PeerBot

A Peer-to-Peer Botnet

Jsca0 - 20 April 2022

Word Count: 9321 (excluding table of contents and references)

BSc Computing Project Report, Birkbeck College,
University of London, 2022

This report is the result of my own work except where explicitly stated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

Peer-to-peer networks are being used by modern, malicious botnets to avoid the take-down methods used against centralised models. This project aims to explore the advantages peer-to-peer models have over client-server architecture. The purpose of this project is design, develop, test, and evaluate a peer-to-peer botnet.

Contents

1. Introduction

The purpose of this project is to build and evaluate a peer-to-peer botnet. The botnet is a proof of concept, intended to demonstrate how peer-to-peer networking can be applied to botnet design.

Peer-to-peer describes a network where each member both shares and consumes some resource(s). This is in contrast to the client-server model, where a participant either consumes or supplies.

A malicious botnet is a weaponised computer network. A cyber criminal sends commands to a web of infected machines in order to perpetrate some attack. Examining the network architecture of a botnet can provide some insight into how the botnet could be disrupted or destroyed. For example, a botnet that solely relies upon the client-server model may be vulnerable to ‘decapitation’, whereby seizing the botnet’s server renders it unable to send commands. This means that, although the botnet remains relatively intact, there is no longer a way to control it. Decapitation would be made far more difficult if instead, the botnet were to rely upon peer-to-peer networking. This is because in peer-to-peer networks, each node behaves as both a client and a server, so there is no single point to target in order to break the entire network.

Given the current reliance on computer infrastructure, botnets pose a significant cyber security threat. In order to better secure systems against cyber threats, it is important to understand the means and mechanisms with which malicious software is developed. In designing a peer-to-peer botnet, this project explores how peer-to-peer botnets might evade some of the detection and decapitation tactics used against centralised models. In evaluating the peer-to-peer botnet, this project investigates different tactics employed to undermine the effectiveness of peer-to-peer botnets.

1.1 Objectives and aims

This project attempts to achieve the following:

- Demonstrate how Peer-to-peer architectures can be applied to botnet design, to provide better resistance against the detection and decapitation threats that challenge centralised models.
- Build and test a peer-to-peer botnet.
- Build and test a command and control mechanism.
- Demonstrate a bot joining the botnet and successfully executing a command from command and control.
- Evaluate the robustness of the botnet.

1.2 Motivation

The main motivation behind this project is to research both peer-to-peer architecture and botnets. To investigate the tools and methods cyber security professionals use to combat malicious botnets.

The goal is to design a peer-to-peer botnet that exemplifies the use of peer-to-peer networks in evading common take down methods. The botnet will not include malware to infect target machines or a means to replicate itself.

1.3 Report Structure

The present report has the following structure:

Section 2 carries out a brief review of peer-to-peer networking and botnets. This sections briefly describes the web technology used by

Section 3 outlines the project’s requirements and specifications.

Section 4 describes and explains the designs and design decisions.

Section 5 outlines the implementation and testing process.

Section 6 evaluates the botnet design. This section discusses the strengths and weaknesses and how the botnet might be disrupted, detected and destroyed.

Section 7 reflects upon the project.

Section 8 provides a list of references.

Section 9 consists of appendices

2. Context

2.1 Background on Peer-to-Peer Networks

A peer-to-peer (P2P) network is a computer network in which a series of interconnected 'peers' or 'nodes' share resources (content, processing power, disk storage). The resources are shared by direct exchange, rather than going through central coordination servers. Figure 2.1 shows and example of a P2P network.

In contrast to peer-to-peer networking is the more traditional client-server model. In P2P, peers are both suppliers and consumers, whereas the client-server model divides supply and consumption between 'clients' and 'servers'. A client requests a resource from a server that provides it. Figure 2.2 shows an example of a network based on the client-server model.

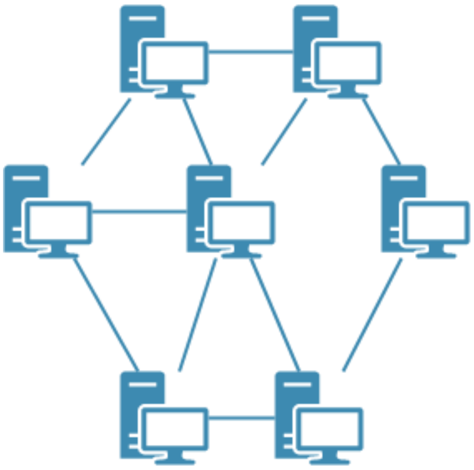


Figure 2.1 A P2P network

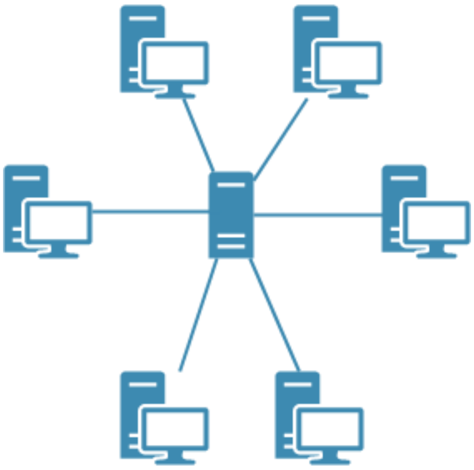


Figure 2.2 A network based on the client-server model

Table 2.1 outlines some of the differences between peer-to-peer and client-sever networks.

Table 2.1 Comparing P2P and Client-Server networks

P2P	***Client-Server***
Nodes consume **and** supply	Nodes consume **or** supply
A decentralised network	A centralised network
Reliable. (A single node failure will not disrupt the others)	Single point of failure. (A server failure will disrupt the functioning of all clients)
Content availability increases when more nodes join the network	Access time for a service may be slower as more nodes join the network
Hard to manage	Easy to manage
Inexpensive	Expensive
Less stable	More stable
Applications include, file-sharing (e.g. BitTorrent), digital cryptocurrencies (e.g. Bitcoin)	Applications include, email, World Wide Web

2.1.1 Peer-to-Peer Architecture

An overlay network is a computer network that is layered on top of another network. Distributed systems such as peer-to-peer networks and client–server applications are examples of overlay networks, because their nodes run on top of the internet.

Nodes in an overlay network are connected by logical links. Connections between logical links correspond to physical links between nodes in the network beneath the overlay. The set of nodes contained in an overlay network form a subset of nodes in the underlying physical network, so a single logical link could correspond to a path through multiple physical links. In peer-to-peer overlay networks, data is exchanged over an underlying TCP/IP network, but logical links allow peers to communicate with each other directly.

Peer-to-peer overlay network structures are classified according to how nodes are linked in the overlay network, and how resources are indexed and located. They can be grouped into the following types:

1. Structured
2. Unstructured
3. Hybrid

Unstructured

In unstructured networks the content location is unrelated to the network topology. Connections between nodes are formed randomly. When a peer is searching for a resource, the search query is usually flooded through the network until the resource is reached or the query expires.

Structured

In structured networks content is mapped to its location. A distributed hash table (DHT) is used to route traffic to the correct resource. For example, a peer looking for a particular file will query the DHT for the file's location. Structured networks allow the network to be searched more efficiently. (Section 4.3.2 discusses DHTs in more detail)

Hybrid

Hybrid networks are a combination of peer-to-peer and client-server models. Typically this means using a central server to help peers find each other.

2.2 Background on Botnets

A botnet is a collection of internet-connected devices, such as computers, smartphones or Internet of things (IoT) devices running one or more “bots”. The word “botnet” is a portmanteau of the words “robot” and “network”.

A “bot” is software application that runs automated tasks over the internet. Bots in a botnet maintain a communication link with a human handler, known as the “bot-master”. The bot-master is able to harness the power of multiple bots at once, in order to accomplish some task.

The term “botnet” is often used in the negative sense. Cyber criminals use malicious botnets to perform Distributed Denial of Service (DDOs) attacks, steal data, spread malware (malicious software); Depending on the botnets capabilities they may be used to illegally mine cryptocurrencies or perform some other computationally expensive work.

Malicious botnets are created using malware to penetrate and force a device into participating in the botnet. These kinds of botnets may also be equipped with the ability to replicate themselves and infect other machines.

However, botnets do not have to be malicious. Non-malicious botnets are are voluntary and require user permission to add a computer, they are often used for scientific purposes.

2.3 Background on Peer-to-Peer Botnets

A peer-to-peer botnet is a decentralised network of compromised devices, working together for the same bot-master. Peer-to-peer botnets are an emerging cyber security threat, due to their resilience against defence countermeasures.

2.3.1 Comparing centralised and P2P Botnets

A centralised botnet is a botnet using the client-server model. Bots in centralised botnets receive commands via a command and control (C&C) server. Figure 2.3 shows a centralised botnet.

Inter Relay Chat (IRC) based botnets are a popular type of centralised botnet. The bot-master uses an infected IRC server to send commands to a designated IRC channel. Infected clients can connect to the server and join the channel to receive the commands.

Contrastingly, peer-to-peer botnets have no centralised server. The bots connect to each other over a P2P network and each acts as both C&C server and client. Figure 2.4 shows a peer-to-peer botnet.

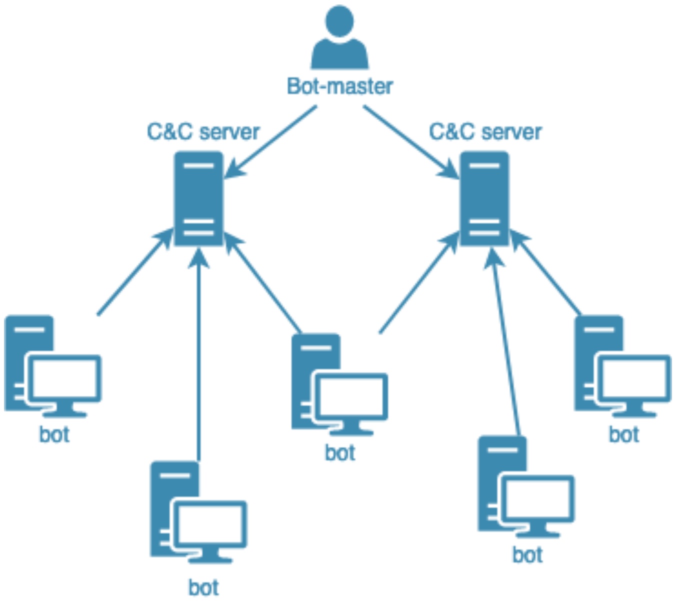


Figure 2.3 A centralised botnet

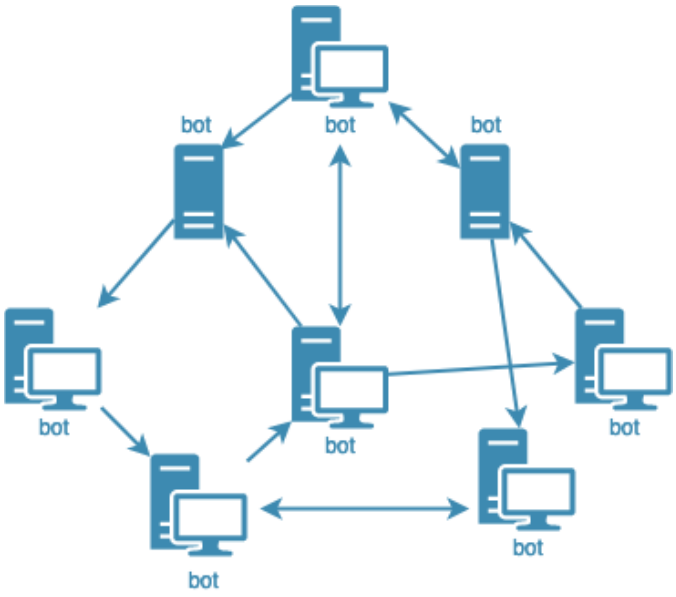


Figure 2.4 A P2P botnet

Typically, botnets use centralised command and control architecture. However, peer-to-peer structured botnets are becoming more common. Peer-to-peer botnets can be more difficult to detect and disrupt, because they don't use central servers. Table 2.2 outlines some comparisons between P2P and centralised botnets.

Table 2.2 comparing P2P and centralised botnets

P2P	*Centralised*
Bots are able to both distribute and receive commands	Bots receive commands from central C&C servers.
No single point to target in order to collapse the entire botnet.	Single point of failure. When central C&C servers are shutdown, bots lose contact with their bot-master
Resilient to high churn rates. Churn refers to the number of peers leaving and joining the botnet.	Less efficient as more bots join the botnet

3. Requirements and Specifications

3.1 Overview

As stated in the introduction, the purpose of this project is to build and evaluate a peer-to-peer botnet. This section will outline the main features of the botnet.

3.2 Requirements

Defining a software system's requirements is a key part of the software development process. The requirements outline important features and functionalities of the system being developed. The botnet's requirements are separated into the following sections:

- Functional requirements
- Non-functional requirements

The requirements have been organised according to the MoSCoW prioritisation technique. This technique manages a project's requirements by establishing a hierarchy of priorities. It is useful for projects where the scope is unclear. MoSCoW prioritisation divides requirements into four groups:

- "**Must** have": These are the essential features and behaviours. These requirements must be implemented for the project to be considered a success.
- "**Should** have": These requirements are important, but not vital. The project would still be acceptable without them.
- "**Could** have": These requirements are desirable but not necessary. They are less important than the "should have" ones.
- "**Won't** have": These features will not be implemented, but may be considered in the future.

3.2.1 Functional Requirements

The functional requirements describe the key features and behaviours. They have been considered during the design and development processes.

Table 3.1 Functional requirements and their MoSCoW priority

	Requirement	*MoSCoW*
A	The botnet must incorporate peer-to-peer architecture into its design.	**Must**
B	The botnet must have a command and control mechanism.	**Must**
C	The botnet must be able to send, receive and execute a few simple commands.	**Must**
D	Any results from the execution of a command should be made available to the bot-master*	**Should**
E	Only the bot-master should be able to send commands to the botnet.	**Should**
F	Each node should have a unique ID, identifying it within the botnet	**Should**
G	Only nodes with a valid ID should be allowed to join the botnet	**Should**
H	It could be possible for a command to be sent only to a specific node	**Could**
I	The botnet could maintain node state information e.g active/inactive	**Could**
J	The botnet could allow the bot-master to generate new nodes	**Could**
K	The bot-master can remove nodes	**Could**
L	The botnet won’t be self-replicating	**Won’t**
M	The botnet won’t include a mechanism for selecting bot candidates (no malware)	**Won’t**

**The ‘bot-master’ is the human owner of the botnet*

3.2.2 Non-Functional Requirements

The non-functional requirements specify the criteria for judging the operation of the botnet.

Table 3.2 Non-Functional requirements and their MoSCoW priority

	Requirement	*MoSCoW*
N	The identity of the bot-master must be protected.	**Must**
O	Nodes leaving and joining the network should cause a minimal amount of disruption	**Must**
P	Fault tolerant. The botnet must not have a single point of failure (such as with centralised botnets)	**Must**
Q	The botnet should be scalable.	**Should**
R	The command and control mechanism should be reliable.	**Should**
S	The botnet should be robust and resistant to common take down methods.	**Should**
T	The botnet should have high availability	**Should**

3.3 Use-Cases

The botnet's sole user is the 'bot-master'. The use case diagram for the botnet can be seen in Figure 3.1, it shows the interactions the bot-master might have with the botnet. The following use cases describe possible scenarios.

3.3.1 Sending commands

The Bot-Master accesses the botnet and sends the a command to all of the nodes in the botnet.

3.3.2 Receiving a response

The Bot-Master accesses the botnet and retrieves a response from a node with id "1234ABCD"

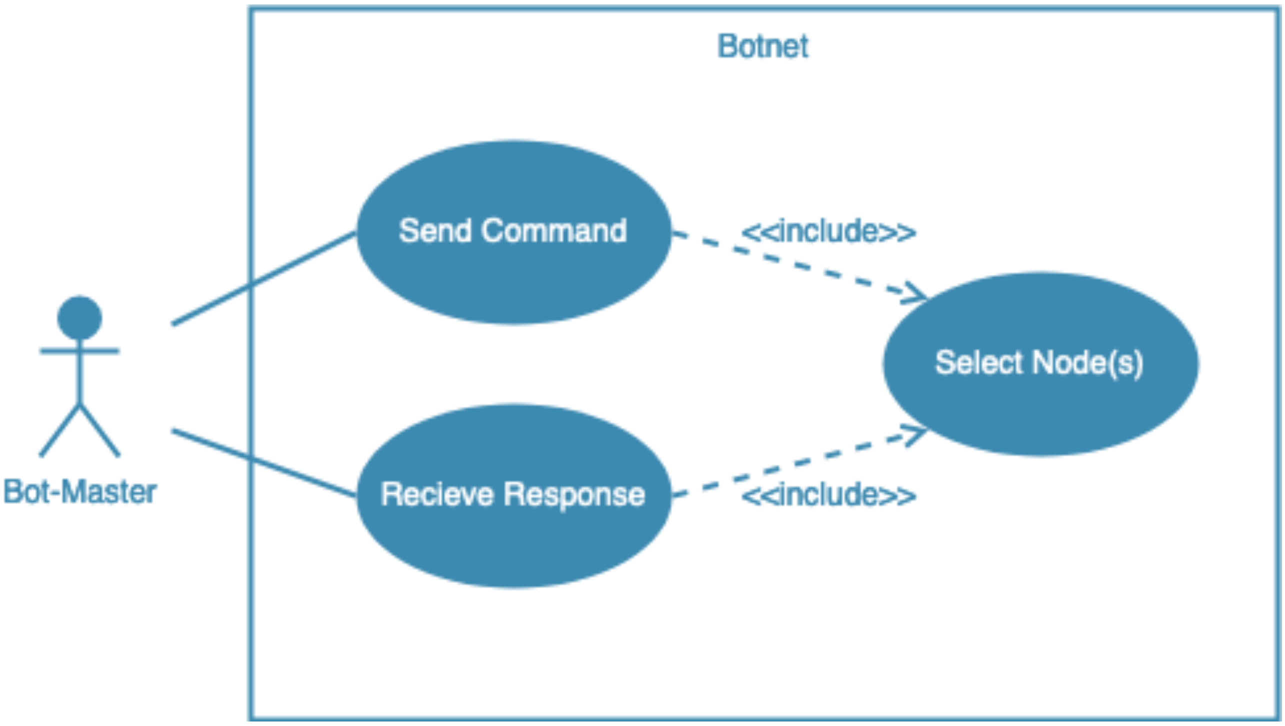


Figure 3.1 Use-case diagram for Bot-Master

4. Designs

4.1 Overview

This section will focus on the important structures involved in constructing the peer-to-peer botnet.

This section is split into the following four parts:

- The first part will describe the overall structure of the botnet.
- The second part will focus on designing the botnets's peer to peer networking architecture.
- The third part will outline the command and control architecture. This part will describe the means by which the bot-master communicates with the botnet.
- The final part will summarise the design of the botnet

4.2 How The Botnet is Structured

Simply described, a botnet is a collection of bots under the control of a bot-master. For clarity, the botnet presented in this project is named **PeerBot.** Figure 4.1 shows PeerBot is divided into two modules:

1. Commander
2. Worker

The **Commander** is the botnet's command and control mechanism, it is used by the 'bot-master' to send commands to the botnet.

The **Workers** are the 'bots' or clients, their job is to receive and execute commands from the Commander. The Activity Diagram for a Worker can be seen in Figure 4.2.

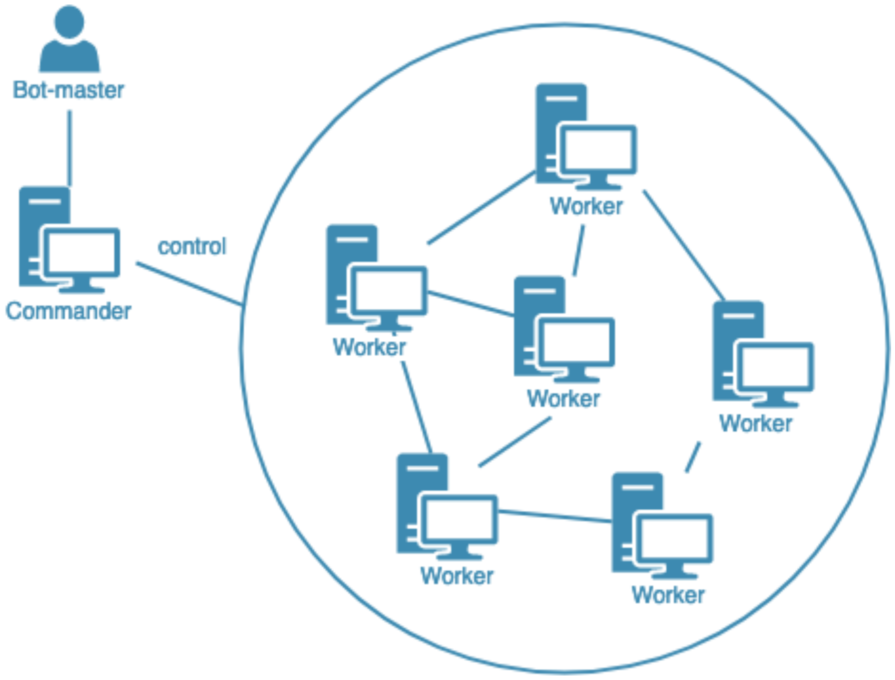


Figure 4.1 Illustration of PeerBot.

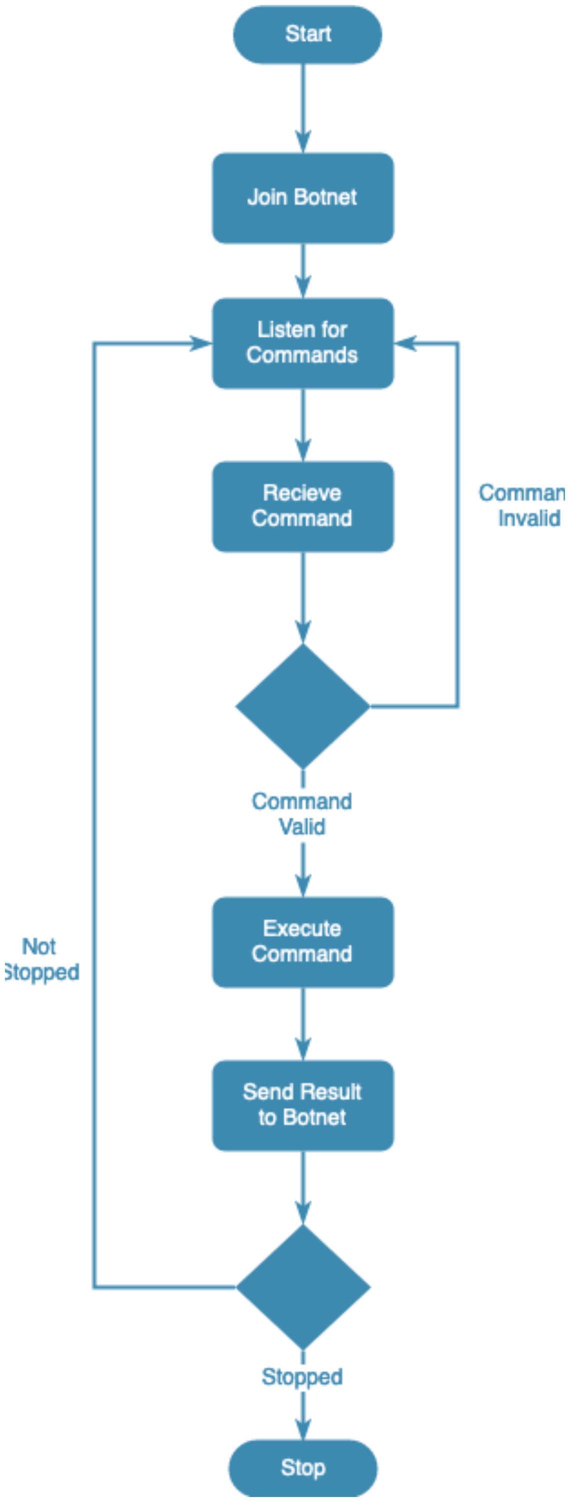


Figure 4.2 Activity diagram describing the workflow of a Worker

Figure 4.3 shows PeerBot's core components and how they are organised.

A Commander is able to communicate with a peer-to-peer network of Workers, in order send commands and receive responses. The next two sections will focus on describing how the peer-to-peer network is structured and the methods the Commander uses to communicate with the botnet.

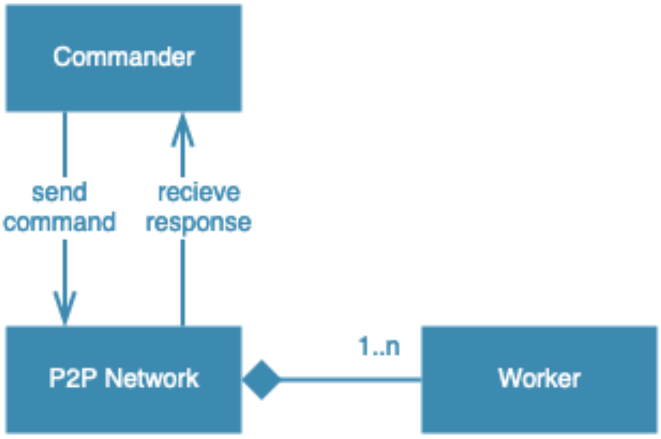


Figure 4.3 A simple structural model

4.3 The Peer-to-Peer Network

PeerBot uses a peer-to-peer network to enable communication between Workers and a Commander. Each Worker participates as a peer in the peer-to-peer network.

A peer-to-peer botnet could either be built upon an existing P2P network, or form its own. P2P Botnets can be sorted into the following categories:

- Parasite botnets — These botnets only infect vulnerable nodes in existing P2P networks. They are called “parasite botnets”, due to the parasitic relationship they have with a host network. Parasite botnets are limited by the number of vulnerable nodes contained in the host network.

- Leeching botnets — These botnets infect vulnerable nodes on the internet, then have those nodes join an existing P2P network. They are “leeching”, because they rely on the existing network.

- Bot-only botnets — These botnets operate on their own independent P2P networks. Bot-only botnets are typically more flexible than parasite and leeching, because they are free to implement their own P2P protocols.

PeerBot is a bot-only botnet, because it forms its own P2P network. This section will discuss how peer-to-peer networks are generally formed, and which of these general characteristics PeerBot implements. This section will also discuss the structure of PeerBot's P2P network and the P2P protocol PeerBot uses.

4.3.1 Forming A Peer-to-Peer Network

The first thing a Worker does is join the botnet (see figure 4.2). “Joining the botnet” means the Worker must become a peer in PeerBot’s P2P network. The method by which a new peer integrates itself into a P2P network is called the “bootstrap” procedure.

The bootstrap procedure can be broken down into two steps:

1. The new peer should acquire the contact information for at least one bootstrap node (existing peer in the network).

2. The new peer should contact each bootstrap node and request information to update the information it has about neighbouring peers. Now the network should be able to communicate with the new peer and the new peer the network.

Step one of the bootstrap procedure is to acquire an initial set of “bootstrap nodes”. There are two general ways to accomplish this:

1. The first method is to have an initial list of peers hard-coded in each Worker.

2. The second method is to have a the location of a shared web cache hard coded in each Worker. The Worker will use the web cache to find neighbouring peers and join the network.

PeerBot uses the first method. Workers are hard-coded with a list of bootstrap nodes, rather than using a central server to discover them. This is to reduce reliance on centralised structures, as they are easily targeted and used to disrupt botnet operations.

Step two of the bootstrap process is to contact the bootstrap nodes and exchange information. To do this, new Workers will need to use the network’s protocols. Protocols describe how peers organise themselves, communicate with each other, choose their neighbours, and locate resources.

As discussed in section 2.1.1 of this report, peer-to-peer networks can be categorised as structured, unstructured or hybrid. PeerBot's peer-to-peer network is structured. In comparison to unstructured, structured network protocols are more efficient. And unlike hybrid networks, structured networks don't depend on central coordination servers.

PeerBot's P2P protocol implements "Kademlia", a distributed hash table (DHT). The next sections explore these topics in more detail.

4.3.2 Distributed Hash Tables

A hash table is a collection of key-value pairs. A distributed hash table is a hash table spread across a distributed network. Structured peer-to-peer networks typically use a DHT to store a mapping from *keys to* node addresses. Peers can lookup the node address associated with a given *key.*

A DHT consists of two main components:

1. A keyspace
2. An overlay network

A **keyspace** is a set of all the *keys* available to the network. "*Keys* are unique identifiers which map to particular *values*, which in turn can be anything from addresses, to documents, to arbitrary data" (Stoica et al., 2001). Ownership of the keyspace is distributed across the entire network: different nodes own different keys.

For example a keyspace could be defined as the set of all possible values given by the SHA-1 hash function. Hashing something with the SHA-1 hash function always produces a 160-bit string. Any two files hashed with the SHA-1 are very unlikely to produce the same hash, unless they are exactly the same.

A peer wishing to share a file with the network, could obtain a key from the keyspace by hashing the file. The peer could then update the DHT, declaring itself owner of the new key. Now, any other other peer can lookup the file, using the key. The DHT will return the address of the original peer, where the file is stored.

Figure 4.4 demonstrates how a hash function is used to associate data with a unique key, that can then be used to locate the data within a distributed network.

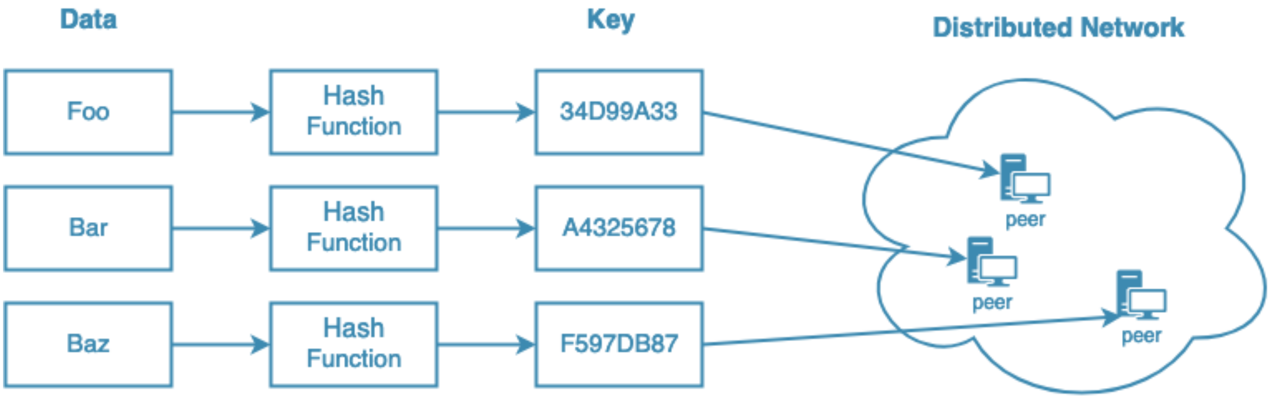


Figure 4.4 Distributed Hash Tables

A DHT allows a peer-to-peer network to operate without the need for a central coordination server. An **overlay network** allows peers to communicate with each other directly. The responsibility of maintaining the communication-links between peers is distributed among the peers themselves. Each peer maintains a set of key-value pairs, each of these pairs describes a link between the peer and a neighbouring peer. Peers and their neighbours are organised according to how close they are, in terms of keyspace distance.(section 4.3.3 discusses this in more detail).

When a peer wishes to find the address of a peer that owns a specific key, it uses the DHT to lookup the key. This generally involves the following steps:

1. The peer (peer1) searches its own list of neighbours for a neighbour (peer2) with a key closest to the target key. If there is no such neighbour, then peer1 must own the key.
2. Otherwise, peer1 asks peer2 to lookup the key in its own list of neighbours. This step is called a “hop”.
3. Peer2 starts the process again, from step one.

The speed at which a network is able to locate a resource is defined by the maximum number of “hops” it takes to lookup any given key.

Each peer is responsible for maintaining an up to date list of its neighbours. The DHT should be maintained in such a way that nodes leaving and joining the network should cause a minimal amount of disruption. A well managed DHT provides the network with the following benefits:

1. Scalability - The network can cope with an increasing number of nodes.
2. Fault tolerance - The network can cope with losing nodes or nodes failing.

The next section introduces “Kademlia”, the DHT used by PeerBot.

4.3.3 Kademlia

Peerbot uses Kademlia to build its P2P network component. Kademlia provides PeerBot with a simple, efficient means to connect peers to each other and a Commander.

Figure 4.5 shows how Kademlia is incorporated into PeerBot. Each Worker contains a node that participates in a Kademlia network. A Commander is able to communicate with PeerBot by owning a Kademlia node too. Both the Commander and the Workers are able to join the Kademlia network using a bootstrap node.

A Commander stores commands on a Kademlia network. Workers connected to the network can retrieve them, and execute them locally. Workers store any responses back on the network for the Commander to retrieve.

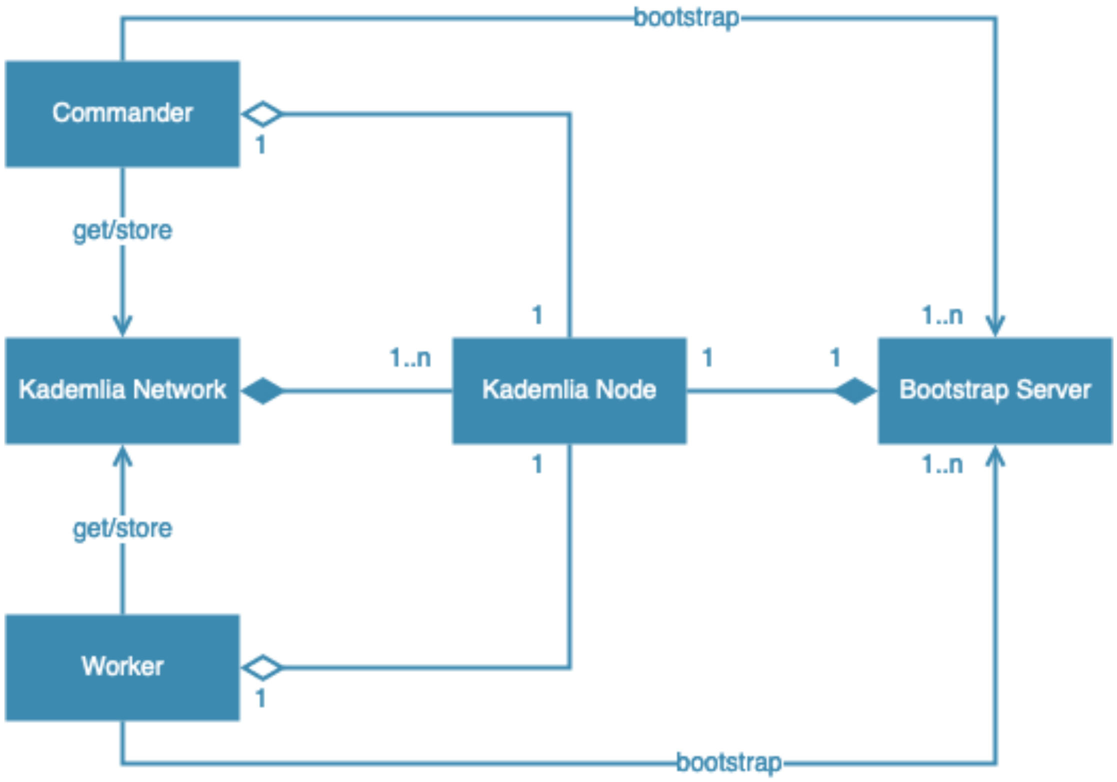


Figure 4.5 A Diagram showing how Kademlia is incorporated into PeerBot

Kademlia is a DHT for peer-to-peer networks. Like other DHTs, Kademlia has a keyspace and an overlay network.

Kademlia nodes communicate with each other using UDP. An overlay network logically links nodes participating in the network. The keyspace in Kademlia is the set of all possible 160-bit strings. Each Kademlia node has a random 160-bit identifier (**nodeID**). NodeID's are the keys used in lookups, to locate both nodes and values. Kademlia is able to look up the value associated with a given key in at most log(n) "hops", where "n" is the number of nodes in the network.

A Kademlia network can be imagined as a binary tree. Nodes are treated as leaves on the tree. A node's position is determined by the shortest unique prefix of its NodeID. The binary tree is partitioned into successively lower subtrees, that don't contain the node. For example, figure 4.6 shows the position of a node with the unique prefix 110. The highest subtree is the half of the binary tree not containing 110. The next highest is the half of the remaining tree not containing the node, and so on.

Each Kademlia node knows at least one node in each of its subtrees. This guarantees any node can reach any other node by its nodeID. Nodes keeps a list of (IP address, UDP port, nodeID) triples for each subtree. These lists are called **K-buckets**, because they each hold up to k contacts.

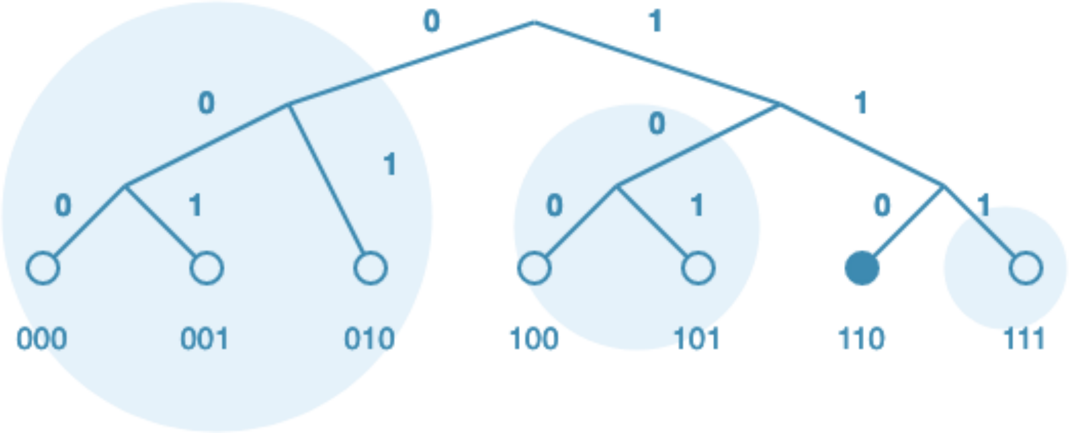


Figure 4.6 A binary tree, showing the position of a node with the prefix 110.

As mentioned in the previous section of this report, peers in a structured peer-to-peer network are organised according to how *close* they are in terms of keyspace distance. Given two nodeID's, x and y, Kademlia defines *closeness* by the exclusive or (XOR) distance between them:

- distance(x, y) = x XOR y

XOR distance is captured in a fully populated binary tree of 160-bit nodeIDs. "The magnitude of distance between two [160-bit] nodeIDs is the height of the smallest subtree containing them both. When a tree is not fully populated, the closest leaf to a nodeID x is the leaf whose nodeID shares the longest common prefix of x." (Maymounkov and Mazières, 2002)

XOR offers the following properties:

- distance(x, x) = 0
- distance(x, y) > 0 If x does not equal y

These two points simply mean that the closest node to x is x itself. All other nodes are further away.

XOR topology is symmetric:

- for all x, y : distance(x, y) = distance(y, x)

Kademlia nodes keep the most information on nodes closest to them and less on nodes further away. As nodes are encountered on the network, they are added to k-buckets. Symmetry is useful, because it means that each of a nodes contacts is gathering knowledge of nodes in the same address space. So a detailed knowledge of the same address space is maintained among all the nodes in it.

XOR also offers the triangle property:

- distance(x, z) less than or equal to distance(x, y) + distance(y, z)

If this weren't the case, a node would be unable to determine which of its contacts would give the shortest path to the target node.

XOR is unidirectional:

- For any given point x and distance d > 0, there is exactly one point y such that distance(x, y) = d

Unidirectionality and the triangle property mean that all lookups for the same key, no matter where they originate, will always converge along the same path. If this were not the case, it would be possible for a lookup to circle a target, never actually reaching it.

4.4 Sending and Receiving Commands

The previous section showed how peer-to-peer networking is incorporated into PeerBot's design. In the current design, a Commander stores commands on a Kademlia network. Workers connected to the network can retrieve them, and execute them locally. Workers store any responses back on the network for the Commander to retrieve. It is unclear whether the this design meets the following requirements:

- Only the bot-master should be able to send commands to the botnet.
- The identity go the bot-master must be protected

Currently, any node in the P2P network can store a command, as long as it knows the location Workers get them from. This means that the bot-master does not have sole control over the botnet. Furthermore, if the location the commands are stored is always the same, this could be exploited and used to disrupt the botnet. The current design also offers the identity of the bot-master no explicit protection.

Overbot is a peer-to-peer botnet that also uses Kademlia. Bots, in the Overbot network, request commands, and send messages to the bot-master by issuing search requests for specially generated cipher texts. The bot-master is able to recognise and decrypt these requests to extract the bot's sequence number. This sequence number is then used as a key to send a command to the bot.

A variation of Overbot's command and control mechanism could be integrated into PeerBot's design. However, rather than duplicating other designs, it is more interesting to try a different approach. This section describes the design decisions regarding PeerBot's command and control architecture.

4.4.1 The Command and Control Architecture

The command and control mechanism is the means by which commands are sent and retrieved. Command and control mechanisms can be classified either:

- Push — bots passively wait for the bot-master to send them commands, then forward them to other bots in the network.

- Pull — also called “command publishing/subscribing”. The bot-master publishes commands to a known location, and bots actively retrieve them.

The push mechanism could be effectively implemented in a bot-only botnet, such as PeerBot. A commander could digitally sign commands, and have Workers spread them across the P2P network. However, this is too similar to the way Overbot operates. So instead, PeerBot uses the pull mechanism.

PeerBot’s command and control mechanism was inspired by Zombiecoin. Zombiecoin is a botnet that uses the Bitcoin network to send and receive commands. Bitcoin is a distributed database, used to track of the ownership of digital currency (bitcoins). Zombiecoin uses bitcoin transactions to publish commands on the Bitcoin network. Bots are able to lookup the transactions and retrieve the commands.

In a similar manner to Zombiecoin, PeerBot uses the Ethereum network to send and receive commands. The next section introduces Ethereum and describes how the public Ethereum network is used by PeerBot.

4.4.2 Ethereum

PeerBot's command and control mechanism is built on top of the public Ethereum network. A "smart contract" intermediates communication between Workers and a Commander.

Figure 4.7 shows Workers and a Commander are able to communicate with each other both via a peer-to-peer network, and via the Ethereum network. Workers identify themselves to a Commander by using a smart contract to join the botnet. A Commander sends commands to Workers by submitting transactions to a smart contract.

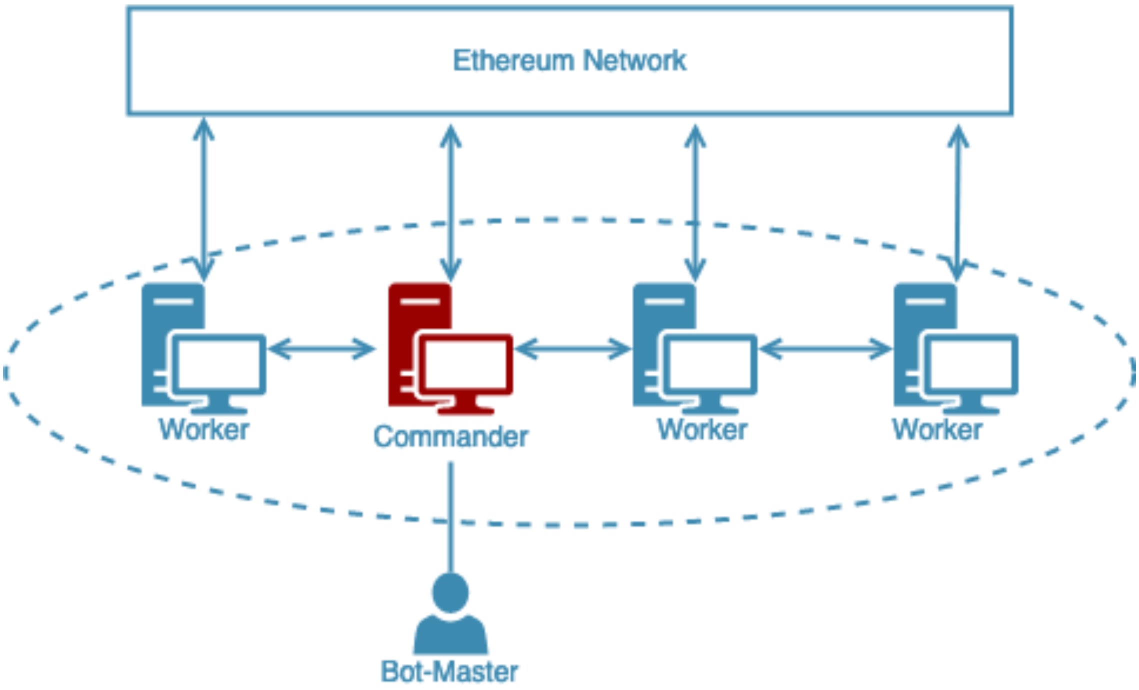


Figure 4.7 The Ethereum network is used by PeerBot to send and receive commands

Ethereum is a decentralised **blockchain** with **smart contract** functionality. A blockchain is a cryptographically linked list of records (blocks). Each block contains transactional data, a timestamp, and a cryptographic hash linking it to the previous block. The Ethereum blockchain is distributed database which tracks the state of digital assets.

A "smart contract" is simply a set of instructions that is stored and executed on the Ethereum blockchain. Smart contracts are sometimes described as “digital vending machines”, because, like a vending machine, a smart contract is pre-programmed to always give a certain output, given the right input. Smart contracts remove the need for third-party intermediaries, much like how vending machines replace vendor employees.

Smart contracts are coded in a smart contract language and deployed to a network (see section 5.3.4). After deployment, smart contracts are located at an address on the network. In order to interact with a smart contract, users send signed messages (transactions) containing the contract’s address, a compiled interface to the contract (ABI), and a transaction fee. Smart contract transactions are public, verifiable, and immutable. These qualities prevent censorship, fraud and third party interference.

PeerBot’s smart contract allows new Workers to join the botnet, and get commands. A commander uses the smart contract to accept new Workers and add commands. Only a Commander can add commands. Workers use the smart contract to join the botnet, and receive commands.

4.5 Design Summary

PeerBot contains both a Kademlia peer-to-peer network, and a smart contract on the Ethereum blockchain. Figure 4.8 is a class diagram showing how these components form PeerBot.

A Commander owns a smart contract and can use it to add commands. Workers can join the smart contract, and use it to get commands. A Worker provides an ID when joining the botnet. Worker IDs are made available to the bot-master via the smart contract.

Workers receive commands from the smart contract. They execute the commands locally, and store any results on the peer-to-peer network with their ID as the location. A Commander connects to the network and retrieves Worker responses using the Worker IDs.

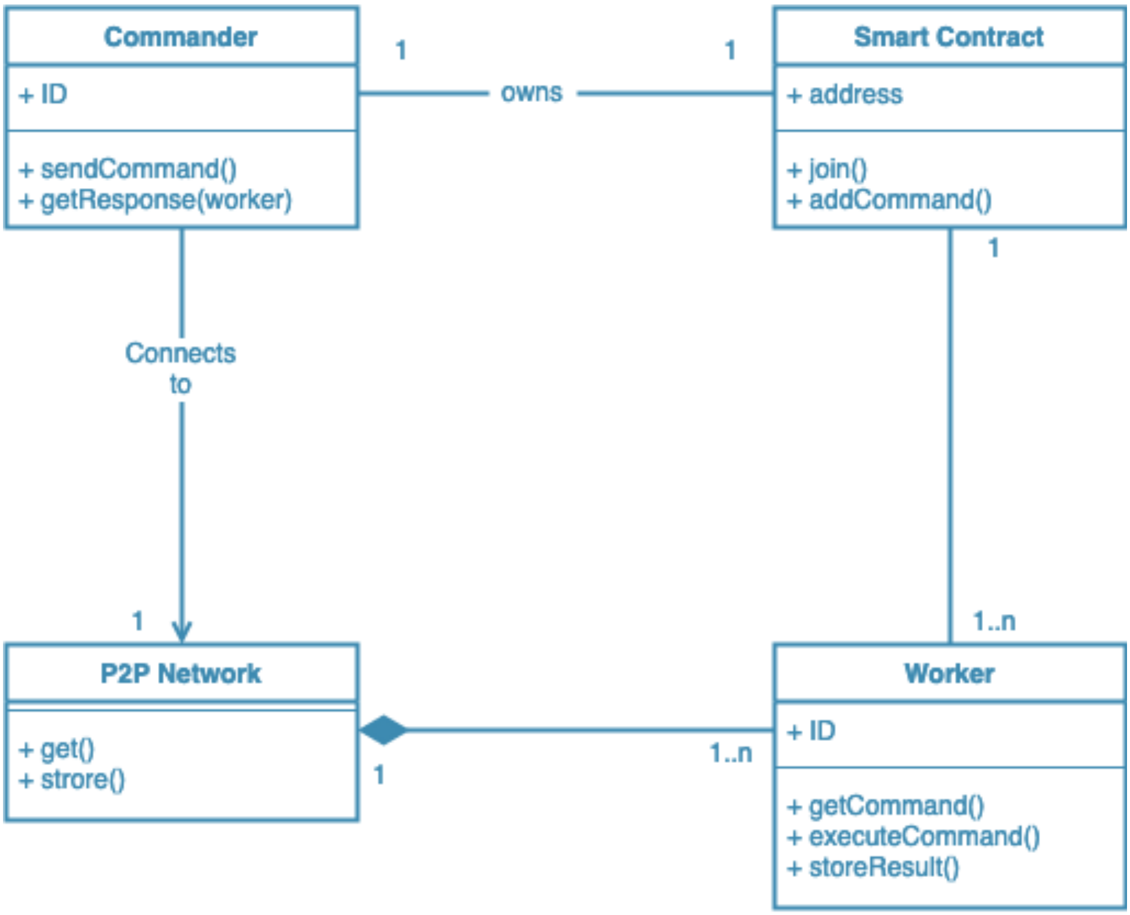


Figure 4.8 Class Diagram for PeerBot

Table 4.1 shows some of PeerBot's requirements, and explains how using Kademlia satisfies them.

Table 4.1 A few of PeerBot's requirements, and how Kademlia helps to satisfy

Requirement	***Justification***
Each node should have a unique ID, identifying it within the botnet	Both Workers and Commanders participate in the network. Each participant in a Kademlia network has a unique nodeID.
Only nodes with a valid ID should be allowed to join the botnet	Participants are assigned a valid nodeID upon bootstrapping to the network.
It could be possible for a command to be sent only to a specific node	Commands could be sent to a specific Worker by specifying the Workers nodeID, or by giving each Worker a unique key, to locate commands specifically for them.
The botnet could maintain node state information e.g active/inactive	Kademlia regularly "pings" nodes. Kademlia evicts inactive nodes.
The botnet should be scalable.	Kademlia adapts well to an increasing number of nodes.
Nodes leaving and joining the network should cause a minimal amount of disruption	Kademlia copes well with a high churn rate
Fault tolerant. The botnet must not have a single point of failure (such as with centralised botnets)	Kademlia does not rely on central coordination servers.

Table 4.2 shows how using Ethereum helps to satisfy some of PeerBots's requirements.

Table 4.2 How using Ethereum helps to satisfy some of PeerBot's requirements

Requirement	***Justification***
The botnet must be able to send, receive and execute a few simple commands.	PeerBot's smart contract allows a Commander to add commands and Workers to receive them.
Only the bot-master should be able to send commands to the botnet.	PeerBot's smart contract specifies that only the owner (bot-master) can add commands.
The identity of the bot-master must be protected.	The bot-master is kept (pseudo) anonymous with public-private key cryptography.
Fault tolerant. The botnet must not have a single point of failure (such as with centralised botnets)	Ethereum is considered fault-tolerant. The smart contract has no single point of failure.*
The command and control mechanism should be reliable.	Ethereum off fast and efficient transactions; meaning Workers will receive commands reliably soon after a Commander send them.
The botnet should have high availability	PeerBot's smart contract resides on a large, globally available public network. There is no downtime.

* It could be argued that using an Ethereum network to implement PeerBot's command and control mechanism, means PeerBot is not completely peer-to-peer; making PeerBot more of a hybrid P2P botnet. As Workers both form their own P2P network, and also each have a client-server like connection with a node in the Ethereum P2P network. These nodes, however are interchangeable (any node will do), if one Ethereum node fails, the Worker can use another.

5. Implementation and Testing

5.1 Overview

This section is split into the following subsections:

- Development Process — describes the process by which PeerBot was implemented and tested.
- Directory Structure — describes the main directory structure of PeerBot
- The Peer-to-Peer Network — discusses the implementation of PeerBot's P2P. network.
- The Command and Control Mechanism — discusses the implementation of PeerBot's command and control mechanism.
- Tests — presents some tests for PeerBot.

5.1.1 Development Process

The development process of this project was done in three main stages. First, the P2P network was developed and tested. Then, the command and control mechanism was separately developed and tested. Finally, both components were combined and tested.

5.1.2 Directory Structure

Figure 5.1 shows the directory structure of PeerBot.

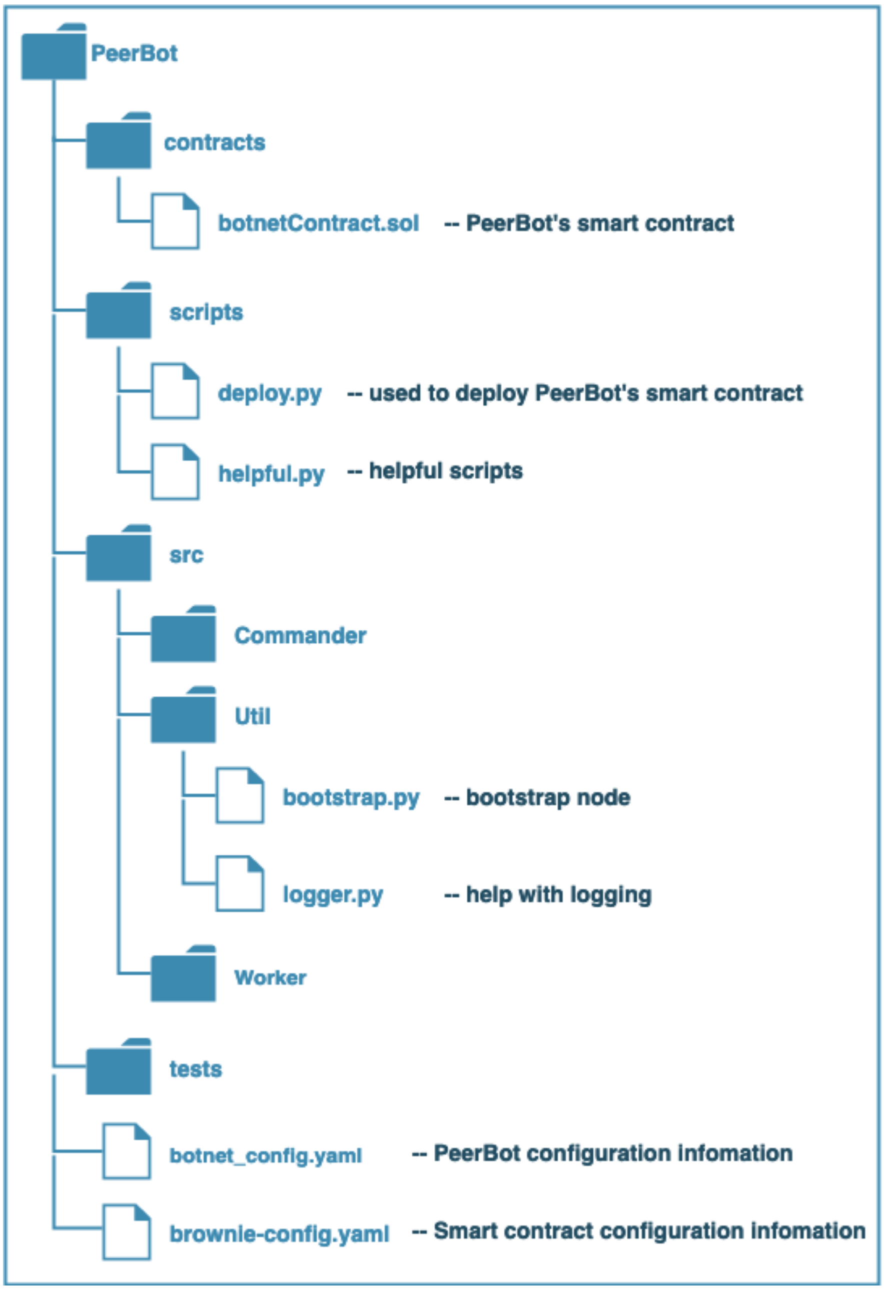

PeerBot
contracts
botnetContract.sol -- PeerBot's smart contract
scripts
deploy.py -- used to deploy PeerBot's smart contract
helpful.py -- helpful scripts
src
Commander
Util
bootstrap.py -- bootstrap node
logger.py -- help with logging
Worker
tests
botnet_config.yaml -- PeerBot configuration infomation
brownie-config.yaml -- Smart contract configuration infomation

Figure 5.1 PeerBot’s directory structure

5.2 The Peer-to-Peer Network

This section discussing implementing and testing PeerBot's peer-to-peer network. This section is divided into the following subsections:

- Kademlia — discusses the Kademlia implementation and API
- Asynchronous programming — explains briefly
- Worker — implementing the Workers role in the P2P network
- Commander — implementing the Commanders role in the P2P network

5.2.1 Kademlia

The peer-to-peer network uses the Kademlia protocol to structure and operate its peer-to-peer network. Each Worker serves as a peer in the network. A Commander can connect to the network and use it to lookup Worker keys.

PeerBot uses Brian Muller's Kademlia implementation (source: Muller, 2021) to build the P2P network. This library is written in python, and adheres to Kademlia's original specification.

The Kademlia protocol provides a simple API:

- **node(ip, port) —** instantiates a new node in the Kademlia network. The new node is assigned a 160-bit ID. The network associates this ID with the supplied ip address and port.

- **bootstrap(peer_list) —** uses the given list of peers to connect to the network.

- **get(key) —** This is method retrieves the item associated with the given key in the network.

- **set(key, item) —** associates the given key with the given item, making it available to any participant in the network.

This Kademlia library uses RPC over UDP to facilitate physical communication among peers. RPC over UDP is a technique used to communicate with devices behind a NAT.

5.2.2 Asynchronous Programming

The chosen Kademlia implementation uses “asyncio”. This is a python library for asynchronous programming. Asyncio achieves concurrency by running processes in an event loop. When a process is unable to do any work, because it is waiting for something, the event loop runs another process while the first process is waiting. This is useful for Kademlia, because a lot of time might be spent waiting for IO.

5.2.3 Worker

Workers form the peer-to-peer network. Workers primarily use this network to communicate with a Commander. To participate in the P2P network a Worker needs the following:

- **ID** — A Commander can connect to the network and locate the results using the Worker's ID as a key.

- **Node** — Nodes form the P2P network. The node should use the Kademlia protocol. The node should provide a method to associate a key with some data on the network. This method will be used to store messages, at the location of the ID, on the P2P network.

Figure 5.2 shows the directory structure of Worker.

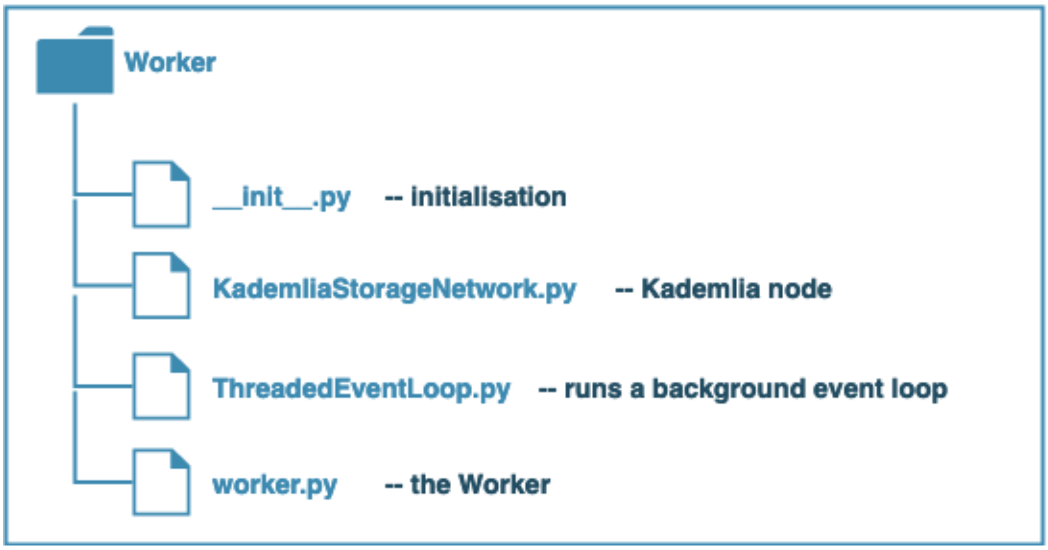

Worker
__init__.py -- initialisation
KademliaStorageNetwork.py -- Kademlia node
ThreadedEventLoop.py -- runs a background event loop
worker.py -- the Worker

Figure 5.2 The directory structure of Worker

Figure 5.3 is a class diagram, showing the structure of a Worker. A Worker contains a **StorageNetwork**, which it uses to connect to a P2P network. The StorageNetwork is also used to store messages on the P2P network.

KademliaStorageNetwork is a type of StorageNetwork. A KademliaStorageNetwork contains a Kademlia node. Kademlia nodes are what make Workers peers in PeerBot's P2P network.

Kademlia requires an asynchronous event loop in order to provide its services (see section 5.2.4.2). **ThreadedEventLoop** runs an event loop in the background. The Kademlia node runs in this event loop. Asynchronous processes can be submitted to run in the event loop without blocking the main loop (see section 5.2.4.2).

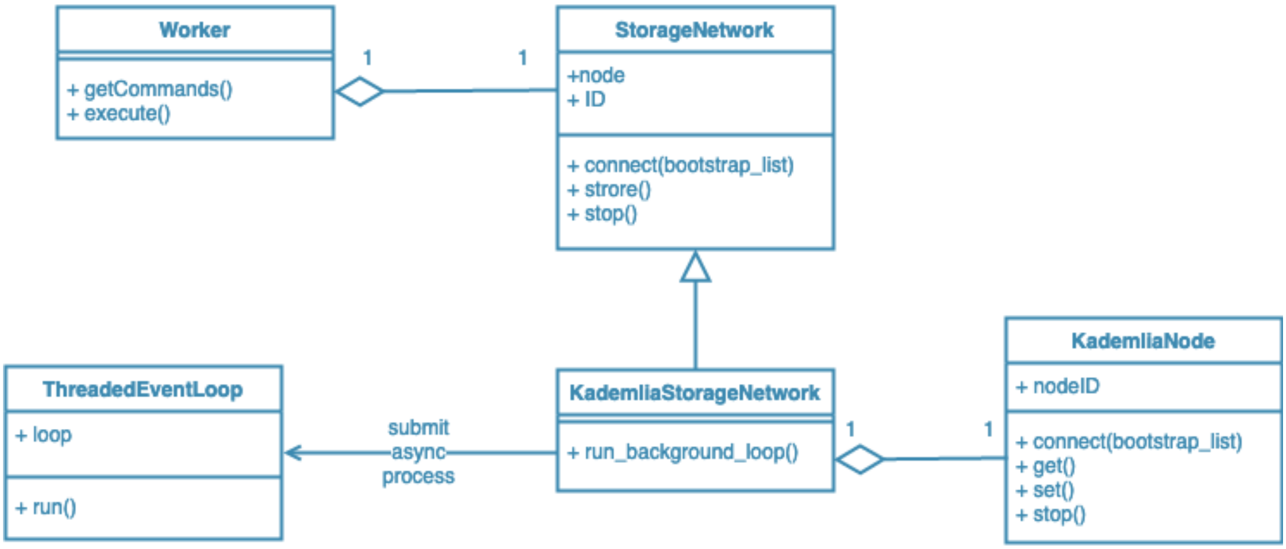


Figure 5.3 The structure of Worker

5.2.3.1 Peer-to-Peer Storage Network

To instantiate a Worker object, a **StorageNetwork** must be supplied to the constructor. A StorageNetwork handles connecting to the peer-to-peer network, running a node in the network, and storing items on the network.

For an object to be a StorageNetwork it must provide the following methods:

- connect(bootstrap_list)
- store(item)
- stop()

A StorageNetwork object must also contain the following:

- node — running a peer in the peer-to-peer network
- id — the ID of the Worker

A **KademliaStorageNetwork** object provides the above methods and variables. A KademliaStorageNetwork object uses the Kademlia API to create and run a new Kademlia node, connect to PeerBot’s peer-to-peer network using a hard-coded list of bootstrap nodes, and store items on the network using the node’s ID.

Workers are supplied with a running KademliaStorageNetwork upon instantiation.

5.2.3.2 Combining Asynchronous and Multi-Threaded Programming

The chosen Kademlia implementation uses asynchronous programming.The Kademlia node requires a running event loop. The library (Web3) used to implement PeerBot’s command and control mechanism is not asynchronous. This problematic, because both the asynchronous processes that run the Kademlia peer, and the processes that handle fetching and executing commands, are incompatible, and both require full control of the main loop.

To solve this problem a Worker combines both asynchronous programming and multi-threading. (This is not typically done, as both asynchronous programming and multithread are usually used separately, as different ways to achieve concurrency. It can also lead to race conditions, however in this instance it works).

A Kademlia node requires a running event loop in order to stay connected to the network. Kademlia’s API methods are asynchronous; They also require a running event loop. The rest of a Worker’s code is blocking.

A **ThreadedEventLoop** is a thread that runs an event loop in the background. This means the event loop can run constantly without blocking the main loop. When the main loop wants to store and item on the P2P network, it submits the method to run in the ThreadedEventLoop’s event loop. The main loop can then continue without waiting for the “store” method to finish. The main loop does not handle any errors occurring in the P2P network, so it does not need to receive feedback from the processes it sends ThreadedEventLoop.

Figure 5.4 shows how a Worker combines asynchronous and multi-threaded programming. A main thread submits asynchronous process to run in a separate thread running an event loop. This allows the main thread to continue its loop without blocking. Meanwhile, the ThreadedEventLoop runs the asynchronous process on its event loop.

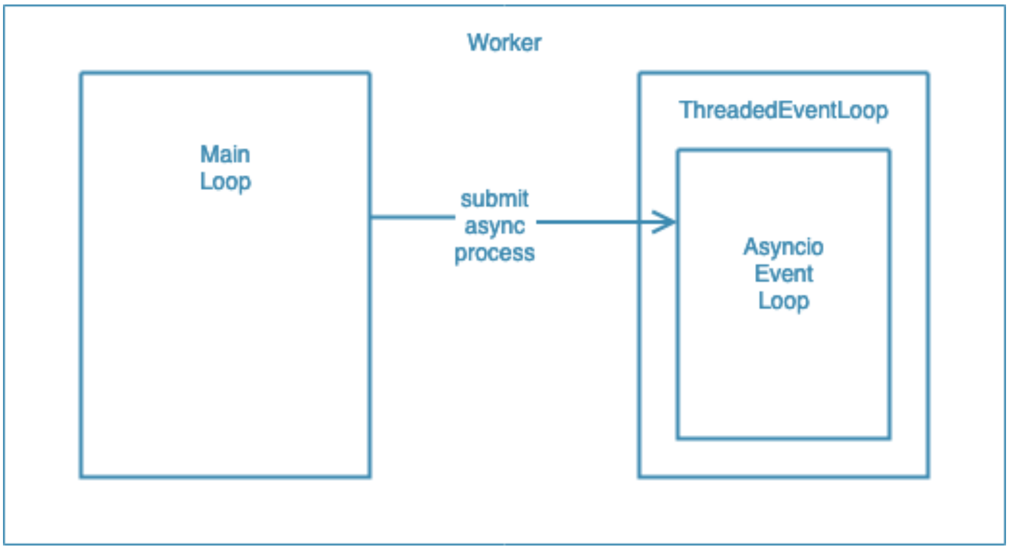


Figure 5.4 Combining asyncio and multithreading

5.2.4 Commander

Figure 5.5 shows the directory structure of Commander.

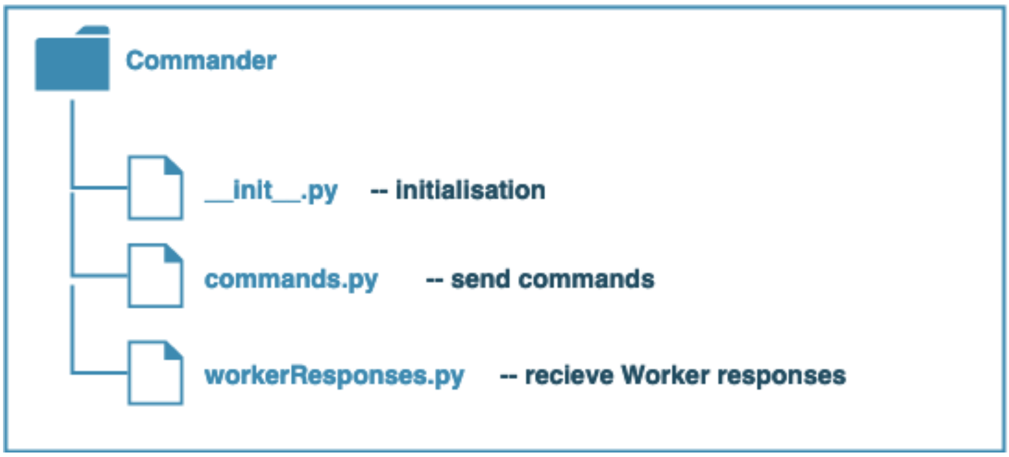

Commander
__init__.py -- initialisation
commands.py -- send commands
workerResponses.py -- recieve Worker responses

Figure 5.5 The directory structure of Commander

5.2.5.1 Connecting to The Peer-to-Peer Network

A Commander uses the peer-to-peer network to retrieve messages from Workers. It is not necessary for a Commander to be running a Kademlia node all the time (like a Worker does). A Commander just needs to be able to connect to the network, collect the messages, and leave. Figure 5.6 is an activity diagram, describing a Commander collecting Worker responses.

To use workerResponses.py, the bot-master uses:

```
$ python workerResponses.py  <worker key>
```

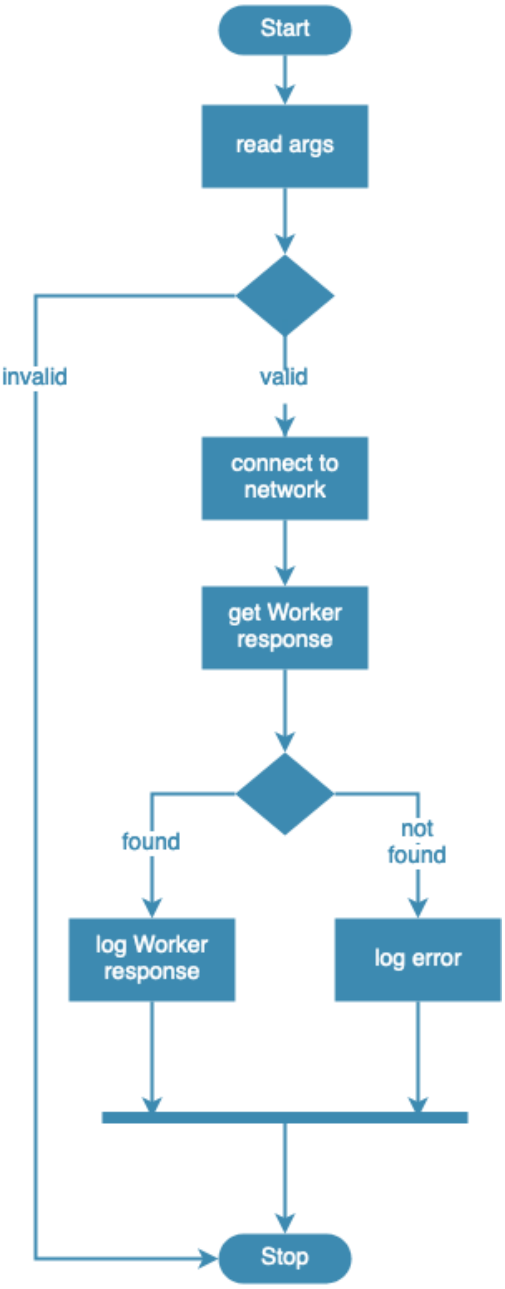


Figure 5.6 A Commander accessing the P2P network

5.3 The Command and Control Mechanism

PeerBot's command and control mechanism is implemented by using a smart contract. Smart contracts allow participants to transact with each other without a third-party intermediary. A transaction is a signed message sent from one Ethereum account to another.

5.3.1 Wallets

Wallets are needed in order interact with ethereum blockchain. A wallet contains accounts. Each account contains the following:

- Public address — a unique identifier. PeerBot uses a public address to provide the bot-master with pseudo-anonymity.
- Private key — the account's password. The private key proves ownership of the public address — anyone who knows the private key is considered the owner of the public address. Therefore, it is very important that the private key be kept secret.

Creating wallets is out of the scope of this project, so pre-made wallets are used for all transactions.

5.3.2 Testnets

Transacting with smart contracts on the Ethereum network costs currency. Deploying and transacting with a smart contract on the Ethereum mainnet can get expensive. PeerBot is not intended for use in the real world. Instead of the Ethereum mainnet, PeerBot's smart contract is deployed on a testnet.

Testnets work exactly the same as Ethereum, but use currencies with no real value.

5.3.3 Transactions

All interaction with ethereum and smart contract are done through transactions. Each transaction contains information on both the sender and the recipient, the smart contract bytecode, and the transaction fee.

Web3.py is a Python library for interacting with Ethereum. Web3 allows PeerBot to transact with BotnetContract. This project uses Web3 to build, sign, and send transactions.

Web3 is used to build a contract object using the smarts contract's address and ABI. This can be used to make make function calls, or filter smart contract events.

In order to build a transaction in web3, the following is required:

- "chainId" — the ID of the network
- "from address" — the sender's public address
- "nonce" — the number of transaction the sender has already made

The sender must sign the transaction with their private key, before it can be sent.

5.3.4 Solidity

Solidity is an object-oriented language for implementing smart contracts. BotnetContract.sol defines a solidity contract object called **BotnetContract**.

BotnetContract contains an address object named “**owner**”. Address objects are public Ethereum addresses. “owner” is the bot-master’s public address.

BotnetContract also contains a list of Workers (called “**workers”**). When a Worker joins the botnet, they are added to the list.

The a collection of following types of objects form the rest of BotnetContract:

- “event” — a way to communicate something has happened on the blockchain.
- “modifier” — used to modify the behaviour of a function.
- “function” — self-contained set of instructions.

BotnetContract defines the modifier “**onlyOwner**”, only allowing the owner to perform some task.

BotnetContract defines the “events”:

- joinRequest(workerKey) — a Commander listens for these
- commandAdded(workerKey, command) – Workers listen for these

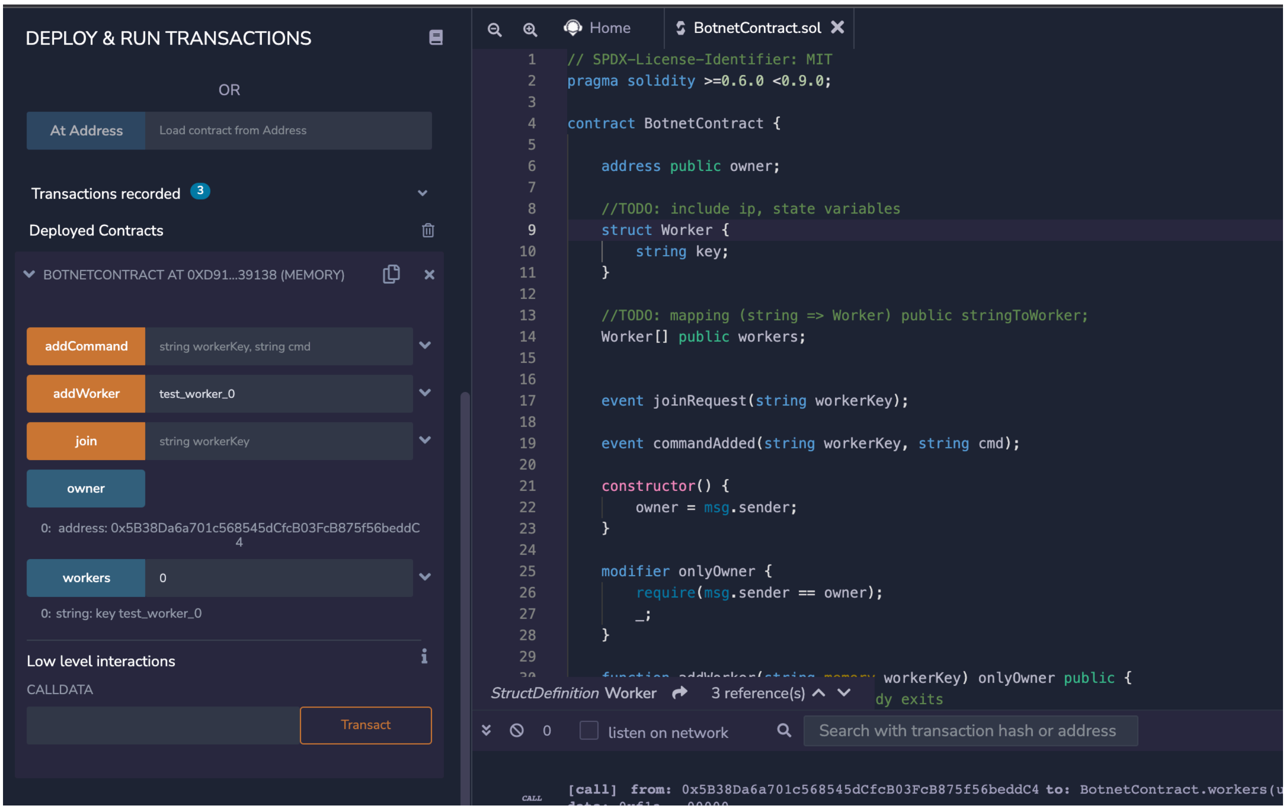
In Solidity “functions” can be used to view or change the state of the smart contract. BotnetContract defines the following functions:

- join(workerKey) — emits a joinRequest event

- addWorker(workerKey) — uses the “onlyOwner” modifier, to only allow the “owner” of BotnetContract to use this function. It adds a Worker to BotContract’s “workers”.

- addCommand(workerKey, command) — emits a commandAdded event.

Figure 5.7 is a screenshot from “remix IDE”, a helpful tool for developing smart contracts. It shows the data and functions made available by BotnetContract (left). The orange buttons are functions. The blue buttons are publicly available data.



addWorker: test_worker_0

owner — 0: address: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

workers: 0 — 0: string: key test_worker_0

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.6.0 <0.9.0;

contract BotnetContract {

    address public owner;

    //TODO: include ip, state variables
    struct Worker {
        string key;
    }

    //TODO: mapping (string => Worker) public stringToWorker;
    Worker[] public workers;


    event joinRequest(string workerKey);

    event commandAdded(string workerKey, string cmd);

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function addWorker(string memory workerKey) onlyOwner public {
                                       dy exits
```

```
0
[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: BotnetContract.workers(u
```

Figure 5.7 BotnetContract on “remix IDE”

5.3.5 Brownie

Brownie is a Python smart contract development platform. Brownie is a useful tool, used to deploy and manage smart contracts. Brownie allows users to easily deploy contracts on different networks and testnets. Brownie can also manage accounts.

This project used Brownie to simplify smart contract development, deployment and interaction.

5.3.6 Sending and Receiving Commands

Now that BotnetContract has been deployed onto the Rinkeby testnet, it can be transacted with, to accept commands from the "owner", and deliver them to the "workers"

5.3.6.1 Connecting to Ethereum

PeerBot uses an "infura API key" to connect to and interact with Ethereum.

"Infura" is an API gateway to the mainnet and some testnets. PeerBot does not run its own Ethereum clients, so must rely on Infura.

5.3.6.2 Sending Commands

A Commander sends commands by taking command-line input from the bot-master. The bot-master can choose to specify a "worker key". When the bot-master does this only the worker whose ID corresponds to "worker key" will execute the command. If the bot-master does not supply a "worker key", then there keyword "ALL" will be used, meaning every Worker will execute the command.

The above service is provided by the "commands.py" module's mainloop. A separate thread listens for "joinRequest" events. If one is detected, then associated "worker key" is added to BotnetContract using the "addWorker" function.

5.3.6.3 Receiving Commands

Before they can receive any commands, Workers must first join the botnet. This is done by calling BotnetContract’s “join” function, giving it their ID. To do this Workers need an account with some funds.

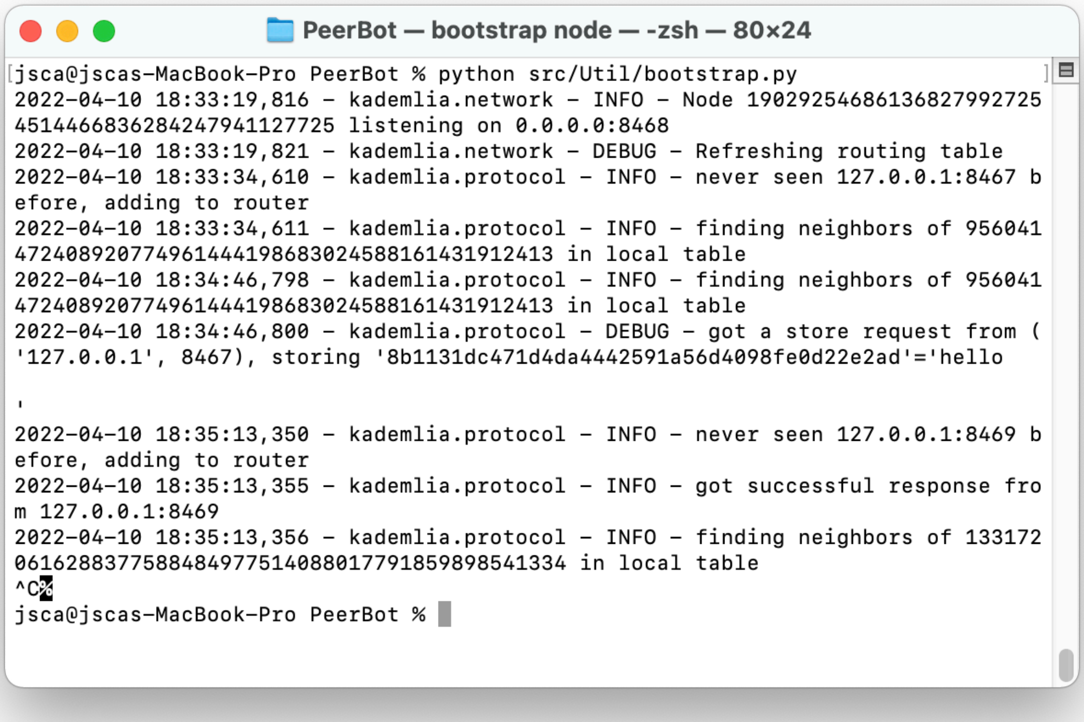
Workers receive commands by listening for “commandAdded” events. When a command is detected, the event’s “workerKey” field is checked. If it is equal to either “ALL” or the Worker’s ID, then the command is executed, otherwise it is discarded.

5.4 Tests

Tests were done locally, but Kademlia should also work behind a NAT.

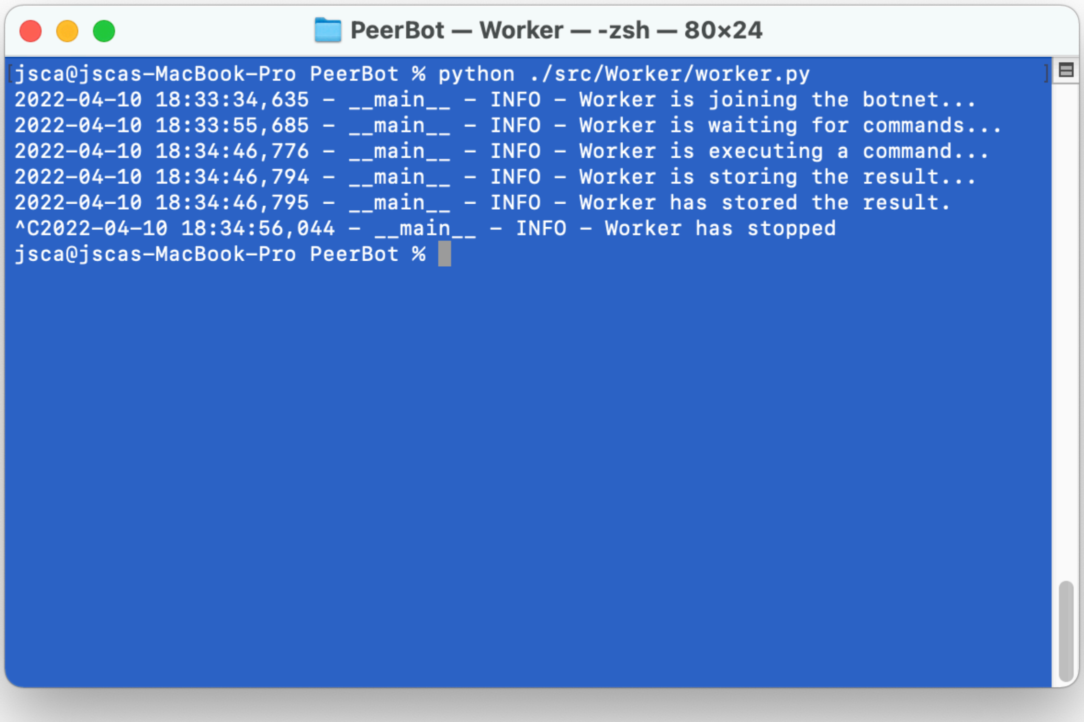
1. Run a bootstrap node: *$ python src/Util/bootstrap.py (ensure there is a few kademlia nodes/ Workers running)*
2. Run a Worker: *$ python src/Worker/worker.py*
3. Run a Commander: *$ python src/Commander/commands.py*
4. Use the Commander to add a command (e.g type “echo hello”)
5. Collect the response: *$python src/Commander/workerResponses.py <key>*

Figures 5.8 - 5.12 show the above steps having been successfully executed.



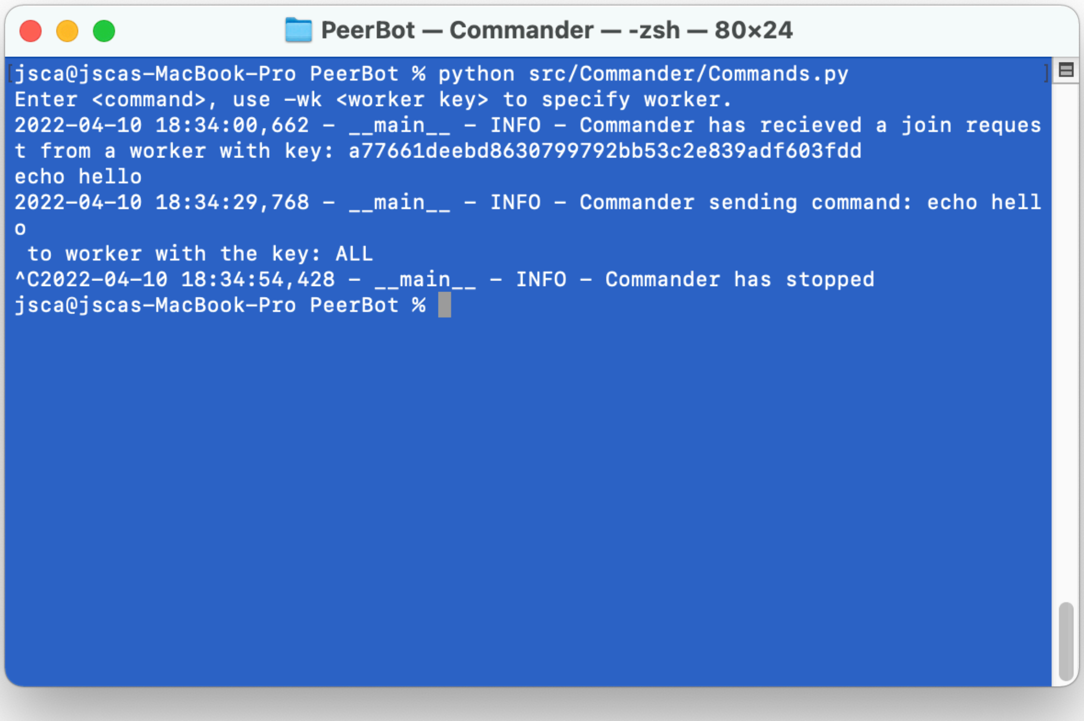
```
jsca@jscas-MacBook-Pro PeerBot % python src/Util/bootstrap.py
2022-04-10 18:33:19,816 - kademlia.network - INFO - Node 19029254686136827992725
4514466836284247941127725 listening on 0.0.0.0:8468
2022-04-10 18:33:19,821 - kademlia.network - DEBUG - Refreshing routing table
2022-04-10 18:33:34,610 - kademlia.protocol - INFO - never seen 127.0.0.1:8467 b
efore, adding to router
2022-04-10 18:33:34,611 - kademlia.protocol - INFO - finding neighbors of 956041
472408920774961444198683024588161431912413 in local table
2022-04-10 18:34:46,798 - kademlia.protocol - INFO - finding neighbors of 956041
472408920774961444198683024588161431912413 in local table
2022-04-10 18:34:46,800 - kademlia.protocol - DEBUG - got a store request from (
'127.0.0.1', 8467), storing '8b1131dc471d4da4442591a56d4098fe0d22e2ad'='hello

'
2022-04-10 18:35:13,350 - kademlia.protocol - INFO - never seen 127.0.0.1:8469 b
efore, adding to router
2022-04-10 18:35:13,355 - kademlia.protocol - INFO - got successful response fro
m 127.0.0.1:8469
2022-04-10 18:35:13,356 - kademlia.protocol - INFO - finding neighbors of 133172
0616288377588484977514088017791859898541334 in local table
^C%
jsca@jscas-MacBook-Pro PeerBot %
```

Figure 5.8 running the bootstrap server



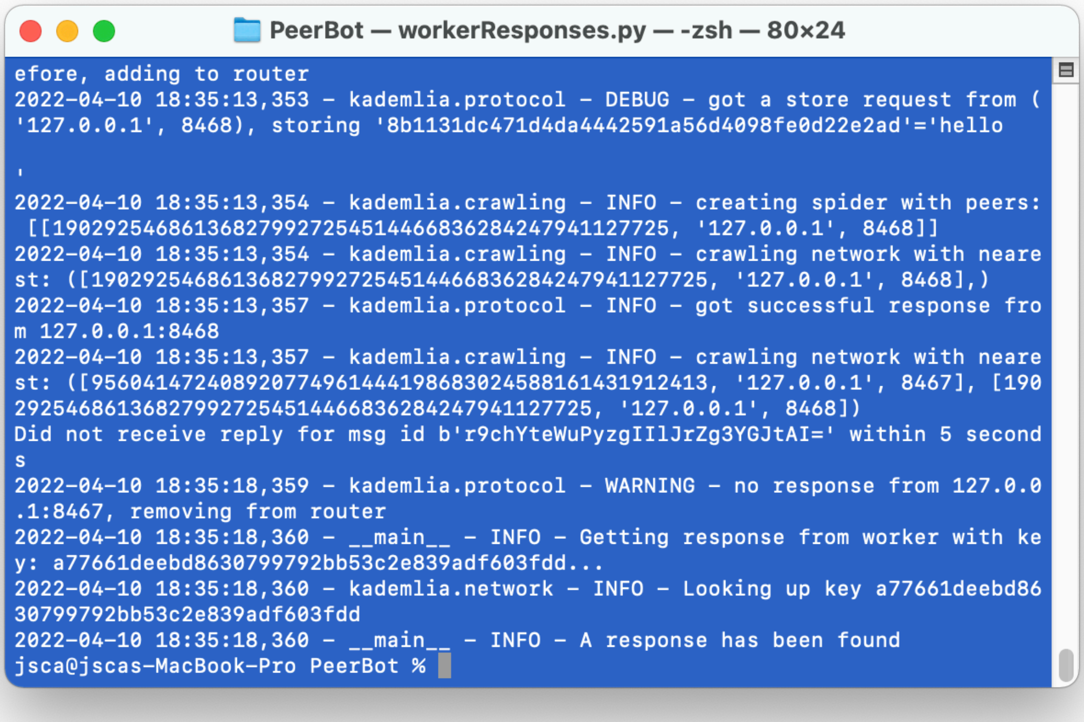
```
jsca@jscas-MacBook-Pro PeerBot % python ./src/Worker/worker.py
2022-04-10 18:33:34,635 - __main__ - INFO - Worker is joining the botnet...
2022-04-10 18:33:55,685 - __main__ - INFO - Worker is waiting for commands...
2022-04-10 18:34:46,776 - __main__ - INFO - Worker is executing a command...
2022-04-10 18:34:46,794 - __main__ - INFO - Worker is storing the result...
2022-04-10 18:34:46,795 - __main__ - INFO - Worker has stored the result.
^C2022-04-10 18:34:56,044 - __main__ - INFO - Worker has stopped
jsca@jscas-MacBook-Pro PeerBot % 
```

Figure 5.9 running a Worker



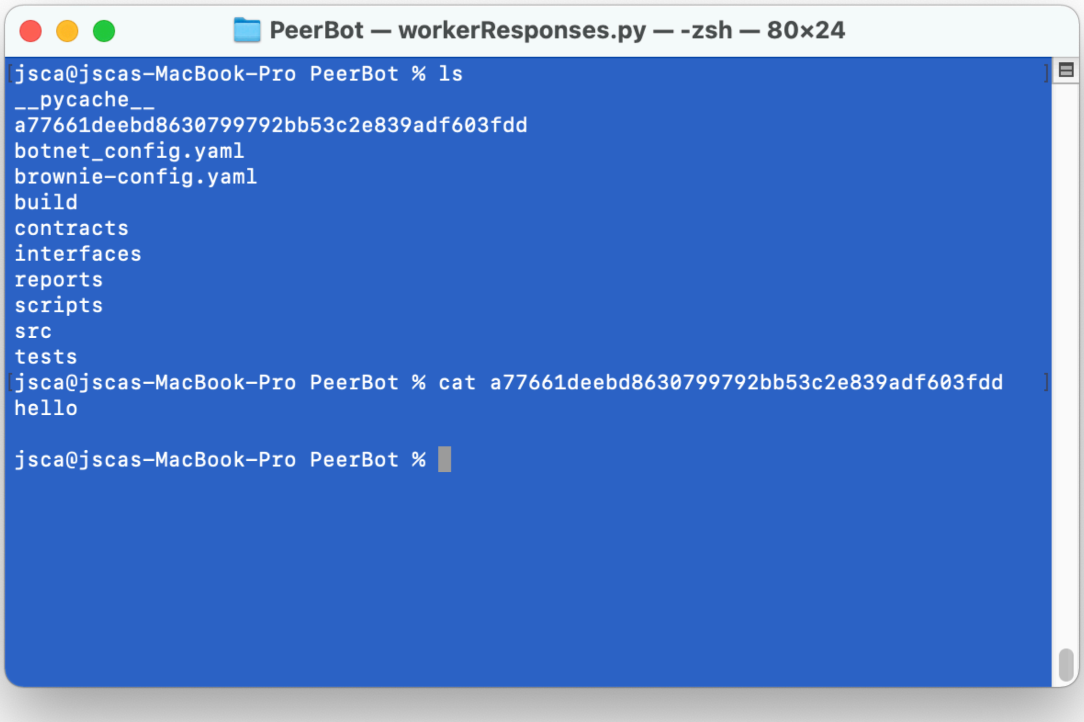
```
jsca@jscas-MacBook-Pro PeerBot % python src/Commander/Commands.py
Enter <command>, use -wk <worker key> to specify worker.
2022-04-10 18:34:00,662 - __main__ - INFO - Commander has recieved a join reques
t from a worker with key: a77661deebd8630799792bb53c2e839adf603fdd
echo hello
2022-04-10 18:34:29,768 - __main__ - INFO - Commander sending command: echo hell
o
 to worker with the key: ALL
^C2022-04-10 18:34:54,428 - __main__ - INFO - Commander has stopped
jsca@jscas-MacBook-Pro PeerBot % 
```

Figure 5.10 running a Commander.
(The Commander receives a join request from a Worker with the key:
"a77661deebd8630799792bb53c2e839adf603fdd")



```
efore, adding to router
2022-04-10 18:35:13,353 - kademlia.protocol - DEBUG - got a store request from (
'127.0.0.1', 8468), storing '8b1131dc471d4da4442591a56d4098fe0d22e2ad'='hello

'
2022-04-10 18:35:13,354 - kademlia.crawling - INFO - creating spider with peers:
 [[190292546861368279927254514466836284247941127725, '127.0.0.1', 8468]]
2022-04-10 18:35:13,354 - kademlia.crawling - INFO - crawling network with neare
st: ([190292546861368279927254514466836284247941127725, '127.0.0.1', 8468],)
2022-04-10 18:35:13,357 - kademlia.protocol - INFO - got successful response fro
m 127.0.0.1:8468
2022-04-10 18:35:13,357 - kademlia.crawling - INFO - crawling network with neare
st: ([956041472408920774961444198683024588161431912413, '127.0.0.1', 8467], [190
292546861368279927254514466836284247941127725, '127.0.0.1', 8468])
Did not receive reply for msg id b'r9chYteWuPyzgIIlJrZg3YGJtAI=' within 5 second
s
2022-04-10 18:35:18,359 - kademlia.protocol - WARNING - no response from 127.0.0
.1:8467, removing from router
2022-04-10 18:35:18,360 - __main__ - INFO - Getting response from worker with ke
y: a77661deebd8630799792bb53c2e839adf603fdd...
2022-04-10 18:35:18,360 - kademlia.network - INFO - Looking up key a77661deebd86
30799792bb53c2e839adf603fdd
2022-04-10 18:35:18,360 - __main__ - INFO - A response has been found
jsca@jscas-MacBook-Pro PeerBot % 
```

Figure 5.11 fetching a response from Worker with the key: "a77661deebd8630799792bb53c2e839adf603fdd"



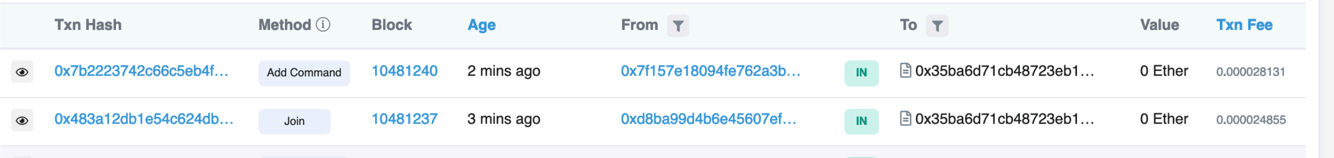
```
jsca@jscas-MacBook-Pro PeerBot % ls
__pycache__
a77661deebd8630799792bb53c2e839adf603fdd
botnet_config.yaml
brownie-config.yaml
build
contracts
interfaces
reports
scripts
src
tests
jsca@jscas-MacBook-Pro PeerBot % cat a77661deebd8630799792bb53c2e839adf603fdd
hello

jsca@jscas-MacBook-Pro PeerBot % 
```

Figure 5.12 viewing the response

The command and control mechanism was also tested using “Etherscan”. (Available at: “https://rinkeby.etherscan.io")

“Etherscan” is a block explorer; It Allows users to view transactions on an ethereum blockchain.

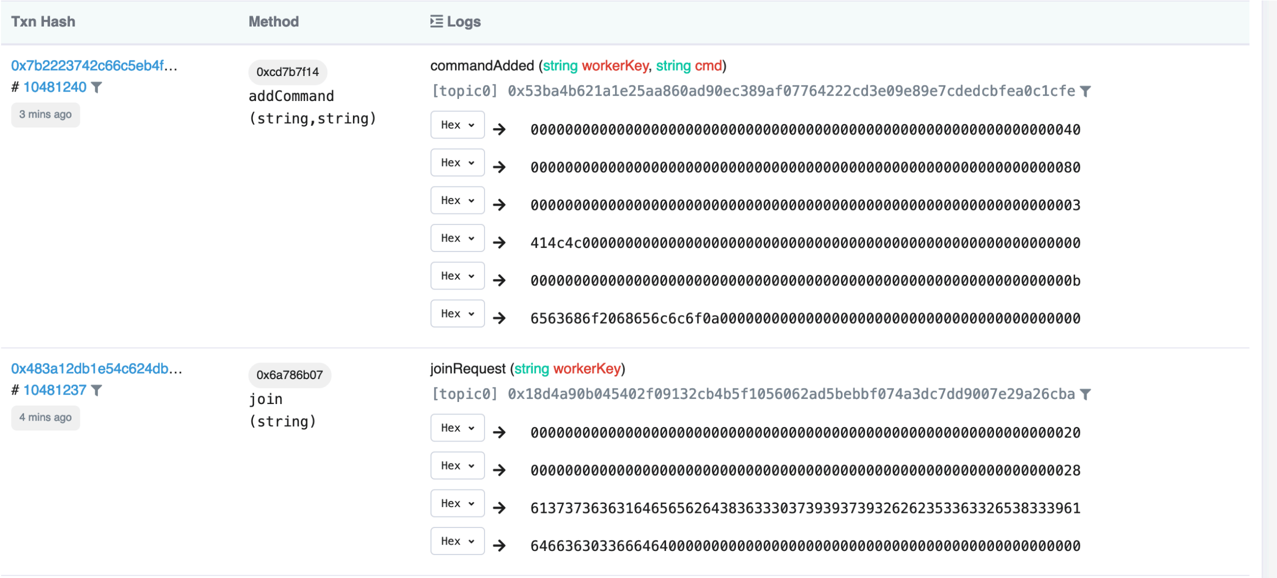
Figure 5.13 shows the transactions made by the above tests. It shows a transaction between the smart contact and the Commander’s public address (AddCommand). It also shows a transaction between a Worker’s public address and the smart contact (Join).



	Txn Hash	Method ⓘ	Block	Age	From		To	Value	Txn Fee
👁	0x7b2223742c66c5eb4f…	Add Command	10481240	2 mins ago	0x7f157e18094fe762a3b…	IN	0x35ba6d71cb48723eb1…	0 Ether	0.000028131
👁	0x483a12db1e54c624db…	Join	10481237	3 mins ago	0xd8ba99d4b6e45607ef…	IN	0x35ba6d71cb48723eb1…	0 Ether	0.000024855

Figure 5.13 Etherscan

Figure 5.14 show the smart contract events from the above tests



Txn Hash	Method	Logs
0x7b2223742c66c5eb4f… # 10481240 3 mins ago	0xcd7b7f14 addCommand (string,string)	commandAdded (string workerKey, string cmd) [topic0] 0x53ba4b621a1e25aa860ad90ec389af07764222cd3e09e89e7cdedcbfea0c1cfe Hex → 0000000000000000000000000000000000000000000000000000000000000040 Hex → 0000000000000000000000000000000000000000000000000000000000000080 Hex → 0000000000000000000000000000000000000000000000000000000000000003 Hex → 414c4c0000000000000000000000000000000000000000000000000000000000 Hex → 000000000000000000000000000000000000000000000000000000000000000b Hex → 6563686f2068656c6c6f0a000000000000000000000000000000000000000000
0x483a12db1e54c624db… # 10481237 4 mins ago	0x6a786b07 join (string)	joinRequest (string workerKey) [topic0] 0x18d4a90b045402f09132cb4b5f1056062ad5bebbf074a3dc7dd9007e29a26cba Hex → 0000000000000000000000000000000000000000000000000000000000000020 Hex → 0000000000000000000000000000000000000000000000000000000000000028 Hex → 6137373636316465656264383633303739393739326262353363326538333961 Hex → 6466363033666464000000000000000000000000000000000000000000000000

Figure 5.14 commandAdded and joinRequest events

6. Evaluation

6.1 Overview

This section evaluates some of PeerBots main features. The aim of this project was to research and design a peer-to-peer botnet. PeerBot was developed in order to explore both peer-to-peer networking and botnet design. Real world deployment and thorough testing are outside of the scope of this project's objectives. The evaluation will focus on the design strengths and weaknesses.

6.2 Evaluating Peer-to-Peer Botnets

Peer-to-peer botnets are typically evaluate by the following measures:

- Efficiency — how fast the botnet receives a command
- Robustness — how resilient the botnet is to nodes going offline/shutting down

6.2.1 Efficiency

PeerBot uses a smart contract to send and receive commands. This provides an efficient means to send and receive commands.

"In Ethereum, the average block time is between 12 to 14 seconds and is evaluated after each block." (Ethereum, 2022). This means that from the point a commander sends a to the point it is visible to the workers should be no more that 14 seconds.

PeerBot's P2P network uses Kademlia to organise its peers and locate data. A lookup in Kademlia can be completed by contacting only O(log(n)) nodes, out of a total "n" nodes in the network.

6.2.2 Robustness

Robustness describes how resilient a Bonet is to nodes getting shutdown. Ethereum itself is a robust network, however in order to send and receive commands in PeerBoth, both the Commander and the Workers use an “Infura API key” to access Ethereum. If, for some reason, this key were to stop providing this service, PeerBot would fail. This is not a robust design.

PeerBot uses Kademlia to structure the P2P network that Workers use to send messages to a Commander. Kademlia is able to cope with nodes failing, because of the way it organises the network. Nodes maintain up to date information on their neighbours. The network is able to patch any holes left by nodes going offline, by updating the routing information of neighbouring nodes.

6.3 Countermeasures

This section examines how PeerBot might perform against the following countermeasures use by security professional to combat botnets:

- Detection
- Monitoring
- Shutdown

6.3.1 Detection

Bot-only botnets may be detected by observable patterns of behaviour. PeerBot does not hide UDP communication among peers. Or obscure connecting to Ethereum, and storing information on the blockchain. These choices make PeerBot detectable.

6.3.2 Monitoring

One of the ways security professionals monitor botnets is by hijacking a node. Another way is by observing selected nodes using “sensors”. The behaviour of the node is tracked, and used to gain a detailed understanding of the whole network.

PeerBot makes it difficult to monitor the entire network with just one node, because a Kademlia lookup only queries the closest nodes. In order to monitor PeerBot, an evenly distributed set of nodes would need to be hijacked.

Another way PeerBot could be monitored it by watching transaction made containing the smart contract address.

6.3.4 Shutdown

The following are some of the methods used by security professionals to shut down a peer-to-peer botnet:

- Targeting the bootstrap procedure
- Index Poisoning
- Sybil attacks

Targeting The Bootstrap Procedure

A list of bootstrap nodes are hard-coded into each Worker. If these nodes were discovered and destroyed, the Worker would be unable to join the P2P network.

The Worker could, however still receive commands, as the command delivery mechanism is decoupled from the P2P network. Without the P2P network, PeerBot could still receive and execute commands, but it would unable to send the bot-master messages.

Index Poisoning

An index poisoning attack is when a peer-to-peer networks routing table are corrupted , by flooding the network with false nodes (nodes that don’t exist). This leave true nodes unable to find each other. PeerBot’s P2P network is vulnerable to this.

As with targeting the bootstrap nodes, even is PeerBot's P2P network is shut down, the bot-master can still send commands via the smart contract.

Sybil Attack

A Sybil attack is similar to index poisoning, except the network is infiltrated by active, disruptive nodes. As with index poisoning, PeerBot's P2P network is vulnerable to this.

6.4 An Improved Design

The following three areas could be improved upon, in order to make PeerBot more robust:

- The P2P protocol
- Connecting to Ethereum
- Design

6.4.1 The P2P Protocol

Storage Network is decoupled from Worker. A More secure implementation could be swapped in.

PeerBot's current network is vulnerable to both index poisoning and sybil attacks. This could be improved by requiring certain nodes to be validated before they are inserted into routing tables.

The hard coded bootstrap nodes may also be used as a way to shut down the network. If a Worker finds itself unable to connect to the network, because all of the bootstrap nodes are unavailable, it could use the ethereum network. Regular bootstrap nodes could be polished to the Ethereum blockchain.

6.4.2 Connecting to Ethereum

Currently, PeerBot uses "infura" to connect to Ethereum. A more robust design would have each worker run their own Ethereum client locally. This removes the single point of failure using "infura" introduces.

However, some Workers may not be able to run their own Ethereum clients, because they do not have enough space, are behind a NAT etc. The next section describes an alternate design to help solve this problem.

6.4.3 Design

In current design of PeerBot all Workers have equal roles. An improved design could give certain Workers more responsibility within the botnet. A Worker who is given more responsibility might have access to more resources, in a useful location, or be more trusted by the bot-master. These Workers (SuperWorkers) would be in charge of a set of ordinary Workers. SuperWorkers would each run an Ethereum client locally, allowing regular Workers to connect to Ethereum through them. Workers would store up to date information on available SuperWorkers; If a SuperWorker goes offline, then Workers can connect to a different one. The Commander communicates with the botnet through the SuperWorkers.

Figure 6.1 shows the improved hybrid P2P design

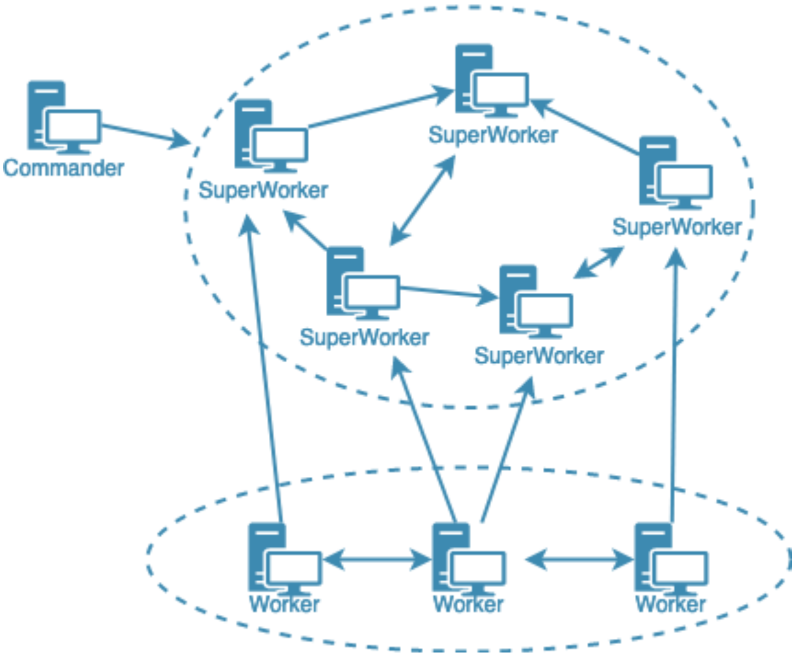


Figure 6.1 A hybrid P2P network

7. Conclusions

The purpose of this project was to build and evaluate a peer-to-peer botnet. PeerBot demonstrates how P2P architecture can be applied to botnet design. Although PeerBot avoids some of the vulnerabilities that centralised botnets are subject to, it still has design features that could be exploited. A Kademlia network avoids the single point of failure that centralised networks have, but it suffers from its own disadvantages. A kademlia network leaves the botnet open to index poisoning and sybil attacks. PeerBot's design needs to be revised in order to strengthen itself against these sorts of disruptive measure.

Although this project achieved the main objectives, the scope may have been too big. The time did not allow for a thoroughly resistant peer-to-peer network to be researched and implemented. Instead, PeerBot had to rely upon an existing library (Kademlia), to provide the peer-to-peer protocols. Given more time, PeerBots networking protocols and layout could have been designed more robustly.

PeerBot used a smart contract to intermediate between the bot-master and the botnet. This proved to be a powerful design, due to Ethereum's speed, availability and fault tolerance. However, due to time constraints, PeerBot had to rely on an "Infura" key to access Ethereum. A better implementation would have been to use local Ethereum clients.

References

Stoica, I., Morris, R., Karger, D., Kaashoek, M. and Balakrishnan, H., 2001. Chord. ACM SIGCOMM Computer Communication Review, 31(4), pp.149-160.

Maymounkov, P., Mazières, D., 2002, Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In: Druschel, P., Kaashoek, F., Rowstron, A. (eds) Peer-to-Peer Systems. IPTPS 2002. Lecture Notes in Computer Science, vol 2429. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45748-8_5

Starnberger, G., Kruegel, C., Kirda, E., 2008, Overbot - A botnet protocol based on Kademlia,

Ali, S., McCorry, P., Lee, P. and Hao, F., 2017. ZombieCoin 2.0: managing next-generation botnets using Bitcoin. International Journal of Information Security, 17(4), pp.411-422.

Muller, B., 2021, Kademlia. Available at : https://github.com/bmuller/kademlia (Accessed: April 2022)

Platdrag, 2018, Unblockable Chains. Available at: https://github.com/platdrag/UnblockableChains, (Accessed: April 2022)

Ping Wang, Sparks, S. and Zou, C., 2010. An Advanced Hybrid Peer-to-Peer Botnet. IEEE Transactions on Dependable and Secure Computing, 7(2), pp.113-127.

Fowler, M., 2022. Python Concurrency with Asyncio. New York: Manning Publications Co. LLC.

Wikipedia, 2022, Kademlia. Available at : https://en.wikipedia.org/wiki/Kademlia, (Accessed: April 2022)

Wikipedia, 2022, Distributed Hash Tables. Available at:
https://en.wikipedia.org/wikiDistributed_hash_table, (Accessed: April 2022)

Ethereum, 2022, Smart Contracts. Available at:
https://ethereum.org/en/developers/docs/smart-contracts/, (Accessed: April 2022)

Infura, Available at : https://infura.io/ (Accessed: April 2022) (used for generating API key to access Ethereum)

freeCodeCamp.org, 2021, Solidity, Blockchain, and Smart Contract Course – Beginner to Expert Python Tutorial, Available at:
https://www.youtube.com/watch?v=M576WGiDBdQ (Accessed: April 2022)

Ethereum, 2022, Available at:
https://ethereum.org (Accessed: April 2022)