provides some suggestions for future research topics.

The appendices to this dissertation provide some more technical detail on the project. Appendix A discusses the build tool, compiler flags and build process, while Appendix B highlights some interesting implementation details from the source code of the bot and the Command-and-Control server.

2. Botnets

2.1 Introduction

The unique qualities of a botnet makes it ideal for information gathering purposes. In fact, a lot of botnets in the wild is used for exactly that purpose. This chapter defines the concept of a botnet, discusses the lifecycle of a botnet and covers botnet topologies. It discusses the two main classes of botnets, namely Command-and-Control and Peer-to-Peer botnets and gives an overview of botnet evaluation. Finally, it gives an overview of botnet detection and defence

2.2 What is a botnet?

Xuefeng [15] defines a botnet as a network of compromised computers, controlled by an attacker (called a "botmaster"). These infected computers (called "zombies" by some) runs malicious software installed via all kinds of attacking techniques [28], and may be located anywhere in the world [4], which makes destroying them virtually impossible [9].

The infected computers can be remotely controlled by the botmaster and can, in some cases, also communicate with each other [30]. In most botnets, this is done with Command-and-Control servers: the bot connects to a Command-and-Control server to receive its instructions [28]. However, recently Peer-to-Peer botnets have emerged. These botnets use the Peer-to-Peer protocol to communicate, eliminating the need for Command-and-Control servers.

The botnet can then be used by the botmaster to do tasks, such as launching Distributed Denial of Service attacks, sending spam, stealing information or spreading other malware [2, 30]. All botnets, no matter what their purpose or structure is, follows the same four lifecycle events.

2.3 Lifecycle

Xuefeng [15] defines the four phases of a botnet as follows:

Construction This phase involves the programming and spreading of a botnet. This includes defining the joining policies, topology construction and propagation techniques of the bots [9, 15].

Maintenance In this part of the lifecycle, the botmaster will update botnets when needed. This can include restructuring the botnet to reduce vulnerability or changing the location of the Command-and-Control server [9].

Attack The botmaster uses the botnet for some malicious activity. This can include sending spam email or executing a Distributed Denial of Service Attack [4, 29]. The attacker uses

some Command-and-Control channel to communicate with the botnets. This Command-and-Control channel can be either a push or a pull channel. A *push* mechanism, sends commands directly to the bot, while a *pull* mechanism publishes the commands and expects bots to retrieve them [28].

Recovery and reconstruction Similar to the maintenance phase, this phase happens after the botnet has matured. In this phase the botnet can be restructured and activities such as server migration (changing the physical hardware or address of a Command-and-Control server) can be executed [9].

Note that these phases may overlap or occur in a different order - it is not a distinct cycle. These four phases define the lifecycle of a botnet. However, they do not dictate the structure of the network created by the botnet, the topology.

2.4 Botnet Topology

The topology of a botnet refers to the way the bots and servers, if the botnet have servers, is structured in a network. This section looks at four possible botnet topologies, namely centralized, Peer-to-Peer, hybrid and unstructured. Emphasis is placed on the centralized and Peer-to-Peer topologies, since these two topologies are most widely used.

2.4.1 Centralized

A centralized botnet topology, illustrated in Figure 2.1, uses a central point to communicate with all the bots in a botnet [4]. Most traditional botnets uses this approach. This allows the attacker to quickly get messages to the bots, but if the central Command-and-Control server is compromised, the entire botnet fails [4].

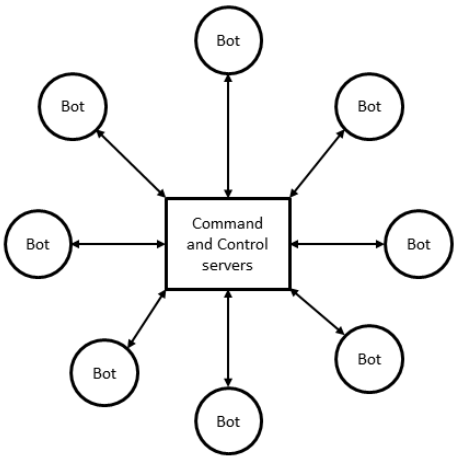

Figure 2.1: Centralized botnet

In a Command-and-Control topology bots can use various channels to communicate with the botmaster. Some of the most popular ones are discussed below.

IRC

Some bots uses IRC (Internet Relay Chat) to receive orders. The bot logs into an IRC channel and scans the channel for new commands issued by the botmaster [29]. This command channel is used because of the built-in support for client-to-client and client-to-multiple communication [2].

HTTP

The bot communicates with the Command-and-Control server using an HTTP channel. For more security this channel can be encrypted [29], either by using HTTPS, or by manually encrypting the data stream.

RSS/comments

Botmasters posts command in the comments section of a public blog, where the bots can read it via RSS. RSS stands for Really Simple Syndication and uses XML to publish content. The XML standard makes the data readable to both humans and machines. The bot then posts the results as a comment on a different blog, where it can be retrieved by the botmaster [10].

2.4.2 Peer-to-Peer

A few years ago, botnet authors realized that a Command-and-Control structure offers a single point of failure for the botnet, namely the one central server. They started using Peer-to-Peer structures, also known as Random Graph botnets [6], where bots communicate with each other instead of one central server [29].

Peer-to-Peer botnets make use of the Peer-to-Peer protocol to reduce or eliminate the necessity for a Command-and-Control server (Figure 2.2). Since the peers communicate with each other, it is much harder to damage or eliminate the botnet, but this topology has no guarantee of message delivery [4].

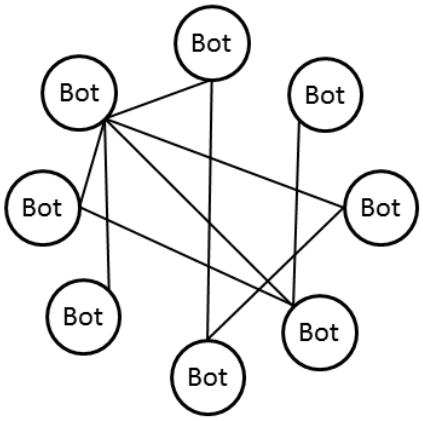

Figure 2.2: Peer-to-Peer botnet

Since there is no single point of failure, Peer-to-Peer botnets are much more robust and harder to take down than the traditional Command-and-Control structure [28]. Especially botnets that combine a Peer-to-Peer structure with encryption and obfuscation are hard to detect and remove [27].

Types of Peer-to-Peer botnets

Wang et al [28] defines three main types of Peer-to-Peer botnets, namely parasite, leeching and bot-only.

Parasitic botnets A parasitic botnet makes use of vulnerable hosts in an existing Peer-to-Peer network [28].

Leeching botnets Bots in the botnet joins an already existing Peer-to-Peer network once they become infected [28].

Bot-only botnet This type of botnet constructs its own Peer-to-Peer network [28].

Joining mechanisms

Joining mechanisms is used by a bot to connect to other bots in the network, in order to become a productive member of the botnet [15]. Xuefeng [15] names 6 joining mechanisms, namely random probing, initial list of bots, bootstrap procedure, connect to father node, using the benign system and multiple phases. The exact details of these mechanisms is not important for this dissertation.

Sending commands

There are various ways a botmaster can communicate with her Peer-to-Peer botnet, namely using the Peer-to-Peer network or some other channel to communicate with bots. The advantage of using the Peer-to-Peer channel is that the bot traffic is indistinguishable from normal traffic [28].

One of the ways to distribute commands to the botnet is by inserting them into the Peer-to-Peer network's file table. Each peer in a Peer-to-Peer network has a list of files available. This list is in the form <hash><file> and makes it simple to look up files another peer is requesting [13]. The botmaster inserts the command under a specific hash (often changed daily) and the bot finds the commands for the day under that specific hash [28].

Some botnets send a command to a bot and then have that bot distribute the command to its peers [30]. However, if a botnet uses an existing Peer-to-Peer botnet, all the peers of a bot may not be bots [28].

2.4.3 Hybrid

A hybrid topology combines the ideas of Command-and-Control and Peer-to-Peer (see Figure 2.3). Some bots in the botnet act like servers to other bots. In this way a hierarchical structure is created [27, 30]. Wang et al [27] calls the bots that acts as both clients and servers *servent bots*, a combination of the words "server" and "client".

2.4.4 Unstructured

This topology takes Peer-to-Peer even further by not allowing bots to know about each other (Figure 2.4). If a bot receives a message it scans the network until it finds another bot to pass the message to [4].

Once the botnet's topology has been chosen, the botmaster must find some way to deploy the bots onto the target machine. This is done by various ways, both manual and automated. The next section discusses some of the ways bots are loaded onto a victim machine.

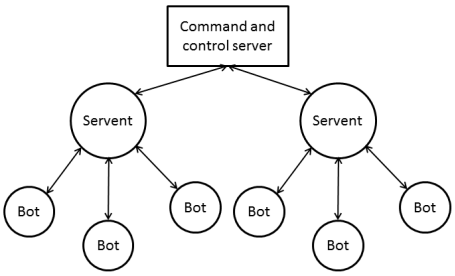

Figure 2.3: Hybrid botnet

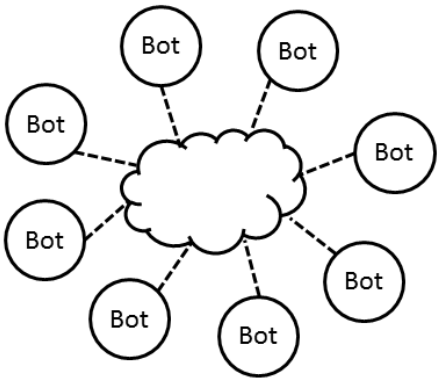

Figure 2.4: Unstructured botnet

2.5 Propagation mechanisms

Propagation mechanisms refers to the way that malware (including botnet clients) spread from one host to another. Botnets usually make use of the same propagation mechanisms as other malicious software [15]. Most botnets use multiple automatic propagation mechanisms [4]. Some of these mechanisms include social engineering attacks and drive-by downloads. Theses are discussed next.

2.5.1 Social engineering attacks

Thornburg [26] defines social engineering as a psychological and social process through which one person tries to gain information they are not authorised to have from someone else. The attacker poses as some trusted party asking the victim for information that the victim will willingly give to the trusted third party. This technique can be extended to convincing a victim to download and install malware.

2.5.2 Drive-by downloads

Drive-by downloads is a propagation mechanism where malware push and execute code onto a victim's system without their consent, and often without their knowledge. These attacks target the application layer of the OSI model, exploiting vulnerabilities in the browser when the user interacts with the attacker's website [18].

Once the bot has been installed on the system, it can start being a productive member of the botnet. However, infected users and the security teams for their systems want to minimize the damage of the botnet and ideally remove it from the system. These detection and defence mechanisms are discussed in the next section.

2.6 Detection and Defence

The detection and neutralizing of a botnet depends on the topology and size of a botnet. Some detection techniques, like DNS detection, are very effective against a centralized botnet, but virtually useless against unstructured botnets. The following section looks at various ways to detect and defend against botnets.

2.6.1 Detection

There are two main ways to detect botnets, namely by monitoring network traffic or by allowing a machine to become infected and join the network (a honeypot) [9]. This two mechanisms are explained in the sections that follow.

Monitoring network traffic

Feily et al [9] defines four classes of network traffic monitoring defences: signature-based, anomaly-based, DNS-based and mining-based.

DNS based bot detection focuses on detecting DNS lookups to suspicious or known botnet domains from within a network [4]. **Signature and anomaly** based detection techniques analyse network traffic to spot botnet-like behaviour, for example sending spam [4].

Honeypots

A honeypot is a host that is purposefully infected and allowed to join the botnet in order to collect more information about the botnet. As attacks become more sophisticated, it becomes more difficult for defenders to create convincing honeypots [4].

Defenders and researchers who uses honeypots to study viruses are bound by liability constraints: they cannot allow their honeypot to send out real attacks [27, 30]. This makes it easier for the attackers to identify honeypots. By launching an attack against another compromised machine, the attackers can quickly see if an infected host exhibits strange behaviour [30].

2.6.2 Defences

Since the earliest days of malware, defenders have been looking for ways to remove the infection from many computers as quickly as possible [25]. This section discusses some potential defences against botnets.

Infection detection

This technique focuses on detecting and neutralizing bots at install time. The technique used for this is the same as for any other malware [28].

Remove servers

Removing server(s) from the internet can severely hamper, and in some cases kill, a Command-and-Control botnet [28].

This technique can also be effective in limiting the growth of a Peer-to-Peer botnet. Many Peer-to-Peer botnets rely on a server for the initial bootstrapping procedure. Removing this server from the network prevents the infection of new hosts [27, 28].

In other cases, a set of peers is hard coded into the source code of the botnet. Removing those peers from the network effectively prevents new bots from joining the botnet [28].

"Hijack" a command server

By controlling the commands sent by a Command-and-Control server, defenders can attempt to minimize the damage done by a botnet. The same technique can be used to take over a intermediate bot in a hybrid network, where the defender only gains access to part of the botnet [30].

The loss of a botnet control server can be easily circumvented by the botmaster. By pointing the bot to a domain name instead of an IP address, it is a simple matter to change the DNS entry of a specific domain [30].

Index poisoning

The idea of index poisoning was originally used to prevent software piracy in Peer-to-Peer networks. The defence consists of inserting a large number of bogus records into the index of the Peer-to-Peer network. This results in bots not being able to find a command or downloading the wrong file [28].

This defence is made possible by two weaknesses of a Peer-to-Peer network: limited index keys and no central authority. Defenders can also use a honeypot to determine which index keys should be poisoned [28].

A way to circumvent this defence is by using dynamically generated keys. However, this comes at a cost: the botnet will be less scalable and, due to the generated keys, more detectable [28]. Another suggested way to mitigate the defence is to have the bots authenticate the index some way. Since only the botmaster has the necessary credentials, it becomes impossible to poison the index [28].

Sybil attack

A Sybil attack inserts fake nodes into a botnet to monitor or change the botnet traffic. This subverts the reputation system of the botnet. As a rule, specific high-value nodes is targeted to be replaced/mirrored by the Sybil attack [28].

In some Peer-to-Peer networks, like Kademlia, node Ids are not generated randomly, but rather calculated by hashing the IP address of the host. This makes it impossible for defenders to target specific nodes. If researchers were to insert a node into such a network, they would be introducing random nodes, reducing the effectiveness of the defence [28].

Cyber immune system

This controversial defence system mentioned in Wang et al [27] involves having a worm spread through the network of infected hosts, using the same propagation mechanism as the botnet. This worm will then remove the bot, patch the system and spread itself further.

2.7 Conclusion

There are many variants of botnets, all of them effective at what they do. The current trend in botnet development is toward Peer-to-Peer networks, which are seen as more robust and less vulnerable.

Botnets can run in virtually any environment, as long as it has some channel to communicate with the botmaster or with other bots. One of these environments is virtual machines within a cloud computing environment. Since this dissertation is concerned with running botnets in a cloud environment, the next chapter examines cloud computing.

3. Cloud computing

3.1 Introduction

Cloud computing is the latest buzzword in the industry. As such, the term is often applied to solutions that is not cloud computing. This chapter discusses the cloud computing definition put forward by the National Institute of Standards and Technology, give an overview of the problems the cloud is facing in terms of adoption and give a brief overview of cloud architecture.

3.2 Definition

The National Institute of Standards and Technology (NIST) defines cloud computing as follows [17]:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models."

The following sections discusses the five characteristics, three service and four deployment models mentioned in the definition.

3.2.1 Characteristics of a cloud

The NIST cloud definition [17] names five essential characteristics of a cloud, namely on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service. The next section gives a brief overview of each of these five characteristics.

On-demand self-service

A system in need of resources will be given them almost instantaneously as it needs them, without human intervention or direct (human) interaction with resource providers [7, 17].

Broad network access

The cloud instance can be accessed over the internet from a wide range of devices, including smart phones, tablets and computers at the user's location [7, 17].

Resource pooling

All the cloud provider's resources are pooled to provide the cloud service to multiple users. The users have no control over which hardware they use, although they may be allowed some choice at a high level (for example the country the resources are located in) [7, 17].

Rapid elasticity

The large amount of resources available to a cloud lets a system to scale up resources on-demand, often within minutes, allowing systems to handle load surges efficiently [3, 14]. To the user, these resources appear unlimited [17]. The user doesn't need an up front contract specifying the resources needed [7].

Measured service

Cloud systems automatically control and optimize resource usage by clients [17]. Since users only pay for the resources they actually consume (usage based pricing) [14], it is possible for users to start small and scale up as their application grows [3], which means that cloud computing has a low barrier for entry [14].

This five characteristics describe how a cloud service should appear to the user, but it does not address which different types of services should be presented to the user. This is the purpose of service models.

3.2.2 Service Models

The NIST cloud definition [17] names three cloud service models: SaaS, PaaS and IaaS. Dillon et al [7] adds a fourth model: DaaS or Data as a Service, providing on-demand virtualized storage. This model can be seen as a special case of IaaS.

Software as a Service

The cloud provider gives the client access to software running on the cloud. The software can be accessed through various devices and the client do not have direct access to the underling infrastructure [17]. Instead, the cloud provider will assign resources in such a way as to make operating the cloud as efficient and cost-effective as possible [7].

Platform as a Service

In this service model, the cloud provider allows the user to deploy their own applications to the cloud. The client do not have direct access to the operating system or the underlying architecture [17]. This model gives the user a development platform to work on, which means that the PaaS infrastructure need to support development tools [7].

Infrastructure as a Service

An Infrastructure as a Service cloud gives users a bare virtual machine, with control over everything from the Virtual Machine's Operating System upwards [3]. This gives the user complete access to all the computing resources, but not the cloud infrastructure [17]. This model makes extensive use of virtualization, since the users appear to have direct access to the hardware [7].

A cloud service can present different functionalities to a user. However, neither the characteristics nor the service models presents a model for how and where the cloud services should be deployed.

3.2.3 Deployment models

The NIST cloud definition [17] names four cloud deployment models: public cloud, private cloud, hybrid cloud and community cloud.

Public cloud

In this deployment model, the cloud resources is offered to the public in a "pay-as-you-go" manner, offering utility computing [3]. The cloud service provider has full ownership of the cloud hardware and software infrastructure and is responsible for setting policies [7].

Private cloud

The term "private cloud" refers to a cloud that is only available to one organisation [17]. It can be operated by the organisation, or the operation of the cloud can be outsourced to a third party [14]. This resources are not made available to the public or other organisations [3]. Organisations selects this option in cases where they are looking to optimise their own resources, but security and privacy concerns do not allow a public or hybrid cloud. Some other concerns include data transfer costs and necessity for full control [7].

Community cloud

A community cloud is a cloud shared by a number of consumers. It can be managed by the organisations using it, or it can be outsourced to a third party [17]. Parties sharing the cloud agrees on policies and requirements and this allows economic scalability for parties involved [7].

Hybrid cloud

A hybrid cloud combines two or more distinct cloud infrastructures bound together by some technology that allows data and application transfer between the two clouds [17]. These clouds remain unique entities, but allow for data and application sharing. This allows organisations to access the power of public clouds while simultaneously keeping business-critical processes on-site [7].

The NIST cloud definition gives a comprehensive description of what a cloud service can and cannot do. This means that an organisation can easily judge if the service they have been offered is in fact a cloud service. However, companies are still reluctant to take up cloud computing, because of various reasons discussed in the next section.

3.3 Challenges

Armbrust et al [3] names ten challenges around cloud computing. These can be divided into 3 categories, namely adoption, growth and policy and business obstacles. The ten challenges named below are categorized in Table 3.1.

3.3.1 Availability

Companies worry about the availability of computing resources from cloud providers. In this regards, large services like Amazon's cloud service, sets a high standard that smaller providers often cannot follow [3].

Table 3.1: Ten challenges in cloud computing

Category	**Problems**
Adoption	Availability
	Data lock-in
	Confidentiality and auditability
Growth	Data transfer bottlenecks
	Performance
	Scalable storage
	Bugs in the distributed system
	Scaling speed
Policy and business obstacle	Reputation sharing
	Software licensing

3.3.2 Data lock-in

Since there is no standard in place, most cloud APIs are proprietary and often not compatible [14]. This can make it hard for companies to extract data and software. It can also complicate a move to a different cloud provider and the process of implementing hybrid cloud solutions [3].

3.3.3 Confidentiality and audit capability

Companies are hesitant to adopt a cloud solution, because of security concerns. Since data are stored in a third party's data centres, there is no way for companies to directly regulate data security [14]. Organizations often require audit capabilities , which many cloud providers do not cater for[3].

3.3.4 Data transfer bottlenecks

Storing large amounts of data in the cloud complicates data transfer, since internet speeds are slow compared to the read/write speed of hardware [14]. A innovative solution for this problem is to write the data to hard drives and then to use overnight shipping to send the physical hard drives to their destination [3].

3.3.5 Performance

Since multiple virtual machines share the same physical resources, it is possible that a virtual machine's performance can be negatively impacted by other machines sharing the same resources. This is especially true for disk I/O [3].

3.3.6 Scalable storage

The advantages of the cloud, namely rapid scaling and on-demand resources, can not be applied as easily to persistent storage as it can to other computing resources [3].

3.3.7 Bugs in the distributed system

Bugs in large, distributed systems are often hard to fix, since they often cannot be reproduced in smaller instances of the same system [3].

3.3.8 Scaling speed

One of the most-mentioned advantages of cloud is rapid scaling: allocating more resources when a system needs them and scaling back down when the load drops. To be useful, this scaling needs to happen very quickly after the load on the system is increased [3].

3.3.9 Reputation sharing

Since many customers use the same cloud, the reputation of a company or business entity may be damaged through the actions of one of the other parties using the cloud. An example of this is the blacklisting of cloud IP addresses because of spam [3].

3.3.10 Software licensing

Most current software licenses deals with the number of computers a piece of software may be installed on. This licensing model does not translate well to a cloud environment and a "pay-as-you-go" costing [3].

After companies decide to adopt cloud computing, they have a number of technical decisions to make, one of which is where to deploy it. The next section gives a brief overview of a cloud's architecture.

3.4 Cloud architecture

Pallis [24] organizes the service models into a cloud stack as pictured in Figure 3.1 and calls it XaaS (everything as a service).

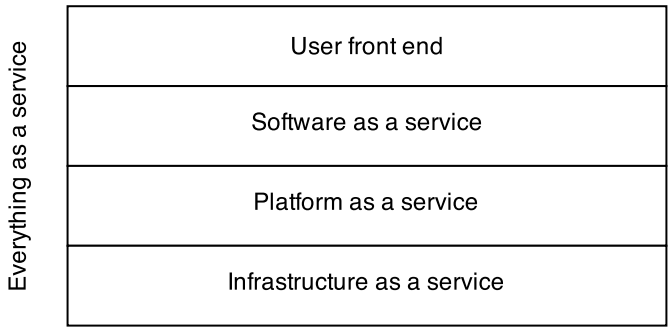

Figure 3.1: Service model stack

Underneath this cloud stack is the cloud software and architecture that combines physical resources and allows users to request virtual resources. This is illustrated in Figure 3.2.

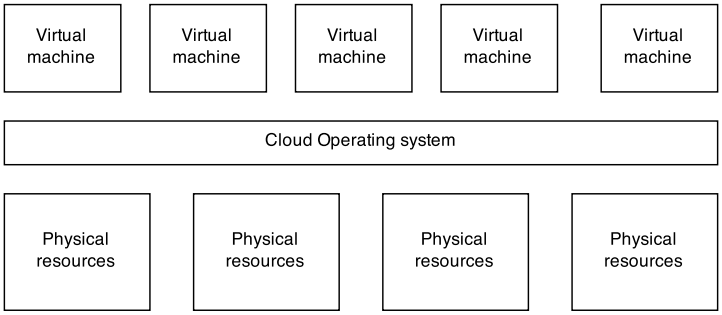

Figure 3.2: The cloud OS acts as intermediary between physical hardware and virtual machines

3.5 Conclusion

This chapter discussed various aspects of cloud computing, including the definition, problems it is facing with regards to adoption and the architecture of a cloud system. The next chapter presents the model for a solution to the problem stated in the first chapter.

4. High-level model for an information-gathering botnet

4.1 Introduction

Companies can not easily monitor activities in their private cloud environments, since virtual machines can easily be created and destroyed. This makes abusing the computing resources of the companies simple and low-risk.

This chapter introduces a high-level model of the solution, namely to create a bot and deploy the bot to every virtual machine in the private cloud. This bot can then gather information on its host and send it back to a central server for processing.

4.2 Components

This solution consists of three parts: the Command-and-Control server, the actual bot and the infection vector. The infection vector is responsible for deploying the bot onto the target virtual machines. This can be accomplished in various ways, as discussed in section 4.2.1. After the bot is up and running on the host, it starts communicating with the server. The bot and the server are discussed in sections 4.2.2 and 4.2.3 respectively.

4.2.1 Infection vector

The infection program (also called the bootstrap program) is responsible for infecting a host and installing the main bot executable. In this case, the hypervisor is modified to install the bot before booting the operating system of the virtual machine.

4.2.2 The bot

The bot is gathering information on a specific host. After being installed by the infection vector, the bot periodically pings the command and control server for commands. It also sends any information collected back to the server. In addition to information gathering, the bot has several other pieces of functionality, namely update, delete and reporting capability. Another important requirement is *extensibility* - it should be easy to add new functionality to the botnet. These components are discussed in the sections below.

The bot can have many information gathering modules running, which means that its implementation needs to be able to handle multi-threading. Shared resources must also be accessed in a thread safe manner.

Update

The bot updates itself from the Command-and-Control server when given the command. This allows the botmaster to add new functionality after the bots is deployed.

Delete

In cases where a bot needs to be removed from a client machine, it can easily be deleted by issuing the delete command. The bot then removes itself from the infected system.

Reporting

This is the functionality to send back the gathered information to the Command-and-Control server. The exact structure of the interaction will be dependent on the specific information gathering module.

Extensibility

It should be easy to add new functionality to the bot. This means that the solution should be structured in a modular way to simplify the additions of new modules.

4.2.3 Command-and-control server

The Command-and-Control server listens for incoming connections from active bots. It then either passes back commands for the bot to execute, or it saves the data the bot is passing back to it. In order to handle multiple bots connecting to it, the server needs to be optimised for multi-threading.

This section described the high-level functionality of each of the components. However, the way they interact with each other is not yet defined. This is the purpose of the next section.

4.3 Bot-server communication

Since the availability of bots is not guaranteed, the communication between the bot and the server makes use of a pull mechanism (see Section 2.3 for a description of this mechanism). The bot "pings" the server at fixed intervals, sending its version number. The server then replies with any commands the bot needs to execute. See Figure 4.1 for an illustration of this process. Section 5.2.1 of the next chapter discusses this in more detail.

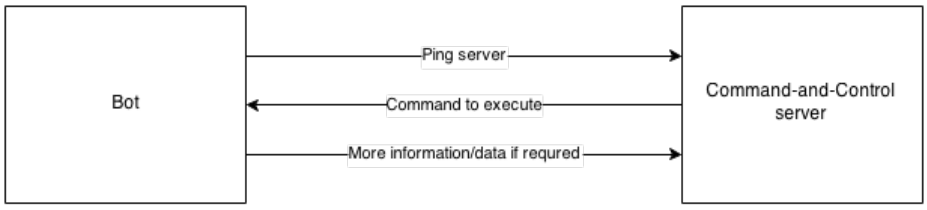

Figure 4.1: The bot pings the server

The response coming back from the server varies based on the command the bot has to execute. Table 4.1 summarises possible server responses.

Table 4.1: Possible server responses to a bot's ping request

Command code	**Description**
D	Destroy. Removes the bot from the host
U	Update. The bot downloads and install the latest version from the Command-and-Control server
.	Server acknowledges connection. No commands right now

In addition to the ping request, the bot has various other request codes it uses. See Table 4.2 for all request codes the bot can send the server.

Table 4.2: Bot request codes

Code	**Description**
P	Ping the server for any new commands
S	Bot is sending gathered data to the server.
U	The bot is requesting to download the latest version.

4.4 Conclusion

This chapter discussed a proposed model for the solution, including an overview of the expected functionality and a model for communication between the bots and the server. The next chapter presents a functional overview of the prototype built to solve this problem.

5. Architecture and prototype

5.1 Introduction

This chapter introduces a prototype to solve the problem described in chapter one. It presents the architecture of the components and describe interesting parts of the implementation. The next section discusses some overall aspects of the solution. The sections thereafter focuses on the specific components, namely the bot and the Command-and-Control server.

5.2 Overall

Many of the architecture and implementation choices are shared by the bot and the server. Since the two components share a code base, reuse is easy.

5.2.1 Architecture

The overall architecture of this system is a client-server architecture. The bots in the botnet act as the clients, while the central Command-and-Control server distributes commands, receives information, and stores that information.

Figure 5.1 gives a high level overview of how the various components in the solution interacts. The bot connects to the server (number 1 in the diagram). The server receives the connection and passes it a separate thread to handle (number 3). The thread sends any information necessary back to the bot (2). The bot component receiving the information passes it on to another component if necessary (4).

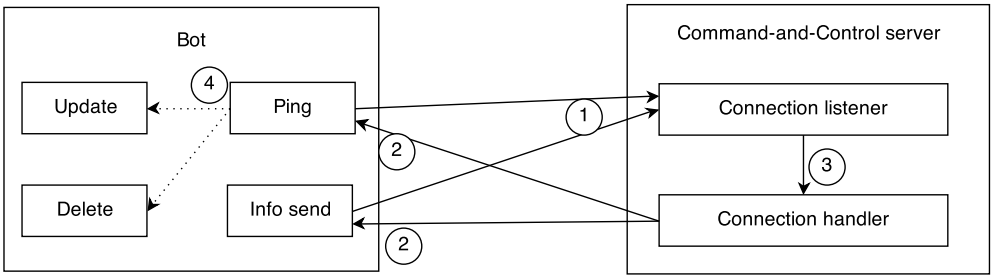

Figure 5.1: Model for an information gathering botnet

To make a chosen concept simpler to implement, third party libraries providing functionality is often used. The next section discusses the third party libraries used in this project.

5.2.2 Libraries used

Both the bot and the server make use of the same set of third-party libraries to allow for an easier, more efficient implementations. The libraries used for this prototype is discussed next.

Boost

Boost is a set of C++ libraries, providing functionality and implementations that the standard C++ libraries do not provide. It is designed to interface well with standard C++ libraries and aims to provide reference implementations, which is often incorporated into the C++ standard libraries at a later stage [5]. Boost is used for various purposes in this project, including being a dependency for OpenP2P.

OpenP2P

OpenP2P was designed to make the implementation of Peer-to-Peer networks easier. Functionality it supports include basic TCP and UDP connections, Kademlia support and various other protocols [23]. In this project, OpenP2P is used for the network connections between the bots and the server, in the form of TCP connections.

Linux headers

The botnet makes use of some libraries provided by the Linux operating system to run. Below of short description of the functionality each header file contains.

fcntl.h This header provides file controls, including functionality to open and read a file and set permissions [19].

linux/input.h The Linux input header defines how programs interact with the Operating System's input events. It uses maps and types to send input events back to user space [16].

sys/stat.h This header contains the data returned by the various *stat* functions, including fields such as file permissions, the number of links to a file and the file modes [20].

sys/types.h This header contains a list of data types, including device IDs, block sizes and file attributes [21].

unistd.h The unistd header contains symbolic constants and types as well as various miscellaneous functions [22].

Easylogging++

Easylogging++ is a single-file, light-weight logging library for C++. It is portable and allows both quick setup and detailed configuration, depending on the needs of the user. It is thread save and provides optional customized log formats [8]. This library is used both bot and server-side to log system state.

Libraries provide pieces of functionality to the programmer. However, libraries are written to be flexible and work across many different projects. This means that the programmer still needs to implement specific pieces of functionality. The next section examines one such piece: the communication protocol between the bot and the server.

5.2.3 Communication protocol

In order to communicate with the server, the bot needs to do network requests to the server. The structure of the request sent by the bot and the possible replies from the server are summarized in Table 5.1. The request codes in this table corresponds to the codes in tables 4.1 and 4.2. The data types of each part of the requests are defined in Table 5.2. This table also includes a short rationale for each data type choice.

Table 5.1: Structure of bot requests

Code	**Purpose**	**Request structure**	**Possible server responses**
P	Ping	<ping code> <bot version>	<update> <destroy> <received>
S	Transmit data	<transmit code> <bot version> <module code> <data length> <data>	<received>
U	Update	<update code> <bot version>	<binary size> <new binary>

Table 5.2: Request and response datatypes

	Data type	**Description**	**Rationale for datatype**
command codes	unsigned 8-bit integer	Codes used between bot and server to send commands, for example, update, transmit data and destroy.	Using an 8-bit integer for these values allows 256 unique commands. If more commands are needed, codes can be combined.
bot version	unsigned 8-bit integer	The bot version is transmitted as part of every request	An 8-bit unsigned version code allows for 256 versions of the bot, which makes provision for a fairly long lifetime.
data size	unsigned 32-bit integer	The size of data files uploaded to the server and the size of binaries downloaded from the server.	An 32-bit integer allows for files with a size up to 2^{32} to be transmitted. This means that large files can be transmitted without any problem.
data	array of unsigned 8-bit integers	Uploading gathered data and downloading bot binaries	Since data is received from the operating system as bytes, it makes sense to transmit it as bytes.

As can be seen in Table 5.1, the version of the bot doing the request is included in every request sent to the server. This was done for two main reasons: service versioning and statistics.

Service versioning As a system ages and matures, updates to the system often changes the structure of network calls. This is not an issue in a strictly controlled environment, but often clients "in the wild" do not update to the latest version of the software immediately. Passing the version code of a client to the server mitigates this problem by allowing the server to use the implementation most suited to that specific version of the client program.

Statistics Reporting the version code with every network request allows the botmaster to see version percentages. In an ideal situation, all bots would be the latest version. If this is not so, the botmaster could find ways to encourage the update of bots.

This section discussed the libraries shared by the bot and the server. It also defined the communication protocol between the two parties. However, each component has some specific implementation details. The next section discusses the implementation of the bot.

5.3 Bot

The bot is the program installed on a client's machine. This bot runs in the background on a system, without the user's knowledge and send gathered information back to the Command-and-Control server. It also periodically checks for additional commands from the server.

This section will first give an overview of the program flow of the bot, showing the flow of each component separately and then combining them. The second part of this sections goes into some more details on how the bot was designed.

5.3.1 Program flow

This section shows the high-level program flow of the bot. Program flows are discussed briefly on a thread by thread basis. Figure 5.9 brings all these components together in one diagram.

Main thread

Figure 5.2 gives a high level overview of the program main flow of the bot. The main thread of the program is responsible for starting other threads and after that just keeping the bot alive.

Installer thread

The installer thread checks if the bot is installed already. If it isn't installed (i.e. the infection just happened), it installs itself and make the system changes needed to start up when the operating system starts up (Figure 5.3).

Ping thread

The ping thread works on a timer. Every n minutes the thread wakes up and contacts the server. Once the command is received from the server, the ping thread dispatches to the correct handler for the command that the server sent (Figure 5.4).

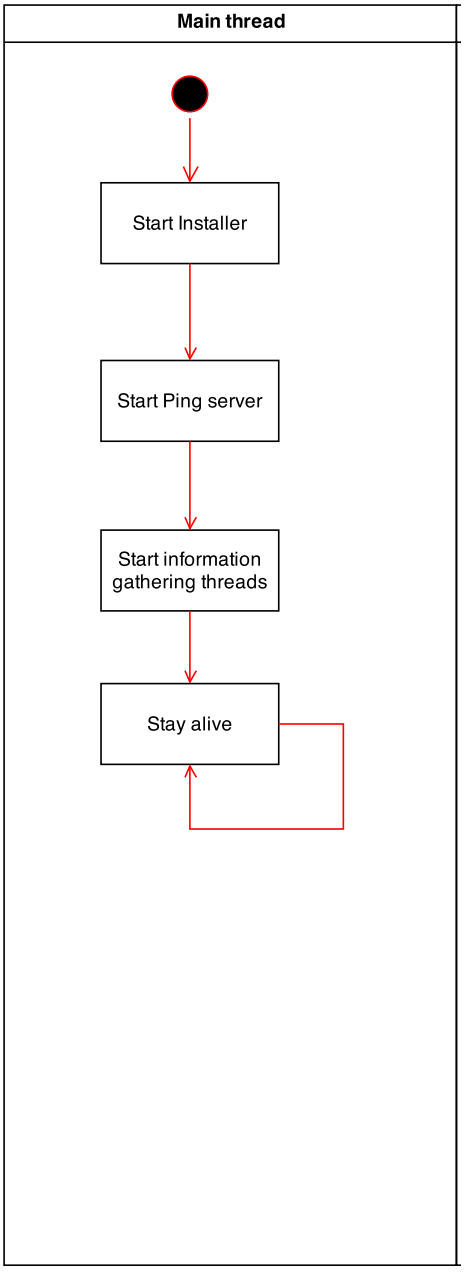

Figure 5.2: Overview of program flow

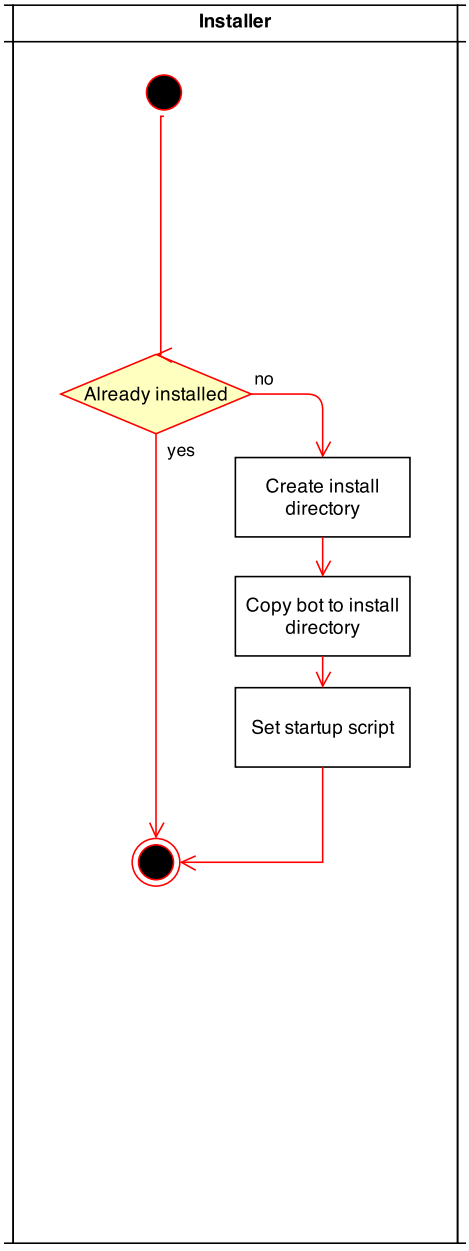

Figure 5.3: Overview of installer program flow

Keylogging thread

The keylogging thread is started up when the bot starts up. It hooks into the keyboard device manager and listens for keypresses. All keypresses are logged to a file. Once the file reaches a certain size, the data in the file are transmitted to a sever and the file is reset (Figure 5.5).

Update

The update thread is started if the ping thread receives the update command from the server. It downloads the new version of the bot, installs it and then restarts itself (Figure 5.6).

Delete

The delete command is kicked of if the ping thread receives the destroy command. The delete command removes the executable from disk and then kill the bot by exiting with an error code. See Figure 5.7.

Send data

The send thread is responsible for transmitting the data gathered by other modules to the server. As parameters, it takes in the data length and a pointer to a byte array of the data. It then transmits the data length followed by the data to the Command-and-Control server (Figure 5.8).

This section discussed the program flow of the bot client. The next section highlights some other choices made in the implementation of the bot.

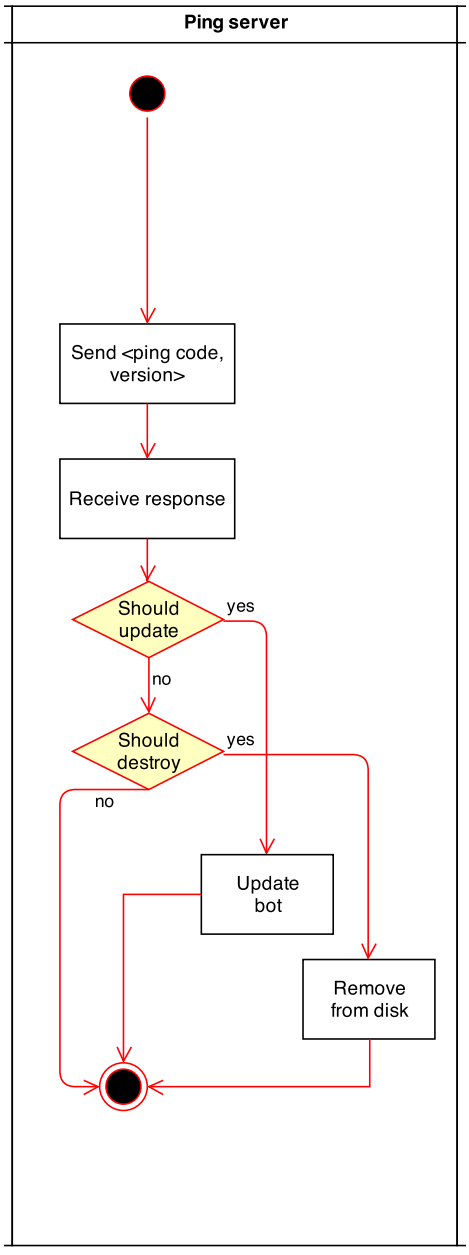

Figure 5.4: Overview of ping program flow

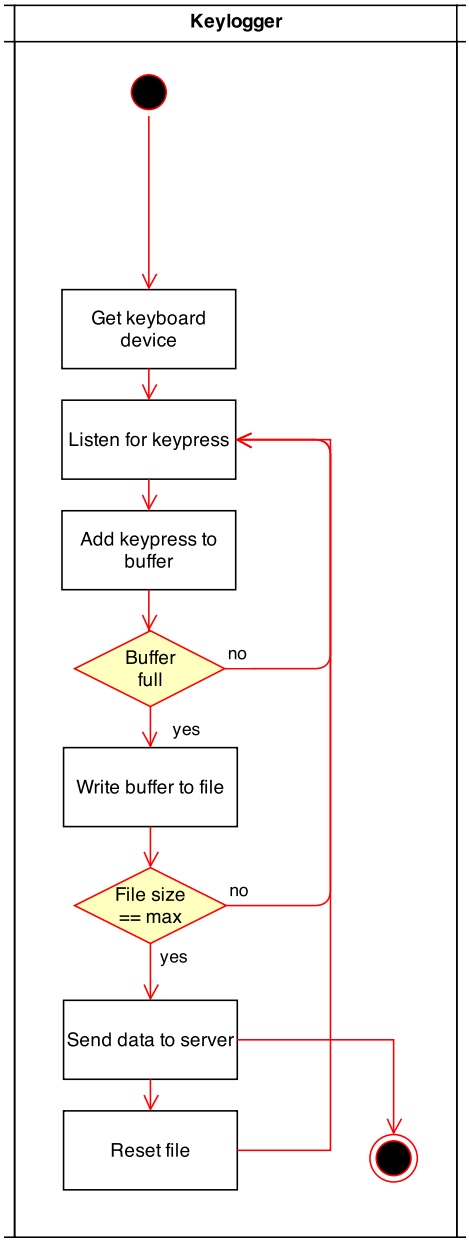

Figure 5.5: Overview of keylogger program flow

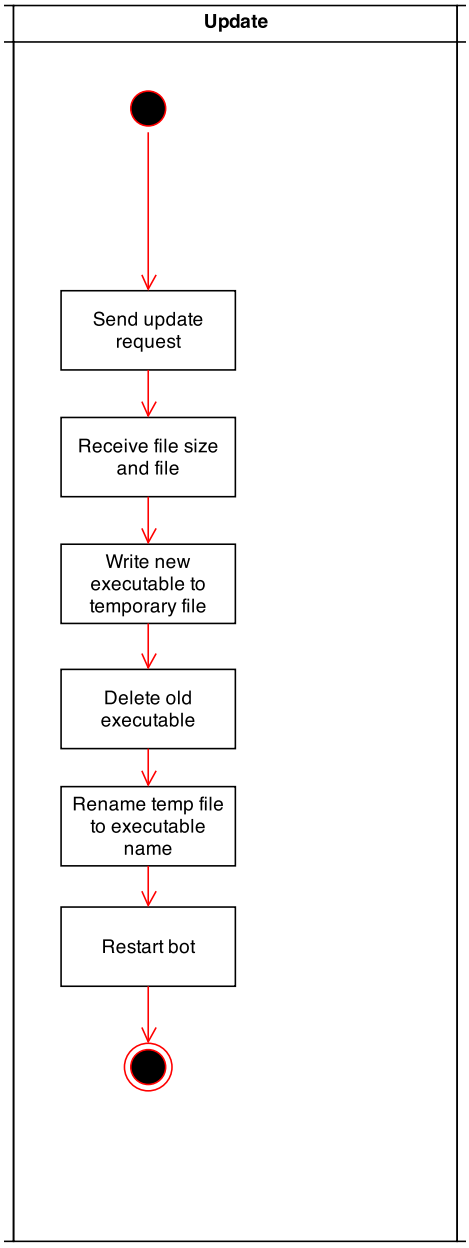

Figure 5.6: Overview of update program flow

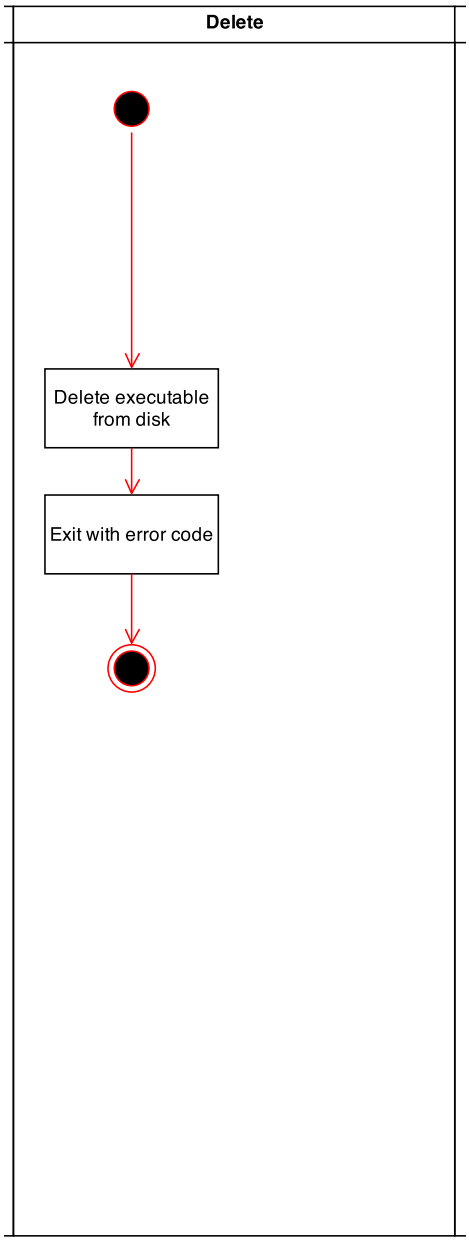

Figure 5.7: Overview of deletion program flow

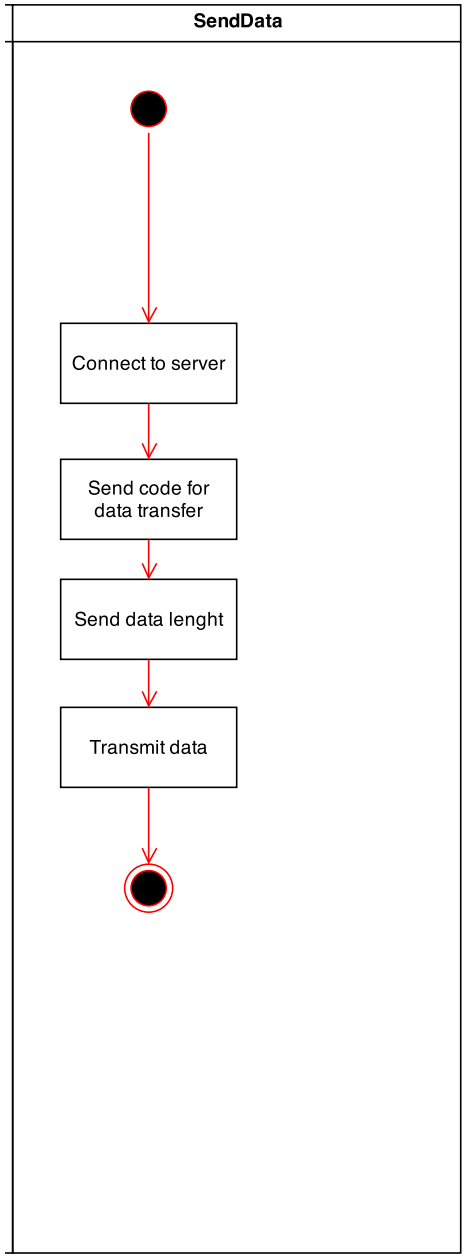

Figure 5.8: Overview of program flow to transmit data

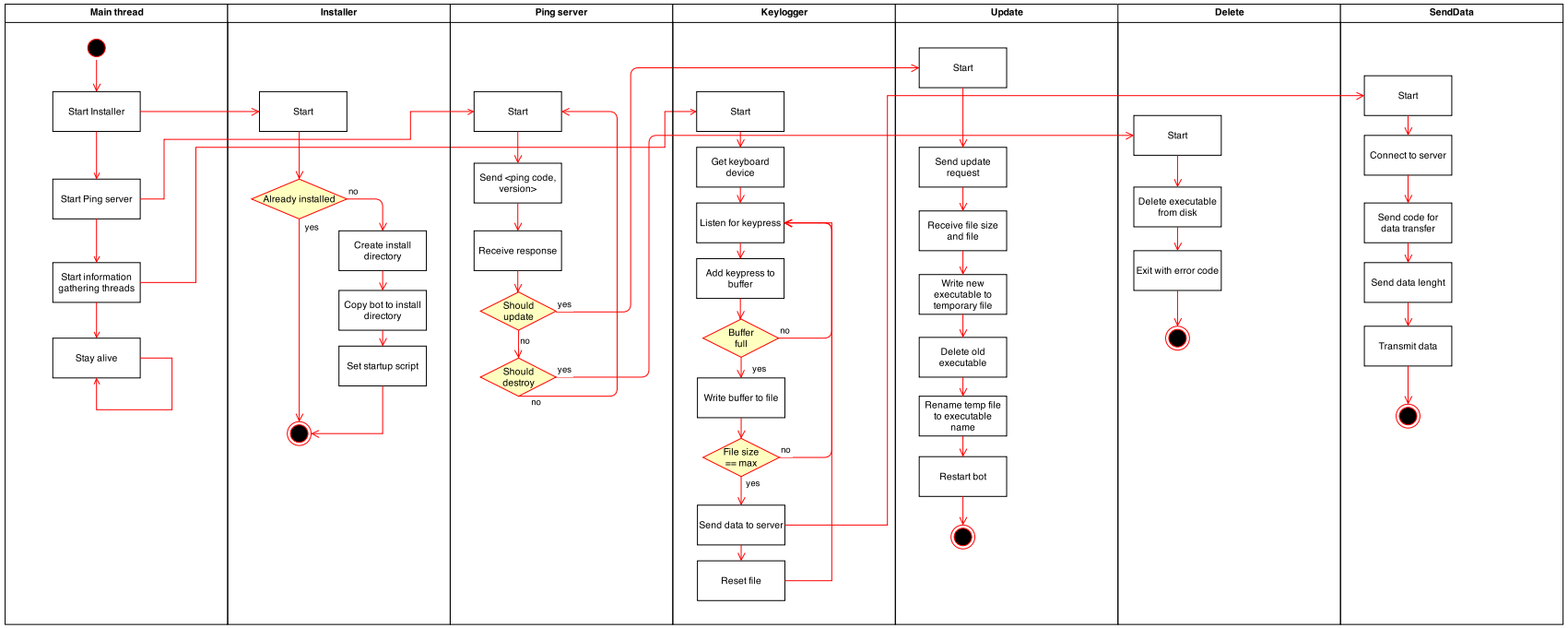

Figure 5.9: Full program flow of bot

5.3.2 Class design

This section highlights interesting design decisions made in the implementation of the bot. These decisions were made to allow for efficiency, re-usability an to reduce code duplication. This section discusses three design decisions, namely multithreading, an abstract base class shared by all information gathering modules and an abstract class shared by all modules communicating with the server.

Multi-threading

To make use of the built-in functionalities of OpenP2P, all classes that has to run on a separate thread extends the OpenP2P::Runnable class. This allows threads to be started by passing an instance of the Runnable to OpenpP2P::Thread. See Figure 5.10 for the UML diagram.

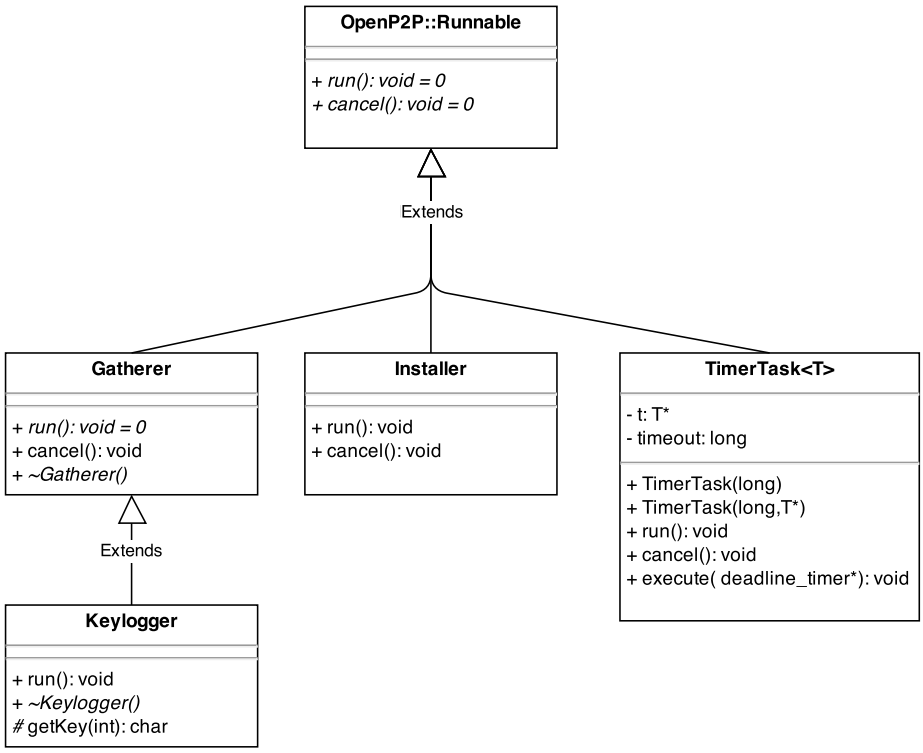

Figure 5.10: Classes inheriting from OpenP2P::Runnable

Information gathering

Some functionality is common between all information gathering processes. Implementing this more than once is redundant and may make bug fixing unnecessarily complicated. To mitigate this problem, the author implemented a Gatherer class, inheriting from OpenP2P::Runnable. This class is pure virtual and contains functionality shared by all information gathering modules.

Communication with the server

All communication with the Command-and-Control server starts with a request code and the version of the bot. In order to reduce code duplication, the author implemented a TwoWayTransmit

class. This class is a pure virtual class that makes a connection to the server, transmits the request code and the bot version, and then hands over control to the child class. See Figure 5.11 for classes inheriting from TwoWayTransmit.

This section discussed implementation details specific to the client component of the client-server botnet. The next section focuses on the server.

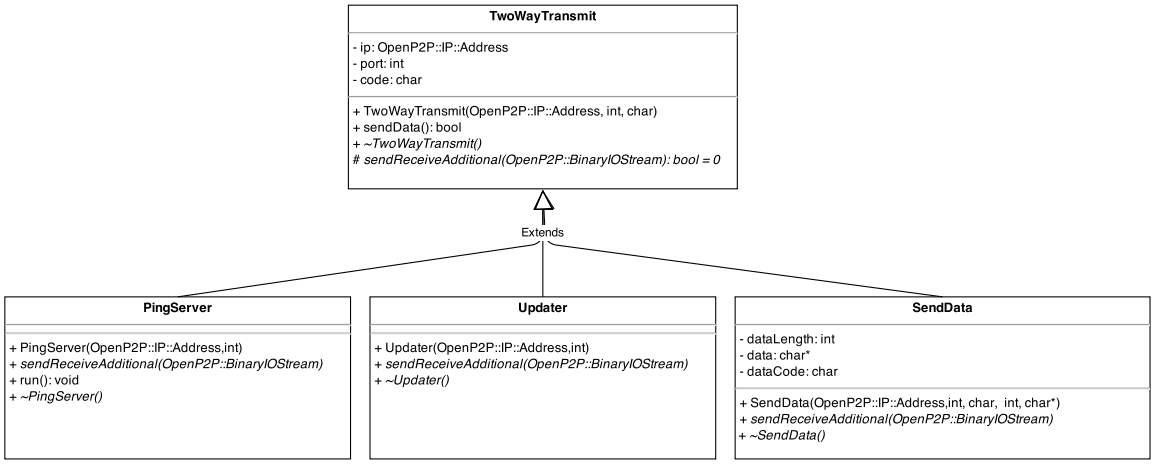

Figure 5.11: Classes inheriting from TwoWayTransmit

5.4 Command-and-Control server

The Command-and-Control server receives data from and sends commands to the bots in the botnet. This section first discusses the program flow of the botnet and then goes into more detail on the design pattern used to implement the request handler.

5.4.1 Program flow

Figure 5.12 shows the program flow of the server. It listens for connections and passes the incoming connection to a separate thread to handle. After the thread takes ownership of the connection, it reads the bot's request and the bot version.

If a bot has pinged the server, the server can either tell the bot to update, delete or simply acknowledge the request. If a bot has data to report, the server receives it and logs it. Finally if a bot needs to download an update, the server replies with the update.

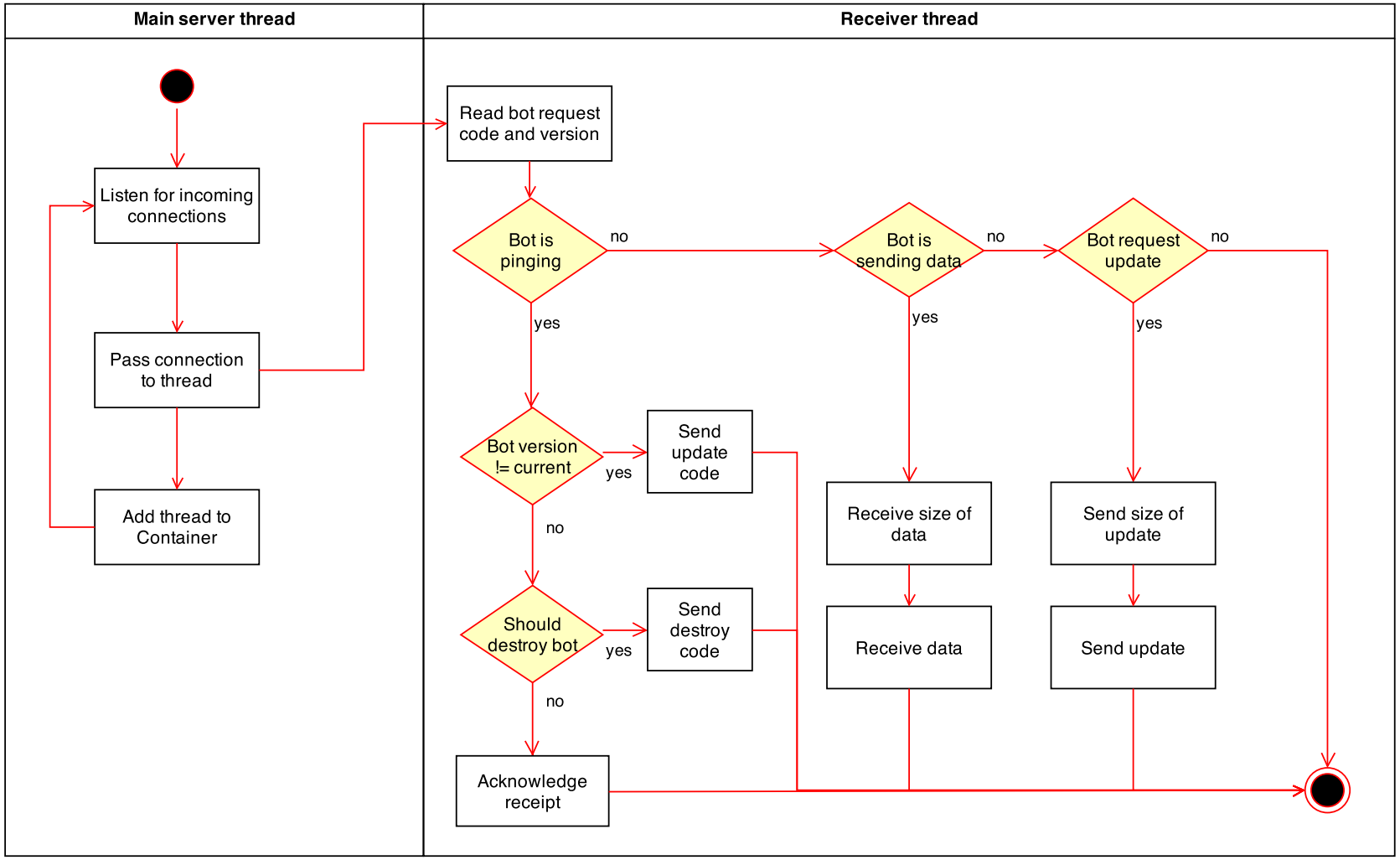

Figure 5.12: Overview of Command-and-Control server program flow

This section discussed the program flow of the Command-and-Control server. The next discuss specific implementation decisions for the server.

5.4.2 Chain of Responsibility

The implementation of the server is interesting because of the number of potential bots it must be able to handle. This section discusses the Chain of Responsibility pattern it uses to handle incoming requests.

The classes responsible for handling the incoming bot requests are implemented as the Chain of Responsibility design pattern. Gamma et al [11] describes the intent of the Chain of Responsibility design pattern as follows:

> Avoid coupling the sender of the request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

There are various reasons for using the Chain of Responsibility design pattern [11]:

- A request can be handled by more than one object, and the handler is not known beforehand
- A request should be issued to several objects without specifying the handler explicitly
- The set of objects that can handle a request can be dynamic

The Command-and-Control request handlers apply to two of these reasons, namely that there are multiple handlers and the handlers cannot be known or specified beforehand.

The UML diagram of the request handlers can be seen in Figure 5.13. The Handler class defines the interface for handling requests. It also implements functionality that is be shared by all concrete implementations, namely the handle function, that checks if the implementation should handle the request. If it should, it calls the process function to handle the request. If it shouldn't, the next handler in the chain is called.

The chain of handlers is set up in the Receiver object's constructor, in the following order:
ping handler $->$ data receiver handler $->$ update handler
The interaction diagram in Figure 5.14 shows the process a request received by the server goes through.

5.5 Conclusion

This chapter introduced the prototype develop for the problem proposed in chapter 1. It discussed the bot, the Command-and-Control server and the interaction between two, proposing a program flow as well as some implementation specifics. The next chapter critically evaluates the proposed solution.

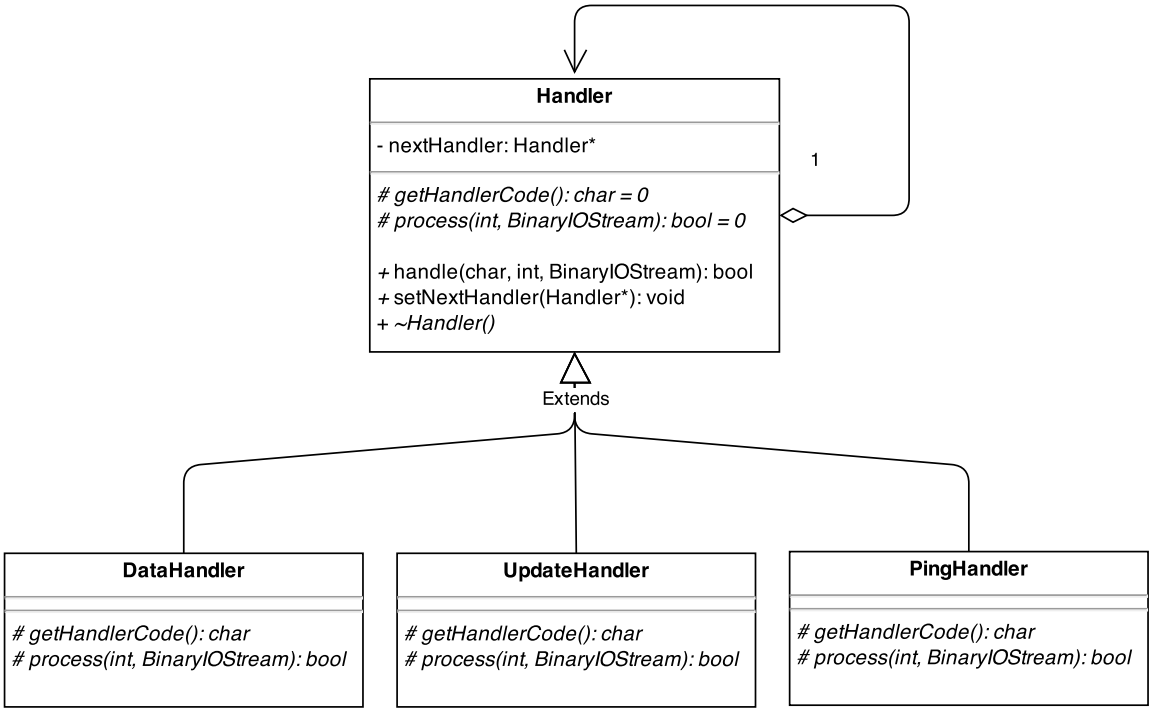

Figure 5.13: UML diagram of request handlers

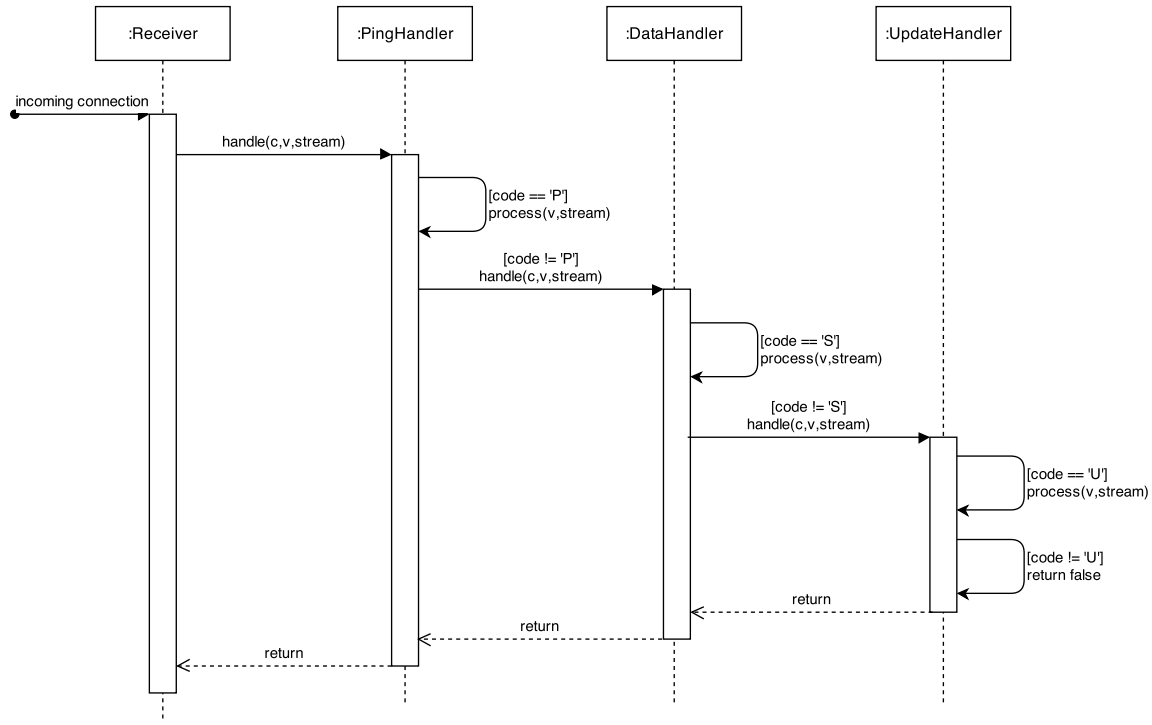

Figure 5.14: Interaction diagram of request handlers

6. Critical evaluation

6.1 Introduction

This dissertation proposed a solution to the problem that organisations cannot easily monitor the usage of their private cloud environments. This makes it easy for both employees and third parties to exploit the resources without serious repercussions.

This chapter critically evaluates the work done in this dissertation and discusses advantages, disadvantages and possible uses. It also examines the ways in which this solution differs from a true cyber immune system. It also gives an overview of privacy issues and suggests solutions for the user privacy problem.

6.2 Advantages

There are several advantages to using a botnet for monitoring cloud environments, especially systems as tightly controlled as a private cloud environment. Three advantages, namely self propagation, extensibility and the fact that the data is in one place, is discussed below.

6.2.1 Self-propagating

Since the botnet propagates by itself, there is no extra burden placed on the cloud administrators to install additional tools. This also means that there are no financial burden in the company to employ more people to operate the solution.

6.2.2 Data centralized

Since all the bots communicate with one Command-and-Control server, all data is gathered in one place. Connecting the server to big data solution allows analysis on all data at the same time. This can point out patterns that would otherwise stay hidden.

6.2.3 Extensibility

The implementation of this system allows for easy extensibility. This mean that the organisation can easily add additional information gathering modules as it is needed. Once the system is updated, the bots will automatically download and install the new version of the executable. It is also possible to create additional modules that is only deployed to certain machines, by modifying the server-side update module.

In addition to the advantages listed above, this approach has several disadvantages as well. These disadvantages are discussed in the next section.

6.3 Disadvantages

These disadvantages are caused by the very nature of the botnet, as well as the amount of data it collects. The next section discusses three of the disadvantages of using a botnet, namely gaps in bot coverage, the abundance of data and the bandwidth used by the botnet.

6.3.1 No guarantee of 100% coverage

Since the spread of the botnet depends on its propagation mechanism, it is possible that the bot will not be installed on all the virtual machines in the cloud environment. This can lead to a gap in coverage. This problem can be mitigated by modifying the hypervisor to install the botnet on every virtual machine starting up as opposed to allowing the botnet to propagate by itself. However, this means that the organisation will need to modify the cloud infrastructure.

6.3.2 Too much data

If an organisation does not have a proper data solution in place, the data gathered by the botnet may quickly become overwhelming and unusable. This means that the botnet will increase workload on systems staff and may cause the solution to become unusable in the long run.

6.3.3 Network traffic

If a private cloud has a large number of virtual machine, the botnet traffic from all those machines may impact the network speed between components negatively. This may cause cloud applications to become slow or unusable which will negatively impact employee productivity.

Even with the disadvantages discussed above, some organisations may decide to use this solution. The next section will discuss possible usages of this solution.

6.4 Usage

This solution, a botnet running on virtual machines, were built to be run on a private cloud maintained by (relatively) small organisations. However, it can also be used for various other purposes.

6.4.1 Public clouds

This solution can be deployed just as easily onto a public cloud to monitor virtual machine usage. In this case, however, the privacy concerns discussed below must be resolved with users. In case of a public cloud, a good big data solution is even more essential. It may also be necessary to modify the solution to efficiently run on a highly distributed system.

6.4.2 Data centres

A traditional data centre contains a set of servers (often mainframes) with virtual machines running on them. The traditional data centre is not cloud computing, since users must request new virtual machines from administrators and resources are not automatically allocated and adjusted. The botnet can easily be deployed in a data centre - only the infection vector will differ, since these machines tend to be more locked down.

If an organisation should decide to deploy the solution onto their private cloud, they first need to address the privacy concerns discussed in the next section.

6.5 Privacy concerns

This solution raises many privacy concerns. The purpose of this botnet is to monitor users and send data back to a central server *without the users' knowledge*. This means that the system may capture usernames, passwords and other personal information.

The most problematic of these are **passwords**, since a username/password combination allows full access to all system the user has access to. In the context of a private cloud, other pieces of personal information are not that sensitive, since all data stays within the company's data centres. There are a couple of solutions to the problem of logging passwords, which are discussed next.

6.5.1 Agreement

The easiest solution is to have the user agree to being monitored. This agreement is a condition of employment in many large companies already. However, this solution will not work for public clouds, where users, rightly, expect their intellectual property to be safe. This also does not solve the problem of usernames and passwords being captured, which is a security concern.

6.5.2 Anonymise data

Key bits of data can be anonymised before sending it to the server. This will erase any privacy concerns. However, it also defies the purpose of the botnet, which is to monitor cloud usage to prevent specific employees from abusing it.

6.5.3 Prevent the logging of some data

This solution involves marking some entry fields as unmonitored. The cloud administrators will mark some input fields as such and the botnet will then not log those fields. This solution is very hard to implement and will require that organisations employ specialized personnel. A rogue administrator may also mark unnecessary fields as unmonitored, hiding evidence of unauthorized activity.

6.5.4 Single Sign-on

Single Sign-on (SSO) is a mechanism where a user signs into one system and then switch to another system. The sign-on system passes a time-bound token to the other system. If SSO is used, the sign-on server can have a separate monitoring system and then pass a token to the virtual machine that is monitored by the botnet. This will take care of the problem of logging passwords, while still being able to link system activity to a specific user.

Another mechanism often used to ensure user safety is a cyber immune system. However, the botnet cannot be seen as a cyber immune system, as discussed in the next section.

6.6 Difference from cyber immune system

In chapter 2, the author briefly mentioned cyber immune systems. These cyber immune systems uses the same exploit as the targeted virus to get onto a system. After it is on a system, it removes the virus, spread itself, patches the exploits and finally removes itself from the system [27].

These four steps can be converted into four criteria that a piece of software must adhere to before it can be called a cyber immune system:

- targets a specific virus
- uses same propagation mechanism as virus
- patches weakness that allows it to propagate
- leaves no trace of the virus or itself

Comparing the solution presented in this dissertation to the criteria above makes it clear that this solution is not a cyber immune system, since it does not remove other viruses or fixes the weakness that allows it to propagate.

6.7 Conclusion

This chapter provided a critical evaluation of the solution proposed as an answer to the problem statement in chapter 1. While this is an interesting solution, the author does not believe that it is an efficient or commercially viable solution.

7. Conclusion

7.1 Summary

As previously stated, companies have a problem finding cost-effective solutions to monitor their private clouds. Insufficient controls on a system makes it simple for employees to abuse the system and use it for unauthorised activities, which in turn can cause both financial and reputation damage for the organisation operating the private cloud. Since the cloud architecture cannot be changed this means that a solution needs to be deployed at virtual machine level.

The solution for this problem proposed in this mini-dissertation is to deploy a botnet on to the virtual machines in the cloud. To achieve this goal, the author did literature studies on both botnets and clouds and then proposed a modular Command-and-Control botnet that is to run on the virtual machines of the private cloud.

This botnet has a central Command-and-Control server that receives the information the bots gather, as well as being responsible for managing the lifecycles of the bots. It is implemented in such a way that it is easy to extend it should new functionality be required.

The bots that is deployed to each client is responsible for gathering usage data and transmitting it back to the server. The prototype contained only a keylogger gathering module, but it is easy to extend the bot to include more modules.

The solution, as it stands, can be deployed to an organisation's private cloud and start logging keystrokes. However, many improvements can still be made to the solution. These are discussed in the next section.

7.2 Future work

This solution is very basic, and there is a number of opportunities for future work. These improvements fall into various Computer Science domains and are discussed in the following sections.

7.2.1 Propagation mechanisms

The author briefly proposed a propagation mechanism for the bots, namely installing it on a virtual machine at runtime. This mechanism, however, may require changes to the hypervisor of the cloud solution. Further work is needed into possible propagation mechanisms for a cloud-based botnet.

7.2.2 Data handling

The amount of data received from bots in the botnet depends on the number of virtual machines. Since virtual machines are easy to create, the solution will most likely have to deal with a large amount of data. A efficient data solution and data mining algorithms will be needed to make sense of the data.

7.2.3 Cloud monitoring

The solution proposed in this paper to monitor private clouds is that if a botnet. However, there may exist other, more efficient and scalable solutions to solve the same problem.

7.2.4 Alternative usages

Solutions similar to the one proposed in this dissertation may be applied to other problems or domains to solve them effectively.

7.3 Final statement

The research field of cloud computing, and more specifically, privacy and security in cloud computing, is still young. This dissertation suggested a solution to one of the problems being researched, namely how to prevent users from abusing cloud resources. However, there are many other problems being researched. Some avenues for research was suggested in the previous section.

Bibliography

[1] About cmake. `http://www.cmake.org/overview/`. Accessed: 14 October 2014.

[2] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 41–52, New York, NY, USA, 2006. ACM.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[4] M. Bailey, E. Cooke, F. Jahanian, Yunjing Xu, and M. Karir. A survey of botnet technology and defenses. In *Conference For Homeland Security, 2009. CATCH '09. Cybersecurity Applications Technology*, pages 299–304, March 2009.

[5] Boost library. `http://www.boost.org`. Accessed: 13 October 2014.

[6] D. Dagon, Guofei Gu, C.P. Lee, and Wenke Lee. A taxonomy of botnet structures. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 325–339, Dec 2007.

[7] T. Dillon, Chen Wu, and E. Chang. Cloud computing: Issues and challenges. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27–33, April 2010.

[8] Easylogging++. `https://github.com/easylogging/easyloggingpp`. Accessed: 13 October 2014.

[9] M. Feily, A. Shahrestani, and S. Ramadass. A survey of botnet and botnet detection. In *Emerging Security Information, Systems and Technologies, 2009. SECURWARE '09. Third International Conference on*, pages 268–273, June 2009.

[10] R Ferguson. The history of the botnet–part 3, 2010.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[12] Gcc options to request or suppress warnings. `https://gcc.gnu.org/onlinedocs/gcc-4.3.6/gcc/Warning-Options.html`. Accessed: 15 October 2014.

[13] Julian B Grizzard, Vikram Sharma, Chris Nunnery, Brent ByungHoon Kang, and David Dagon. Peer-to-peer botnets: Overview and case study. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding tnets*, pages 1–1, 2007.

[14] R.L. Grossman. The case for cloud computing. *IT Professional*, 11(2):23–27, March 2009.

[15] Xuefeng Li, Haixin Duan, Wu Liu, and Jianping Wu. Understanding the construction mechanism of botnets. In *Ubiquitous, Autonomic and Trusted Computing, 2009. UIC-ATC '09. Symposia and Workshops on*, pages 508–512, July 2009.

[16] Linux kernel documentation: input event codes. `https://www.kernel.org/doc/Documentation/input/event-codes.txt`. Accessed: 14 October 2014.

[17] P Mell and T Grance. The nist definition of cloud computing. `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`.

[18] Julia Narvaez, Barbara Endicott-Popovsky, C. Seifert, Chiraag Aval, and D.A. Frincke. Drive-by-downloads. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–10, Jan 2010.

[19] Open group publications: fnctl.h specification. `http://pubs.opengroup.org/onlinepubs/009695399/basedefs/fcntl.h.html`. Accessed: 14 October 2014.

[20] Open group publications: sys/stat.h specification. `http://pubs.opengroup.org/onlinepubs/009695399/basedefs/sys/stat.h.html`. Accessed: 14 October 2014.

[21] Open group publications: sys/types.h specification. `http://pubs.opengroup.org/onlinepubs/009695399/basedefs/sys/types.h.html`. Accessed: 14 October 2014.

[22] Open group publications: unistd.h specification. `http://pubs.opengroup.org/onlinepubs/009695399/basedefs/unistd.h.html`. Accessed: 14 October 2014.

[23] Openp2p. `http://openp2p.org/Main_Page`. Accessed: 24 March 2014.

[24] George Pallis. Cloud computing: The new frontier of internet computing. *IEEE Internet Computing*, 14(5), 2010.

[25] Sam Stover, Dave Dittrich, John Hernandez, and Sven Dietrich. Analysis of the storm and nugache trojans: P2p is here. *USENIX; login*, 32(6):18–27, 2007.

[26] Tim Thornburgh. Social engineering: The "dark art". In *Proceedings of the 1st Annual Conference on Information Security Curriculum Development*, InfoSecCD '04, pages 133–135, New York, NY, USA, 2004. ACM.

[27] Ping Wang, S. Sparks, and C.C. Zou. An advanced hybrid peer-to-peer botnet. *Dependable and Secure Computing, IEEE Transactions on*, 7(2):113–127, April 2010.

[28] Ping Wang, Lei Wu, B. Aslam, and C.C. Zou. A systematic study on peer-to-peer botnets. In *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*, pages 1–8, Aug 2009.

[29] Zhaosheng Zhu, Guohan Lu, Yan Chen, Zhi Fu, P. Roberts, and Keesook Han. Botnet research survey. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 967–972, July 2008.

[30] C.C. Zou and R. Cunningham. Honeypot-aware advanced botnet construction and maintenance. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 199–208, June 2006.

A. Compiling and running

This appendix describes the compilation and running process of the prototype. This includes an explanation of the preprocessor defines used and the compiler flags.

A.1 Compiling

This project makes use of the *CMake* tool to build both the bot and the server. CMake is an open-source, cross-platform build tool that allows extensibility and supports complex project structures [1].

A.1.1 Preprocessor defines

CMake allows the user to set C++ preprocessor definitions as part of the build process. These definitions can be accessed in the same way as normal variables during runtime. Table A.1 gives a list and short description of each preprocessor define injected by CMake.

OPENP2P_VERSION	Version of the OpenP2P library
KILROY_VERSION	The current version of the **bot**, used to determine if an update is necessary.
HEADERS	Location of header files
OPENP2P_ROOT	Root location of OpenP2P source files
_ELPP_THREAD_SAFE	Flag to indicate to the easylogging++ library to log in thread safe mode.
_ELPP_DEFAULT_LOG_FILE	Pass the default log file location to the easylogging++ library
__STDC_LIMIT_MACROS __STDC_CONSTANT_MACROS	These two constants have to be defined to allow C++ programs to make used of the **stdint.h** header.

Table A.1: Preprocessor defines

A.2 Compiler flags

In addition to passing in preprocessor defines, the build process also passes a number of flags to the compiler. These are listed in Table A.2.

-pedantic	The compiler will issue all warnings required by the C++ ISO standard and will reject non-compliant programs [12].
-Wall	This flag will enable a number of warnings about questionable code. A full list of the warnings that are enabled can be found in gcc warning flag documentation [12].
-Wextra	This flags adds a number of warning flags that is not covered by the -Wall option [12].
-Wpointer-arith	Warns when **sizeof** is called on a void pointer or a function pointer, since the compiler always assigns these a value of 1 [12].
-Wcast-align	Warns when casting a pointer would change the alignment of the target [12].
-Wwrite-strings	Warns about trying to cast string literals to a **char** $*$ type [12].
-std=c++11	Compile against the C++ 11 standard and libraries.

Table A.2: Compiler flags

A.3 Compiling

CMake generates all the files needed to build the project. To build the project, the user first needs to run CMake to generate the build files and after that run *make* to build the project.

```
cd botnet-code # go to root directory of source code
mkdir build    # create a directory for the build files...
cd build       # ...and swith into that directory
cmake ..       # generate build files using cmake
make           # use make to build the executables
```

A.4 Running

The *make* process generates the final executable files. These files can be found in the build/main directory inside the source root. The files are named KILROY and KILROY_CONTROL respectively. The KILROY_CONTROL program is the Command-and-Control server and can be run by calling

```
./KILROY_CONTROL
```

The KILROY program is the bot.To allow the bot to install itself on a host, it needs to be run with root permissions. After the bot has been installed, it will start up automatically when the host starts up. To run the bot for the first time, execute

```
sudo ./KILROY
```

B. Functionality

This appendix highlights some technical details of the solution. Like the previous appendix, this one is not needed to understand the solution, but may be useful if the reader requires a more comprehensive understanding.

This appendix is divided into three sections: functionality shared by the bot and the server, bot-specific implementations and finally the server implementations.

B.1 Shared functionality

Some pieces of code is shared by the bot and the server. This section gives an overview of the shared pieces. The bot and the server can make use of the same code because of how the compilation process works (see appendix A).

B.1.1 Container

The Container class is adapted from a class used internally by the OpenP2P library. It is used to ensure that resources from objects that are terminated unexpectedly are still released correctly.

```
template <typename T>
class Container {
private:
        std::vector<T*> data;
public:
    ~Container();
    T* add(T* t)
}
```

Internally, the Container pushes the pointers it receives in the add method into a standard C++ vector. When the Container's destructor gets called, it cycles through the vector of pointers and calls **delete** on each pointer.

This implementation works because of a very specific behaviour of C++. Since the container itself is located on the *stack*, it gets destroyed correctly even if the application terminates unexpectedly. However, this also means that adding a pointer to an object on the stack to this vector causes strange behaviour, since deleting objects on the stack can cause memory to be freed twice.

B.1.2 Exec method

```
std::string exec(char* cmd);
```

The exec method takes in a operating system command as a string and executes it, returning the result of the call to the caller. This method is necessary because the bot uses system commands for various pieces of functionality.

B.2 Bot

This section gives a short overview of some of the interesting implementation details of the bot.

B.2.1 TimerTask

The TimerTask class (TimerTask.h) is responsible for firing of a task at a fixed interval. To make it as extensible as possible, it is implemented as a template class, with the template parameter T the task that needs to be executed. It also extends OpenP2P::Runnable to allow callers to run it in a separate thread.

```
template <class T>
class TimerTask : public OpenP2P::Runnable
```

The TimerTask only places two restrictions on the parameter T. The first is that T must implement a method with the signature **void** run(); that contains the logic that needs to be executed when the timer runs out. The other is that T must implemented a default constructor if the TimerTask(**long**) constructor is used to create the TimerTask.

B.2.2 getCnCDetails method

```
Server getCnCDetails();
```

The getCnCDetails method returns the details of the Command-and-Control server. Using this method means that the details of the Command-and-Control server needs to be maintained in only one place. The Server object returned by the method is a **struct** containing the server's IP and port.

```
struct Server {
OpenP2P::IP::Address ip;
int port;
};
```

B.2.3 getExecPath method

```
std::string getExecPath();
```

The getExecPath method gets the full path of the bot on disk, by making use of system calls.

B.3 Command-and-Control server

The main areas of interest in the implementation of the Command-and-Control server, is in the incoming connection listener and the request handlers, as discussed below.

B.3.1 Incoming connection listener

The main class of the Command-and-Control server listens on a specific port for incoming connections. When a connection request is received, the server creates a new thread to handle all interaction with the incoming connection. The thread is then added to a Container, as describe in appendix B.1.1, to manage the life cycle of the thread object.

B.3.2 Handlers

The majority of the Command-and-Control server is implemented as a Chain of Responsibility, as describes in chapter 5.

The base handler handles the process of checking if the specific implementation should be processing the request. If the code of the request matches that of the current handler, the process method of the current handler is called. Otherwise, the request is passed to the next handler in the chain.

```
bool Handler::handle(char code, int version, OpenP2P::BinaryIOStream
stream){
if (code == getHandlerCode()) {
process(version, stream);
} else if (nextHandler != 0) {
return nextHandler->handle(code, version, stream);
}
return false;
}
```