A Flexible Architecture for Secure and Anonymous Web Crawling

School of Engineering in Computer Science

Master of Science in Engineering in Computer Science (MSE-CS)

Candidate:

Michele RULLO

1328929

Supervisor:

Leonardo QUERZONI

Academic Year 2015/2016

Acknowledgement

I wish to thank Giada, from the bottom of my heart, for being present in the most important decisions of my life.

I dedicate this work to my brother, Pierpaolo: never be afraid, what seems impossible now it will be possible tomorrow.

To my parents: thank you, for never giving up on me.

Finally, thanks to all those wonderful people who played and keep playing a fundamental role in my life.

Abstract

It is a fairly common issue, in the wide context of Web Crawling, of having our crawler banned or even actively attacked while performing its operations. In extreme cases, it might be even possible to have our crawling target inclined to take legal (or extra-legal) actions against us, due to information retrieval from restricted domains. This thesis project aims to solve such issues, presenting a novel technique to add an anonymity layer to our web crawler while making it resilient with respect to direct attacks. A software architecture is presented, giving a full fledged, modular, anonymous and flexible test environment.

Contents

Chapter 1

Introduction

Around the 40% of the entire world population has an Internet connection today, and we expect this number to grow in the near future. As more people get connected every day, Internet established a real revolution for information exchange. In this fast-paced context, it has become crucial to protect data from potential attacker and/or eavesdropper. To address this problem, security measures have to be taken into account. In order to secure our system we must ensure three particular properties:

- *Confidentiality*: protecting data from being accessed from wrong parties. Encryption, authentication, biometric verification ecc. are the most common methods to ensure confidentiality.

- *Integrity*: ensuring data trustworthiness guaranteeing it has not been actively tampered. Integrity is met relying on checksums.

- *Availability*: maintaining the system in a state such that data can constantly be accessed performing hardware and software maintenance. Backups, RAID, redundancy, high-availability clusters ecc. are usual methods to guarantee availability.

However, even if we might consider a system guaranteeing this three characteristics "secure", there is a fourth property that is not covered from the CIA model: *Anonymity*. By *Anonymity* we refer to the capability of a system to guarantee the non-disclosure of the user identity. Indeed, Anonimity is the main subject of this thesis, in the specific context of *web crawlers*.
A web crawler is an internet bot which automatically retrieves informations from websites, social networks ecc. It is a very frequent scenario that a web crawler retrieves confidential information, especially in the context of intelligence operations. Therefore, it becomes crucial to protect its identity. In the case where no anonymity measures are taken, the undesiderable effects might result in a simple ban, or, in extreme cases, active attacks or (il)legal coercions.
A software architecture is proposed, realized as a perfect environment for anonymous web crawler, designed to be resilient with respect to active attacks, leaving no trace of previously accomplished tasks.
An introduction to anonymous networks will follow, focusing on a specific network named *Tor* in particular. Furthermore, an experimental evaluation of Tor is given, in order to have a clear picture of how the proposed architecture will perform.
Discussing about Tor, a threat model will be covered, to picture an hy-

pothetical adversary attempting to attack the crawler.
This work ends discussing the results achieved, and the actual limitation of the current architecture version.

1.1 Anonymous Networks

"An anonymous P2P communication system is a peer-to-peer distributed application in which the nodes or participants are anonymous or pseudonymous. Anonymity of participants is usually achieved by special routing overlay networks that hide the physical location of each node from other participants." [1]

Considering the modern network architecture based on the OSI model, it is known that the user identity is univocally identified by his IP address, according to the TCP/IP Internet protocol suite. Hence, to properly ensure anonymity, the user IP address must be hidden.
One might easily assess how confidentiality does not concern the obfuscation of the IP address, since its encryption would make packet forwarding (routing) impossible.
This is where networks like *Tor* come into play, by providing a network infrastructure that guarantees anonymity (under certain circumstances) which is totally transparent with respect to the TCP/IP stack. It comes natural to think about "why" would be so important to hide our identity while surfing the web. The answer is obvious in the case that its disclosure might be dangerous for the user (whistleblowers, dissidents ecc.), but why would the average user might want to protect his identity? There are many answers to this question. Without deepening in this topic (which would take a whole paper) we simply remind that *"If*

You're Not Paying for It; You're the Product", in the sense that many companies are actually interested in collecting your personal data while you navigate the web with or without your consensus (and they successfully do it on a daily basis).
In the next chapter an insight of the Tor network is presented, introducing the basic concepts to understand in order to extensively comprehend its limitations.

Chapter 2

Tor Architecture and Internals

Tor (The Onion Router) is a network that aims to guarantee *anonymity* to the user. At the base of its functioning there is a number of computer nodes (Tor relays) that take care of delivering network packets to the final destination. Therefore the information firstly passes through a number of Tor relays before reaching the destination, so that the receiver is not able anymore to know who is the original sender. Before transmitting a packet, the user willing to take advantage of Tor, establishes a *Tor circuit* by randomly choosing three Tor relays. After the circuit has been set, the user encrypts the network packet three times, using the public keys of the circuit nodes (starting from the last relay). As soon as a circuit relay receives a packet it simply decrypts the top layer (it *peels the onion*) using its private key and successively forwards the packet to

the next relay. This guarantees anonymity, since the intermediate nodes are only able to decrypt the top layer of the packet, being unable to trace back the original sender and discovering the actual end-point of the communication.
Tor allows to establish an arbitrary-length circuit. However, to reduce the performance overhead, the default length is set to the minimum number which guarantees anonymity, that is 3.
It is not known if a longer number actually improves security/performance balance. As Hotpets-Bauer paper states:

"Choosing a path length for low latency anonymous networks that optimally balances security and performance is an open problem." [2]

To further increase anonymity, Tor shuffles the circuit every 10 minutes.
One might compare Tor to the real world by imaging to walk from A to B in a very crowded place, changing direction multiple times before reaching the final destination.
It comes natural to think how performance is affected by this architecture. To partially solve this issue, Tor relays are chosen based on their bandwidth capabilities. As we will see, this can represent a vulnerability which might be exploited to de-anonymize users.

As mentioned in the official Tor design document:

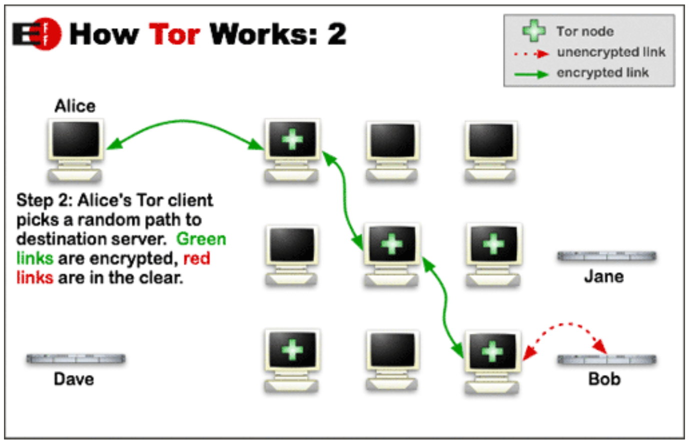

How Tor Works: 2
Tor node
unencrypted link
encrypted link
Alice
Step 2: Alice's Tor client picks a random path to destination server. Green links are encrypted, red links are in the clear.
Jane
Dave
Bob

Figure 2.1: Tor Scheme

"This second-generation Onion Routing system addresses limitations in the original design by adding perfect forward secrecy, congestion control, directory servers, integrity checking, configurable exit policies, and a practical design for location-hidden services via rendezvous points. Tor works on the real-world Internet, requires no special privileges or kernel modifications, requires little synchronization or coordination between nodes, and provides a reasonable tradeoff between anonymity, usability, and efficiency." [3]

Under a user-perspective, each user uses a local software called *Onion Proxy*, which takes care of managing Tor connections. Each onion router maintains a private long-term key and a public short-term key, according to the Diffie-Helmann approach. To provide confidentiality, the private key is used to sign TLS certificates, while the short-term keys are used when communicating between onion routers. It is worth mentioning that the short-term keys are periodically rotated, to further increase security.

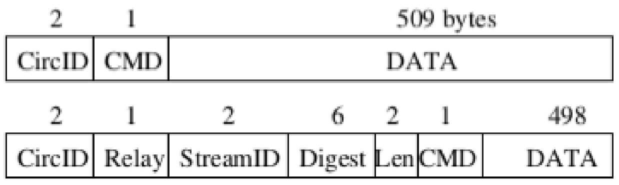


Figure 2.2: Tor Cell

Tor relays exchange fixed-size messages named *cells* via TLS connections. Cells have a size of 512 bytes, and they are of two types: *control cells* and *relay cells*, both consisting of a payload and a header. A *relay* cell carry end-to-end data, while a *control* cell is used to give instructions to the node that receives it. In particular there are 4 types of control commands for a control cell:

- *padding*: currently used for keepalive.
- *create* and *created*: used to set up/acknowledge a new circuit.
- *destroy*: used to tear a circuit down.

There are 10 types of commands for a relay cell:

- *relay data*: for data flowing down the stream.
- *relay begin*: to open a stream.

- *relay teardown*: to close a stream.
- *relay connected*: to notify the success of a connection opening.
- *relay extend*: to extend a circuit by a hop
- *relay extended*: to acknowledge a *relay extend* command.
- *relay truncate*: to tear down only a part of the circuit.
- *relay truncated*: to acknowledge a *relay extended* command.
- *relay sendme*: for congestion control.
- *relay drop*: used to implement long-range dummies.

The header is composed of several fields:

- *Circuit ID* (circID): since many circuits are associated to a single TLS connection, each cell contains a circID number wich univocally identifies the circuit to which they are associated to.
- *Command* (CMD): the operation to execute with the cell payload.
- *Stream ID* (streamID): univocally identifies a stream belonging to a certain circuit (identified by circID).
- *Digest*: checksum to ensure end-to-end data integrity.
- *Length* (Len): size of payload data.

While a *relay* cell contains all the above mentioned metadata, only the circuit ID and a command are contained into a *control* cell.
The actual establishment of a circuit is accomplished taking advantage of Diffie-Hellman exchange, sending control and relay cells among the involved nodes. For a deep explanation of how the circuit is constructed please refer to the Tor design document. It is worth mentioning that all the TCP requests/responses are accomplished using the SOCKS protocol, which makes possible the existence of TCP streams across multiple proxies.

2.1 Threat Model

It is important, under the anonymity point of view, to define what are the actual capabilities of an attacker. We remark how the goal of a malicious user concerns de-anonymization rather than decrypting flowing packets. Focusing on general issues is useful, but it is not sufficient to outline a realistic threat.

In this section two types of attacker are analyzed: *single-node attacker* and *multi-node attacker*. Both the models are interested in disclosing the original IP address of the sender.

2.1.1 Single-Node Attacker

A *single-node attacker* owns a single Tor relay. He can be *active* and/or *passive*, and while he can be part of our Tor circuit we also consider the case in which he is the actual destination.

We assume, with no loss of realism, that TLS keys cannot be stolen, and *replay* attacks are unfeasible. Such assumptions are motivated by the fact that Tor periodically rotates TLS keys, and the replay of a handshake between two Tor relays will result in a different negotiated session key[3]. Furthermore, the hypothesis of an *iterated compromise* attack is questionable. This type of attack consists of the compromission (by system intrusion, legal coercion, or extralegal coercion) of the relays that are part of our circuit. This threat is made unlikely by the fact that Tor switches circuits every 10 minutes, so that it becomes unrealistic

to assume the malicious user has enough resources to perform such an attack.
We identify four different scenarios:

- The attacker is a *guard* node: if end-to-end traffic is not encrypted the attacker is able to read packets that are flowing back from our target. This affects privacy, but it might lead to user de-anonymization since network packets may contain user information. The presence of a proxy between the crawler and the Tor network mitigates the issue, as well as establishing a secure end-to-end protocol.

- The attacker is an *intermediate* node: This is the worst case scenario for the attacker. Being an intermediate relay means that, in every case, traffic passing through us is encrypted by Tor (recall *"onion"* encapsulation).

- The attacker is an *exit* node: this case is similar to the first one, in which the attacker impersonates a guard node. If end-to-end traffic is not encrypted he is able to sniff traffic going toward the recipient.

- The attacker is the *crawling target*: obviously, the attacker is able to decrypt our traffic. As long as it does not contain any information regarding the crawler identity we are safe. Unless he is able to monitor our crawler network adapter he is not able to correlate and de-anonymize it.

It is worth mentioning that traffic analysis techniques are quite ineffective, unless the attacker is able to directly sniff incoming/outgoing traffic to/from our crawler and the target.

2.1.2 Multi-Node Attacker

A *multi-node attacker* owns an arbitrary number of nodes, allowing him to perform more complex attacks. A list of the most important attacks are presented.

2.1.2.1 Malicious exit/guard nodes

Due to the fact that eventually a network packet must enter (exit) the Tor network, one of the major issue regards who controls the guard (exit) node. Assume that an attacker both owns a guard and an exit nodes. If the end-to-end connection between Alice (our crawler) and Bob (our target) is not encrypted the attacker may inject "tags" (e.g. an HTTP tag/comment) inside the network packets that flow between Alice and Bob. Exploiting this technique, the attacker may de-anonymize Alice correlating the traffic that flows between the guard and the exit nodes by means of the previously injected tags.
This issue can be mitigated forcing end-to-end encryption (TLS/SSL or VPN) or checking integrity (hashes). However, since this might not always be possible (e.g. Bob does not support secure protocols), a proxy between Tor and Bob might be used.

Another solution might be running a proxy as a bridge between Alice and the Tor network.

2.1.2.2 Traffic Analysis

Recent researches point out how traffic analysis is a powerful tool which can successfully lead to user de-anonymization. Several complex attacks have been researched using this technique, and some of them have been proved to be extremely effective[5][7]. Most of these kind of attacks rely on traffic correlation and pattern analysis. Observing the flowing encrypted data, an attacker owning a certain number of malicious nodes may de-anonymize a particular user by observing data pattern.
In certain situations, the attacker might also overload certain nodes to route Alice traffic toward nodes he owns.
Tor does not guarantee protection against end-to-end timing correlation. Assuming the attacker is sniffing Alice and Bob at the same time, he might be able to correlate traffic by observing request/response pattens. The effectiveness of these kind of attacks highly depends on the amount of resources owned by the attacker. Indeed, assuming the attacker owns a high number of nodes in the network (i.e. *Sybil* attack), it becomes easier to successfully identify traffic. To mitigate this problem, data pattern must be concealed. In the specific case of a net crawler, a solution might be to simulate a human-like traffic, adjusting packets flow in a proper way. It is worth mentioning that Tor relays communicate exchanging *cells*, which are fixed-size packets (512Kb).

2.1.2.3 Sybil Attacks

In the case of an entity owning a significant number of relays, it might be straightforward to successfully de-anonymize users. On July 4 2014 a cluster of nodes successfully delivered a Sybil attack:

"[...]they signed up around 115 fast non-exit relays, all running on 50.7.0.0/16 or 204.45.0.0/16. Together these relays summed to about 6.4% of the Guard capacity in the network. Then, in part because of our current guard rotation parameters, these relays became entry guards for a significant chunk of users over their five months of operation." [4]

Fortunately, the issue is mitigated by periodical monitoring of Tor network. A service called *DocTor* (https://gitweb.torproject.org/doctor.git) is in charge of scanning the network, in order to identify potential malicious relays. To further improve security, Tor automatically builds circuits choosing relays that are, as much as possible, geographically distant. Such a solution can be ineffective in the case of collaborating ISP's (or agencies) that take control of a certain number of Tor relays.

2.2 Defending from Node Compromission

While it is important to take into account vulnerabilities of Tor architecture, we should also consider the possible compromission of a Tor relay, or, in the worst case, of our Crawler/Proxy. Defending the controllable perimeter should not be overlooked, since successful attacks to our nodes might lead to the disclosure of our identity. Operating systems like *Tails*[12] or *Whonix*[11] aim to avoid de-anonymization even if the machine is compromised.
In particular, *Whonix* is characterized by an interesting design. As shown in the figure, it is composed of two modular blocks: a *Whonix-Workstation* and a *Whonix-Gateway*. While the former is the actual OS, the latter specifies the network protocol used. As mentioned in the official documentation:

"Whonix is divided into two parts: Whonix-Workstation for your work and Whonix-Gateway for automatically routing all internet traffic through Tor. This is security by isolation, and it averts many threats posed by malware, misbehaving applications, and user error."[11]

Hence, it basically works enforcing all the traffic coming from the *Workstation* to the *Gateway*, ensuring that no packets will flow using a different network protocol.
As we will see, the structure of *Whonix* inspired the design of the archi-

tecture presented in this thesis work.

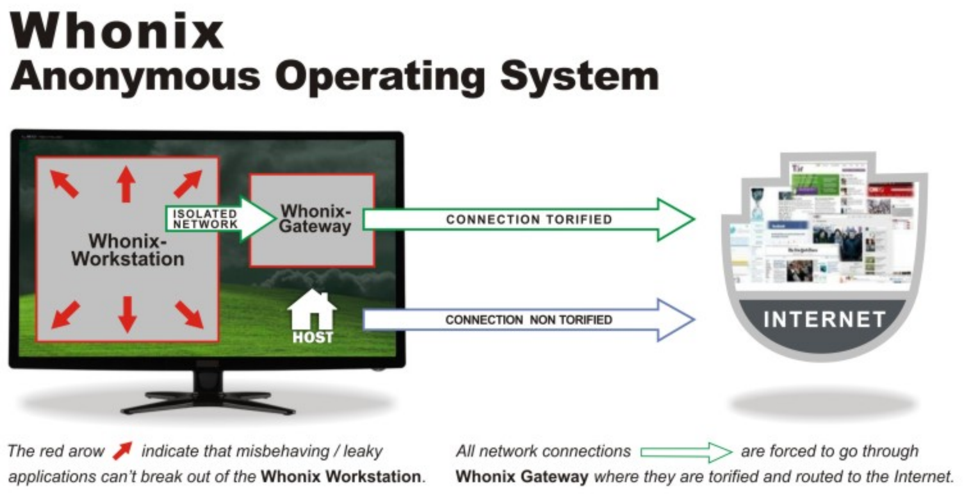

Whonix
Anonymous Operating System
ISOLATED NETWORK
Whonix-Workstation
Whonix-Gateway
HOST
CONNECTION TORIFIED
CONNECTION NON TORIFIED
INTERNET
The red arow indicate that misbehaving / leaky applications can't break out of the Whonix Workstation.
All network connections are forced to go through Whonix Gateway where they are torified and routed to the Internet.

Figure 2.3: Whonix

While *Whonix* is thought to be installed on static machines, *Tails* is thought to be portable:

"Tails is a live system that aims to preserve your privacy and anonymity. It helps you to use the Internet anonymously and circumvent censorship almost anywhere you go and on any computer but leaving no trace unless you ask it to explicitly."[12]

As it will be discussed later, the software architecture developed for this thesis it is designed to be resilient with respect to successful direct attacks. However, one might decide to let it run on a *Whonix* or *Tails* machine depending on the user needs.

2.3 Hidden Services Vulnerabilities

A *Hidden Service* is a server which hosts web services within the Tor network. It is uniquely identified by an alphanumeric hash ending with *.onion*.
Now, assume Alice wants to connect to Bob's hidden service. Both parties want to stay anonymous and communicate in a secure way. It is clear that Alice and Bob must talk in an *indirect* manner. Setting up such a Tor circuit involves six steps:

- Prior to the actual communication, Bob chooses a set of *introduction points* (i.e. some Tor relays) and builds Tor circuits to them.
- In order to advertise his service, Bob communicates to the Directory Server (through a circuit) the information about it.
- Alice is now able to contact the directory server and gets the information about Bob's hidden service. Furthermore, Alice picks randomly a relay which will act as a "bridge" between her and Bob. This relay is named *rendezvous point*. Alice shares a one-time secret with the rendezvous point.
- Alice sends the one-time secret and the rendezvous address to Bob, encrypting the message using Bob's public key. Alice picks one *introduction point* in order to indirectly send the message to Bob.

- Bob connects to the rendezvous point providing the one-time secret.
- Alice and Bob are now able to communicate through the rendezvous point.

Since Alice and Bob communicate through a set of Tor relays, an attacker might decide to subvert them in order to de-anonymize Alice and/or Bob; therefore, we have to include the attacks against directory server, introduction and rendezvous points in the definition of our threat model. Several type of attacks are specified in the official design document.
A sophisticated attack against Tor hidden services proposed by Kwon, AlSabah, Lazar, Dacier, Devadas is based on traffic analysis[5]. Machine learning algorithm have been used (support vector machine) to correlate traffic and de-anonymize users. Important results have been achieved:

"We found that we can identify the users' involvement with hidden services with more than 98% true positive rate and less than 0.1% false positive rate with the first attack, and 99% true positive rate and 0.07% false positive rate with the second. [...] we show that we can correctly determine which of the 50 monitored pages the client is visiting with 88% true positive rate and false positive rate as low as 2.9%, and correctly deanonymize 50 monitored hidden service servers with true positive rate of 88% and false positive rate of 7.8% in an open world setting."

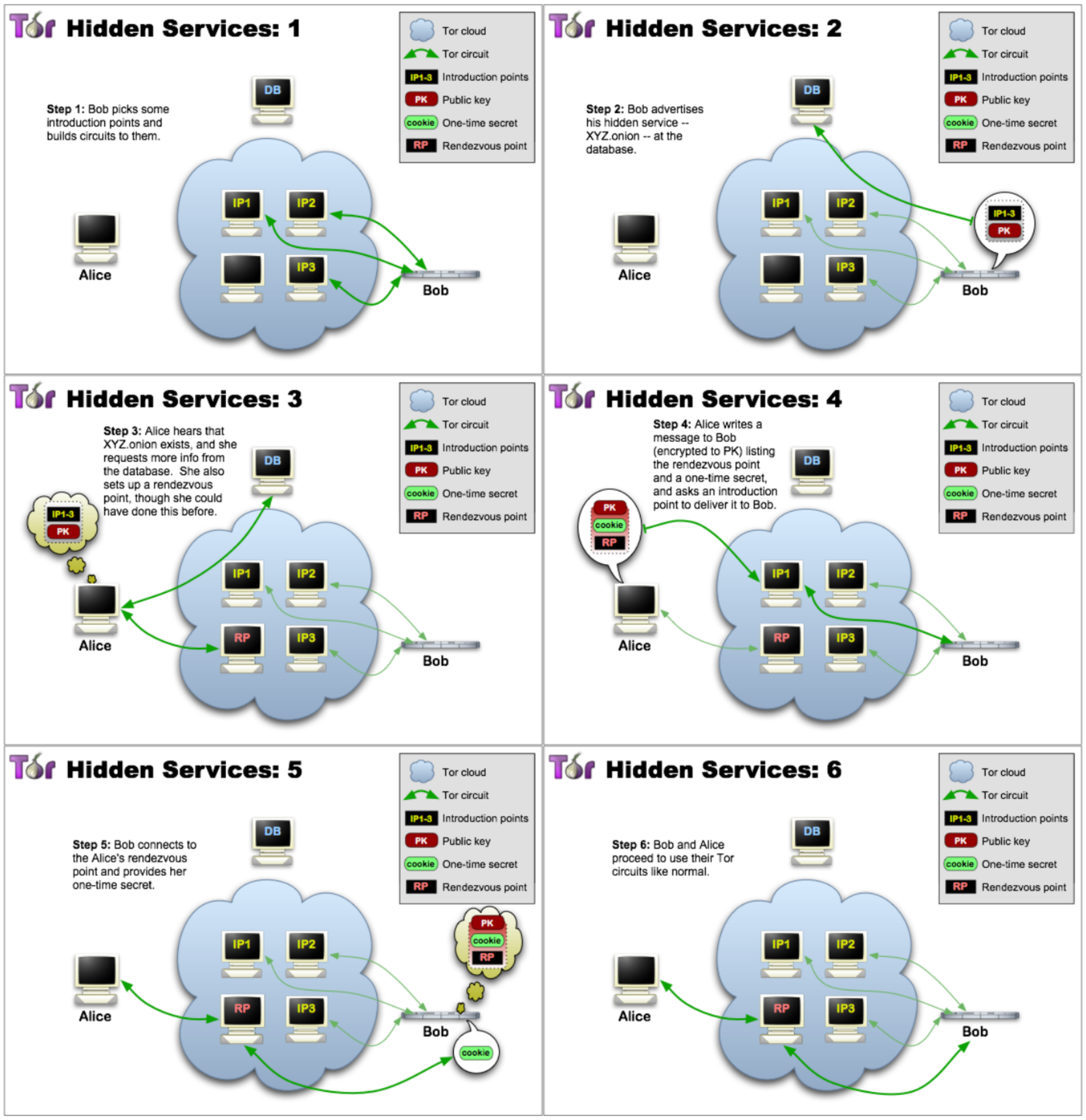

Hidden Services: 1
Step 1: Bob picks some introduction points and builds circuits to them.
Hidden Services: 2
Step 2: Bob advertises his hidden service -- XYZ.onion -- at the database.
Hidden Services: 3
Step 3: Alice hears that XYZ.onion exists, and she requests more info from the database. She also sets up a rendezvous point, though she could have done this before.
Hidden Services: 4
Step 4: Alice writes a message to Bob (encrypted to PK) listing the rendezvous point and a one-time secret, and asks an introduction point to deliver it to Bob.
Hidden Services: 5
Step 5: Bob connects to the Alice's rendezvous point and provides her one-time secret.
Hidden Services: 6
Step 6: Bob and Alice proceed to use their Tor circuits like normal.
Tor cloud
Tor circuit
IP1-3 Introduction points
PK Public key
cookie One-time secret
RP Rendezvous point
DB
IP1
IP2
IP3
RP
Alice
Bob

Figure 2.4: Hidden Service

2.4 Tor Performance Evaluation

Due to Tor complexity, it comes natural to think about how performance is affected by the high amount of overhead needed to handle the circuits. A summary of [6] illustrates the main issues about Tor performance.
In the mentioned paper, six reasons for low performance have been identified:

- Tor flow control does not work properly, meaning that low-traffic streams (as web browsing) does not co-exist well with high-traffic ones (bulk transfers).
- Tor relays put too much traffic inside the network with respect to the traffic they actually forward.
- Tor network capacity is not enough to guarantee anonymity to current amount of users.
- Current path selection algorithms have to be improved, since some relays are overloaded with respect to others that are underloaded.
- *"Tor clients aren't as good as they should be at handling high or variable latency and connection failures. We need better heuristics for clients to automatically shift away from bad circuits."*[6]
- *"[...]low-bandwidth users spend too much of their network overhead downloading directory information."*[6]

Tor performance has an important impact on our crawler architecture, since we have to know in advance how much if a certain crawling task can or cannot be accomplished in a reasonable time.
As a first option an analytical approach has been considered to properly model Tor performance (queueing theory), but due to the highly heterogeneous nature of Tor (being composed of very different type of relay under a performance point of view), it would not have led to interesting results.
At this point, the option to lead empirical tests directly on the Tor network has been designed as the best hypothesis.

2.4.1 Tor Network Tests

A virtual machine from the University department has been set up to download test files from a specific url. Tests have been organized in the following way:

- *For each day*: execute tests at 10 a.m., 12 a.m., 2 p.m, 4 p.m
- *For each test*:
  - Download 1 Kb test file using / without using Tor
  - Download 1 Mb test file using / without using Tor
  - Download 10 Mb test file using / without using Tor
- *Repeat for 4 consecutive days*

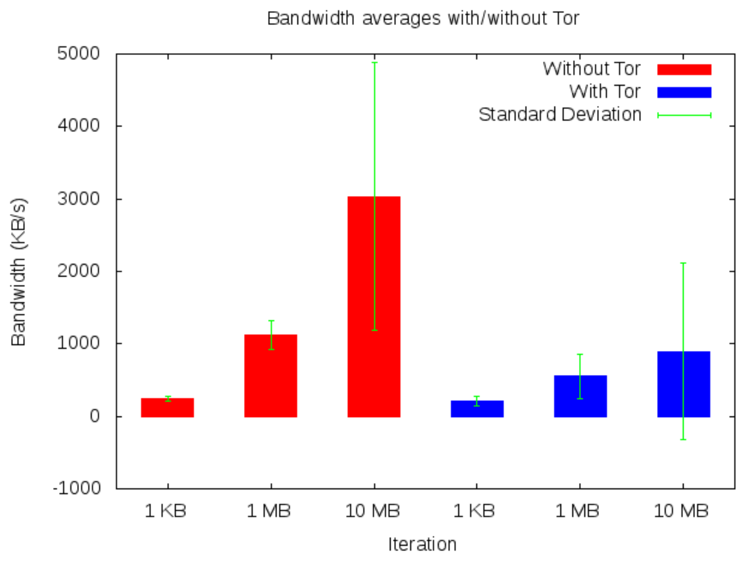


Figure 2.5: Bandwidth Test - Day 1 - Average and Standard Deviation

The tests surprisingly led to interesting results. As one may notice from the picture, there are no substantial differences when downloading small sized data, while the Tor overhead has a noticeable impact on data of higher dimensions. For completeness, the average bandwidth and the standard deviation is shown, to remark the performance loss when using Tor. In the appendix, it is possible to assess how all the test results look very similar across the different days.

It is worth to underline how the packet loss measurement has been omitted, since it did not give useful information (always 0%). The obtained results had a remarkable impact on the design of our software architecture: as mentioned before, we designed the crawler to work on a virtual machine that is, basically, a proxy. To further increase anonymity, one might want to deliver tasks to it passing through an anonymous network (i.e. Tor), thus it becomes crucial to have a clear picture of how performance are affected. Indeed, how we will see in the next section, the design of our crawler takes into account these results.

Chapter 3

Architecture and Web Crawling

One of the most important aspect of writing a web crawler, concerns the anonymity of the crawler itself. While it might be easy to accomplish this relying on anonymizing network (e.g. Tor) it might not be as easier to protect the source identity when a crawler is compromised by an attacker. A simple attempt to solve this issue is to set the crawler as an external proxy, sending tasks to it covering the original source ip.
However, it is interesting to face another important issue: what if we want our crawler to be more resilient with respect to active attacks by hiding the crawling tasks we set from a remote position? In this case, we need to design a software architecture that has no "memory" about the work it does.
Another aspect the architectures aims to face is purely technical: we

want the crawler to be technology-agnostic, in such a way it can perform whatever type of task regardless the programming language or environment. Furthermore, we want to have the possibility to specify the network protocol to use for every single task. For instance, we would like to execute two tasks that contain the same crawling logic but different network specifications (e.g. crawl the target passing through Tor).

To summarize, the architecture aims to reach the following goals:

- Hide the task-provider source ip
- Avoid the disclosure of the tasks even in case of crawler compromission
- Allow the crawler to perform every kind of task (whatever technology)
- Realize a modular architecture, to decouple the crawling logic from the network transmission protocol.

3.1 Design

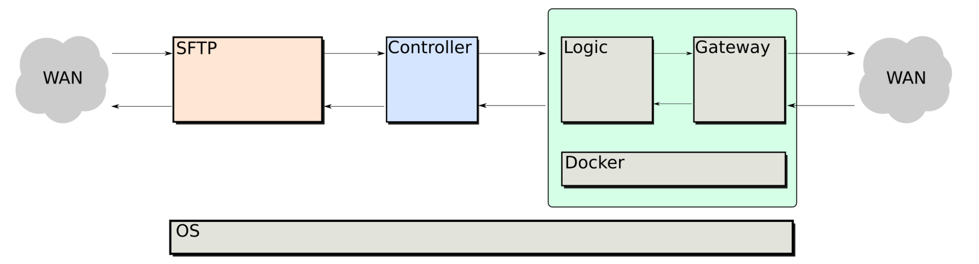


Figure 3.1: Architecture Overview

In order to realize a modular and flexible architecture, Docker is used. Docker runs as a *middleware* between the applications and the operating system, and offers the possibility to create lightweight virtual machines (called *containers*) on the fly.
As the Docker website states:

"Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in."

A container is created building a *dockerfile*, which contains the configuration needed to launch it. In the dockerfile it is possible to specify the container *image* (e.g. ubuntu) and a set of scripts needed to set it up properly. Docker implements its own caching logic to avoid re-downloading container images. This is a positive aspect with respect to performance evaluation, since we can avoid sending the whole container to the crawler (which would result in a huge throughput bottleneck).

In order to launch the machine, we need to provide the following objects:

- Crawling logic container (dockerfile + scripts)
- Gateway logic container (dockerfile + scripts)
- Input files

It can be easily guessed that the *logic* container is in charge to perform the crawling logic, routing every packet to the *gateway* container, which implements the network logic to reach the crawling target.
Once all items are received, a software service running on the machine starts a *controller* that builds the containers and starts the crawling task. As soon as the crawling operation starts and the containers are set up, all the received files are removed from the disk, this is done to prevent an attacker from analyzing the crawling tasks. It could be also possible to encrypt the sftp folder, to further increase the security.
Once both the *logic* and the *gateway* containers are successfully built, the former forwards all of its packet toward the gateway container, that in turn forwards the packets toward the crawling target according to its

implementation.

The task output is then moved into the *sftp* folder, ready to be retrieved.

3.2 Task Execution Pipeline

The following steps are executed:

1. *Crawling task delivery through SFTP*
2. *Controller execution and clean-up*
3. *Task execution*
4. *Output production*

3.2.1 SFTP Input Delivery

The task files (dockerfiles and input scripts) are delivered using SFTP into a folder named *sftp*. It is important to remark that it is possible to access the machine from a remote position using Tor in order to increase our anonymity.
It is also worth to mention that the input files can be delivered taking advance of an anonymizing network like Tor.

3.2.2 Controller Execution and Clean-Up

Once all files are delivered, a script named *controller*, running as a background service, automatically checks if an empty file named *done* is present before starting the crawling task. The reason behind this is

to avoid running a task with incomplete input.
A script called *executor* is in charge to accomplish the following steps:

- Building the *logic* and the *gateway* containers by means of the dockerfiles.
- Starting the containers and linking them.
- Cleaning up the received input files to be more resilient with respect to active attacks.
- Producing the output.

3.2.3 Task Execution

The crawling logic is performed by the *logic* docker container. To avoid any kind of network leak the default gateway of the *logic* container is changed, in order to forward all of its packets through the *gateway* container. Being a fundamental step, it is forced by our controller.

3.2.4 Output Production

Once the task is completed, the *executor* stops the containers and remove their images from the docker local repository. This is done to remove any clear evidence of the completed task. The output is placed in the *sftp* folder, ready to be retrieved remotely.

For further information about the implementation, refer to the appendix at the end of this document.

Chapter 4

Validation

This chapter aims to show how the proposed architecture has been tested, illustrating the technical details needed to comprehend its functioning.
The software has been primarily tested on a CentOS virtual machine, however, thanks to Docker portability, it runs on any other platform supporting Docker. It is worth mentioning that it has also been tested on a Ubuntu virtual machine without any modification.
The main difficulty encountered lies in defining a communication standard between the *logic* and the *gateway* containers. Indeed, it is important to underline that the logic container is not aware about the actual implementation of the gateway, since it might not always be the same due to the modularity constraint we set from the beginning.
Recalling the fact that a Docker container is, in effect, a virtual machine, it has its own IP address. Thanks to the capability of Docker of linking two containers, it is possible to let them communicate as if they are on

the same network. Therefore, we have the following scenario:

- *Logic* container needs to forward all traffic to the gateway in order to reach its final destination.
- *Gateway* container acts as a *proxy*, listening on a set of pre-defined ports.

It is crucial that the ports exposed by the gateway are *always the same*, since the logic has to know where to send its traffic. Before explaining exactly what ports to expose, we first illustrate how the gateway is implemented, and what kind of traffic it expects to forward.

4.1 Gateway Container

To validate the architecture, three gateway containers have been designed and realized:

- *Pass-through*: forward packets directly to the crawling target
- *Tor*: forwards packets through the Tor network
- *VPN*: forwards packets using a VPN

They coincide with the three main types of useful proxies for a web crawler. Indeed, depending by the requirements, a configuration might suit better with respect to the others.

The *pass-through* gateway acts as an elementary proxy, forwarding all the incoming traffic from the logic container to the crawling target. It has been implemented building a Docker container which installs Squid in order to improve performance:

"Squid is a caching and forwarding web proxy. It has a wide variety of uses, from speeding up a web server by caching repeated requests; to caching web, DNS and other computer network lookups for a group of people sharing network resources; to aiding security by filtering traffic. Although primarily used for HTTP and FTP, Squid includes limited support for several other protocols including TLS, SSL, Internet Gopher and HTTPS."[9]

The *Tor* gateway pulls an image from the Docker Hub called *torprivoxy* (*arulrajnet/torprivoxy*) which natively exposes three ports:

- *9050*: for *socks5* traffic
- *8118*: for *http* traffic
- *9051*: to control Tor behavior (e.g. switch Tor circuit)

Being Tor built on top of the socks5 protocol, it might be handy for our architecture to support socks5 traffic. As we will see later, it is possible to launch a task choosing the desired gateway port (i.e. the gateway protocol). Therefore, according to this standard, port 8118 has been set, by design, as the default http port to forward traffic through the gateway container.

The *VPN* gateway container features a Squid proxy and Openvpn. For validation purposes, a free VPN service has been used.

Before introducing the validation tests, a technical overview of the logic container will follow.

4.2 Logic Container

As already mentioned, the logic container is in charge to execute the crawling task. Two tasks are executed for validation purposes:

- A *curl* command to get the machine public ip
- A *scrapy* script to execute a realistic crawling task

We are interested in knowing the public ip to check the difference between the different gateway types available. Indeed, we expect three different ip's for each gateway, witnessing the correct functioning of the architecture.

The second task involves one of the most used web crawling frameworks, *Scrapy*:

"Scrapy is an application framework for crawling web sites and extracting structured data which can be used for a wide range of useful applications, like data mining, information processing or historical archival. Even though Scrapy was originally designed for web scraping, it can also be used to extract data using AVPIs (such as Amazon Associates Web Services) or as a general purpose web crawler."[10]

The reason behind such choice, is to report a log file which is a result of an actual crawling task, effectively showing how the architecture performs in a realistic scenario. In particular, a crawling task targeting *stackoverflow.com* has been executed, retrieving a list of the most upvoted questions on the renowned website.

Once again, the logic container communicates with the gateway by means of ports 8118 and 9050.

4.3 Validation Objectives

The validation tests have been executed according to the following criteria:

- Verify the correct functioning of each gateway type, providing the logs produced by the scrapy script and checking the public ip's for each gateway type.
- Assert the absence of any network leak

It is worth to mark the importance of the second point: we want the logic container to forward *all* traffic through the gateway, no packets must be forwarded directly from the logic container to the crawling target. To accomplish this, the *executor* launches the logic container modifying the default gateway, so that no packet is able to reach the Docker network adapter and, eventually, the crawling target.
For validation purposes, *tcpdump* is used. As its name suggests, *tcpdump* prints out a description of the contents of packets on a network interface that match a certain boolean expression. In particular, a packets dump on port 53 of the docker interface (*docker0*) will be reported, showing how no DNS leaks are present.

4.4 Validation Results

The following steps are followed in order to test the architecture:

- Public ip address verification (pass-through, tor, vpn)
- Crawling task output verification
- Tcpdump leaks test

Note: only the output is reported, for an exhaustive guide on how to correctly setup and launch the architecture please refer to the appropriate section in the appendix.

The first step is to check the ip returned by querying *checkip.dyndns.org* using all the three gateway types. Obviously, we expect to have three different ip addresses.
Using the pass-through configuration, we obtain the following ip address:

```
87.16.237.48
```

For the gateway implementing Tor proxy:

```
109.163.234.5
```

One might verify how the returned ip is associated to a german ISP. Lastly, the output for the gateway implementing a vpn client:

```
176.126.237.217
```

In this case we got a romanian ip. Therefore, we have the confirmation that everything works properly.

To witness the correct functioning of the Scrapy script, the header of the output coming from the crawling task is reported:

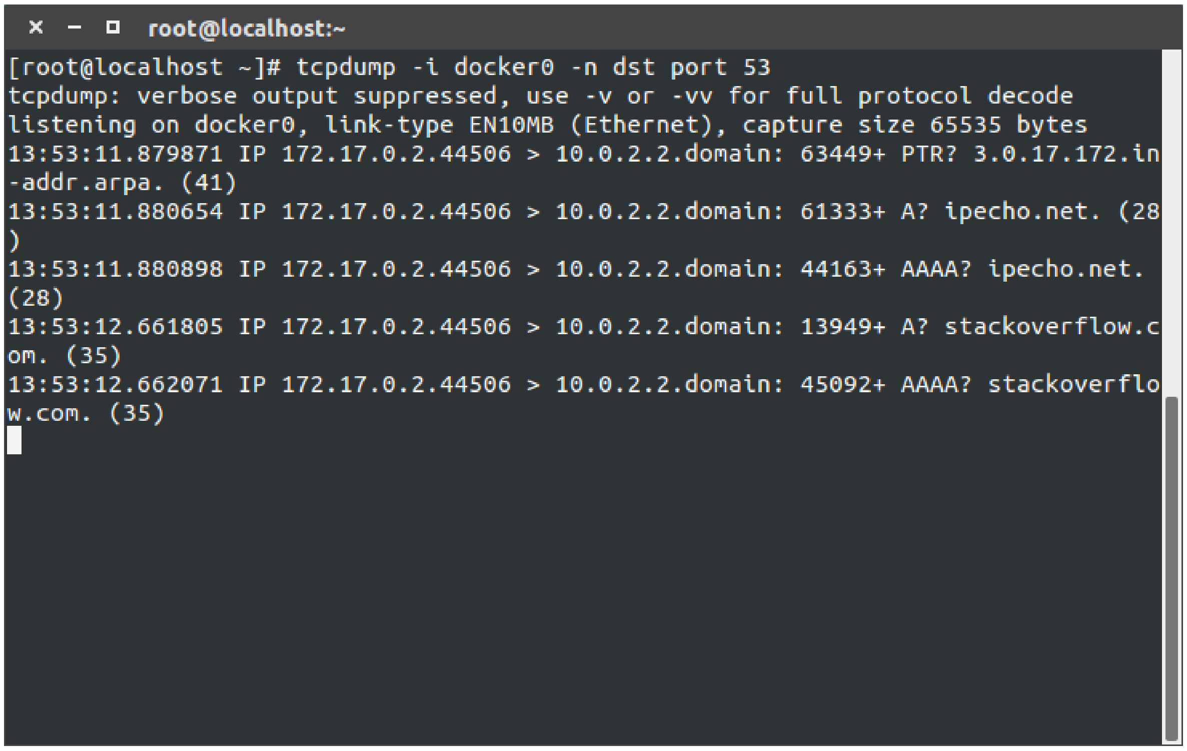
```
2016-05-09 18:41:51 [scrapy] INFO: Scrapy 1.0.6
started (bot: scrapybot)
2016-05-09 18:41:51 [scrapy] INFO: Optional features
available: ssl, http11
2016-05-09 18:41:51 [scrapy] INFO: Overridden
settings: {'FEED_FORMAT': 'json', 'FEED_URI':
'/shared/spider.json'}
2016-05-09 18:41:51 [scrapy] INFO: Enabled
extensions: CloseSpider, FeedExporter,
TelnetConsole, LogStats, CoreStats, SpiderState
2016-05-09 18:41:51 [scrapy] INFO:
Enabled downloader middlewares:
```

```
HttpAuthMiddleware, DownloadTimeoutMiddleware,
UserAgentMiddleware, RetryMiddleware,
DefaultHeadersMiddleware, MetaRefreshMiddleware,
HttpCompressionMiddleware, RedirectMiddleware,
CookiesMiddleware, HttpProxyMiddleware,
ChunkedTransferMiddleware, DownloaderStats
2016-05-09 18:41:51 [scrapy] INFO: Enabled
spider middlewares: HttpErrorMiddleware,
OffsiteMiddleware, RefererMiddleware,
UrlLengthMiddleware, DepthMiddleware
2016-05-09 18:41:51 [scrapy] INFO:
Enabled item pipelines:
2016-05-09 18:41:51 [scrapy] INFO:
Spider opened
2016-05-09 18:41:51 [scrapy] INFO:
Crawled 0 pages (at 0 pages/min),
scraped 0 items (at 0 items/min)
2016-05-09 18:41:51 [scrapy] DEBUG:
Telnet console listening on 127.0.0.1:6023
2016-05-09 18:41:51 [scrapy] DEBUG:
Crawled (200)
<GET http://stackoverflow.com/questions?sort=votes>
(referer: None)
2016-05-09 18:41:52 [scrapy] DEBUG:
```

```
Crawled (200)
<GET http://stackoverflow.com/questions/11227809/
why-is-processing-a-sorted-array-faster-
than-an-unsorted-array> (referer:
http://stackoverflow.com/questions?sort=votes)
2016-05-09 18:41:52 [scrapy] DEBUG:
Scraped from <200 http://stackoverflow.com
/questions/11227809/
why-is-processing-a-sorted-array-faster-
than-an-unsorted-array>
{.. RESPONSE BODY ..}
```

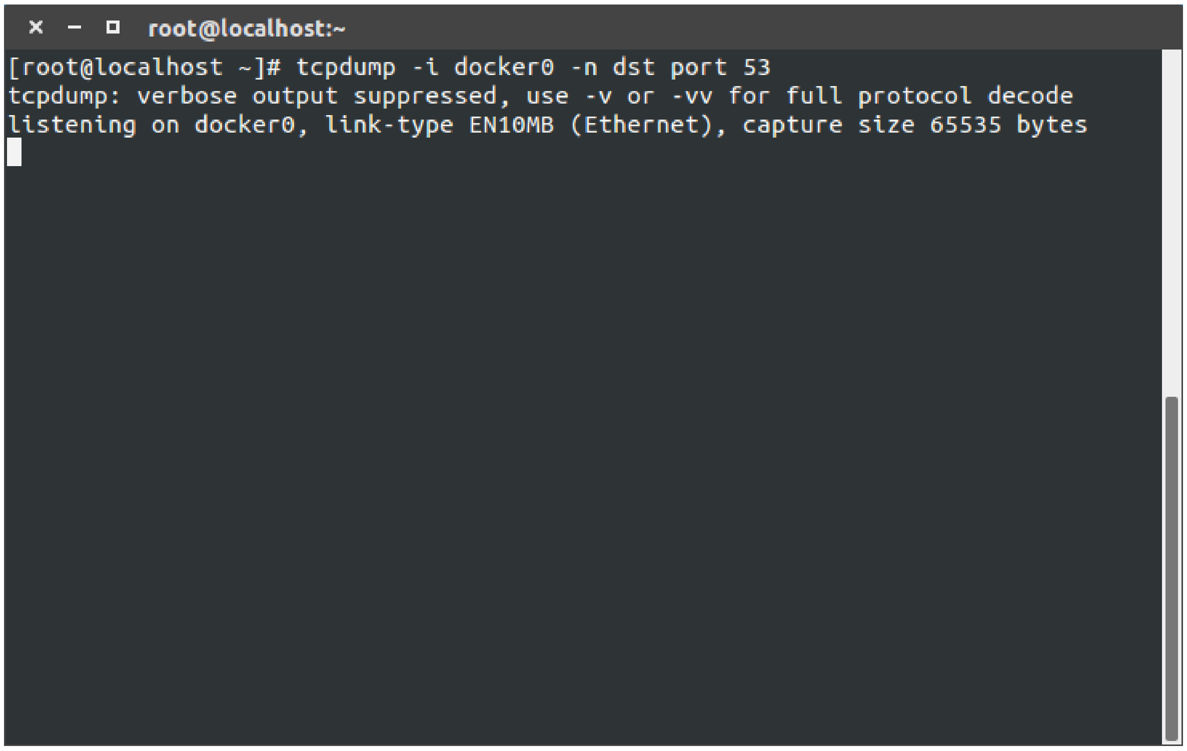
Clearly, the output reports a 200 http code to confirm that the crawling task was successful. Obviously, the output is exactly the same for each gateway type.

The last test aims to show how the architecture forwards each packet through the gateway: we expect that tcpdump shows DNS requests when using a pass-through gateway, while no logs are supposed to be shown when using a Tor or a VPN container. Indeed, in the last two cases, all the traffic is never directly routed to the crawling target.



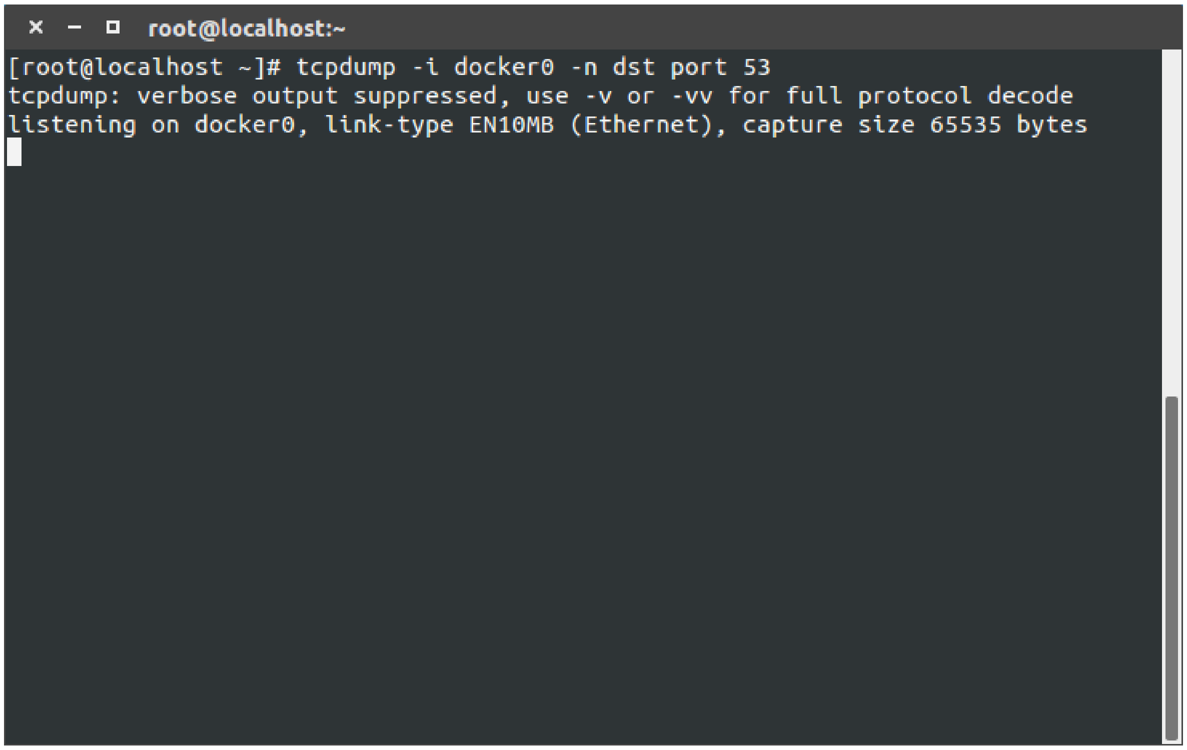
```
[root@localhost ~]# tcpdump -i docker0 -n dst port 53
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on docker0, link-type EN10MB (Ethernet), capture size 65535 bytes
13:53:11.879871 IP 172.17.0.2.44506 > 10.0.2.2.domain: 63449+ PTR? 3.0.17.172.in
-addr.arpa. (41)
13:53:11.880654 IP 172.17.0.2.44506 > 10.0.2.2.domain: 61333+ A? ipecho.net. (28
)
13:53:11.880898 IP 172.17.0.2.44506 > 10.0.2.2.domain: 44163+ AAAA? ipecho.net.
(28)
13:53:12.661805 IP 172.17.0.2.44506 > 10.0.2.2.domain: 13949+ A? stackoverflow.c
om. (35)
13:53:12.662071 IP 172.17.0.2.44506 > 10.0.2.2.domain: 45092+ AAAA? stackoverflo
w.com. (35)
```

Figure 4.1: DNS Leak Test: Pass-through Gateway



```
[root@localhost ~]# tcpdump -i docker0 -n dst port 53
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on docker0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

Figure 4.2: DNS Leak Test: Tor Gateway



```
[root@localhost ~]# tcpdump -i docker0 -n dst port 53
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on docker0, link-type EN10MB (Ethernet), capture size 65535 bytes
```

Figure 4.3: DNS Leak Test: VPN Gateway

As it possible to notice from the pictures, the results are clear: no DNS requests are captured from *tcpdump* when using the Tor and the VPN gateways. On the contrary, this is not true for the pass-through gateway.

4.5 Conclusions and Future Work

Being an experimental software architecture, it is worth to underline its main current limitations:

- *There is a time window in which the task input is actually visible.* Assuming the presence of an active attacker monitoring the crawler, it might be possible for him to analyze the crawling input before the *controller* is launched. The issue can be solved encrypting the *sftp* folder.

- *Serial tasks processing.* The actual architecture does not provide the possibility to parallelize the execution of multiple tasks. A software hypervisor might be necessary to handle multiple requests, reorganizing the pipeline structure.

- *Source ip discovery.* Assuming we are reaching the *sftp* folder without relying on an anonymizing network, it might be possible for an attacker who has already compromised our crawler to discover our source ip. An effective way to solve this issue is to use anonymous operating systems such as *Whonix*[11] or *Tails*[12].

In order to improve the architecture, a list of features are proposed:

- *Creating a repository of Logic and Gateway containers.* Thanks to the modularity and flexibility offered by the design of this thesis work, it might be interesting and useful to have a repository of containers, ready to be plugged in. Indeed, one of the most important issue for a web crawler concerns the need to receive frequent updates. A centralized repository allows to maintain and update the containers efficiently

- *Batch tasks processing.* While task parallelization has been reported as one of the main limitations, it might be handy to allow the system to enqueue multiple tasks for batch processing. Such feature might be enabled configuring the Controller module.

- *Temporary or persistent storage.* To keep the system anonymous, the architecture automatically deletes container images and input scripts. However, there could be situations in which the anonymity hypothesis can be relaxed in favor of better performance. Under these circumstances, it would be convenient to have the system accepting a configuration file as input, letting the user specifying whether if the input has to be stored (optionally specifying a time-to-live) or not.

Appendix A

Architecture: Technical Guide

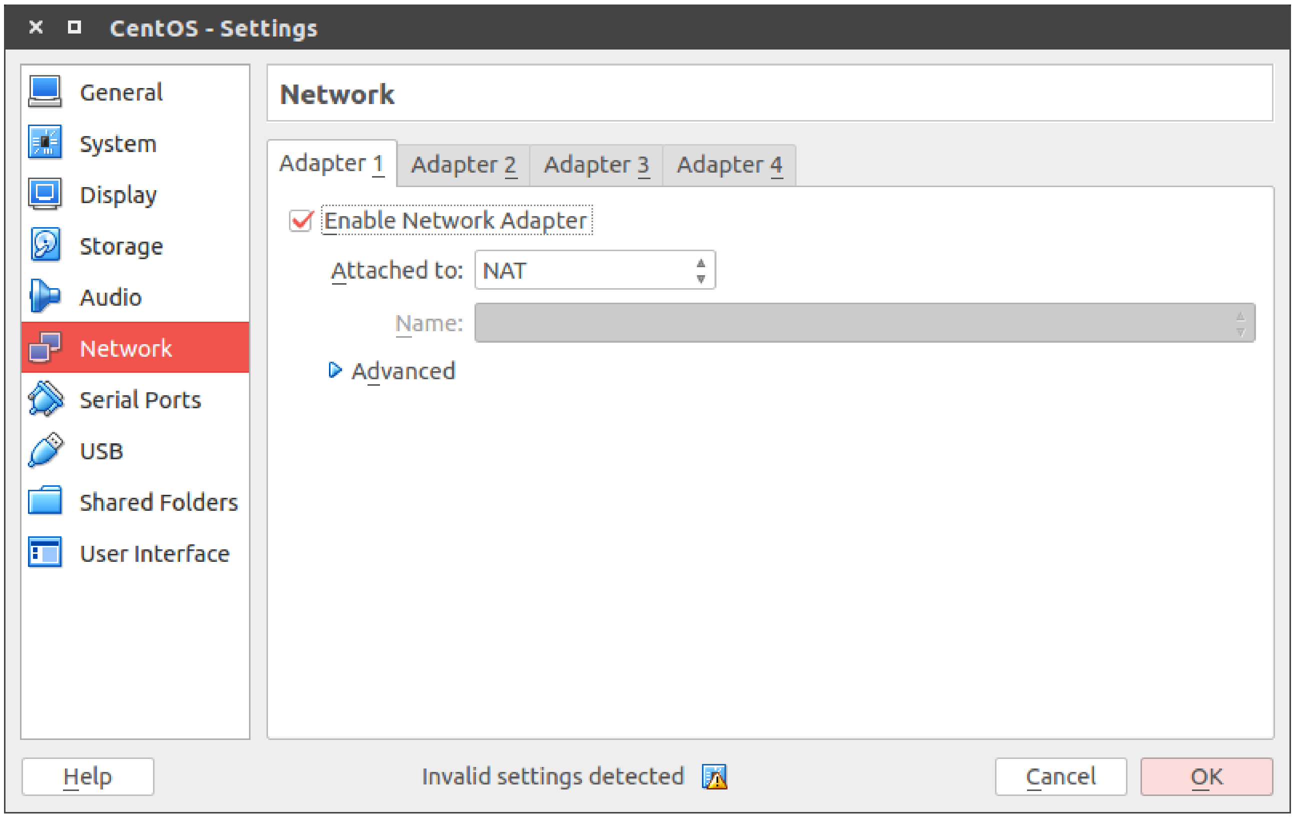
This chapter illustrates how to interact with the architecture step by step, showing how the validation test has been executed. These are the steps needed to correctly deploy the virtual machine and launching a crawling task:

- Virtual Machine (CentOS) deployment and sftp/ssh setup
- Start background service
- Input delivery
- Output retrieval

To add a further level of obfuscation, a section on how to access the architecture through Tor is reported at the end.

A.1 Architecture Deployment

As already mentioned, the architecture has been tested on a CentOS virtual machine. *Virtualbox* has been used to accomplish this. In order to have our machine reachable from sftp/ssh, we need to setup two network adapters, as shown in the pictures.



Network

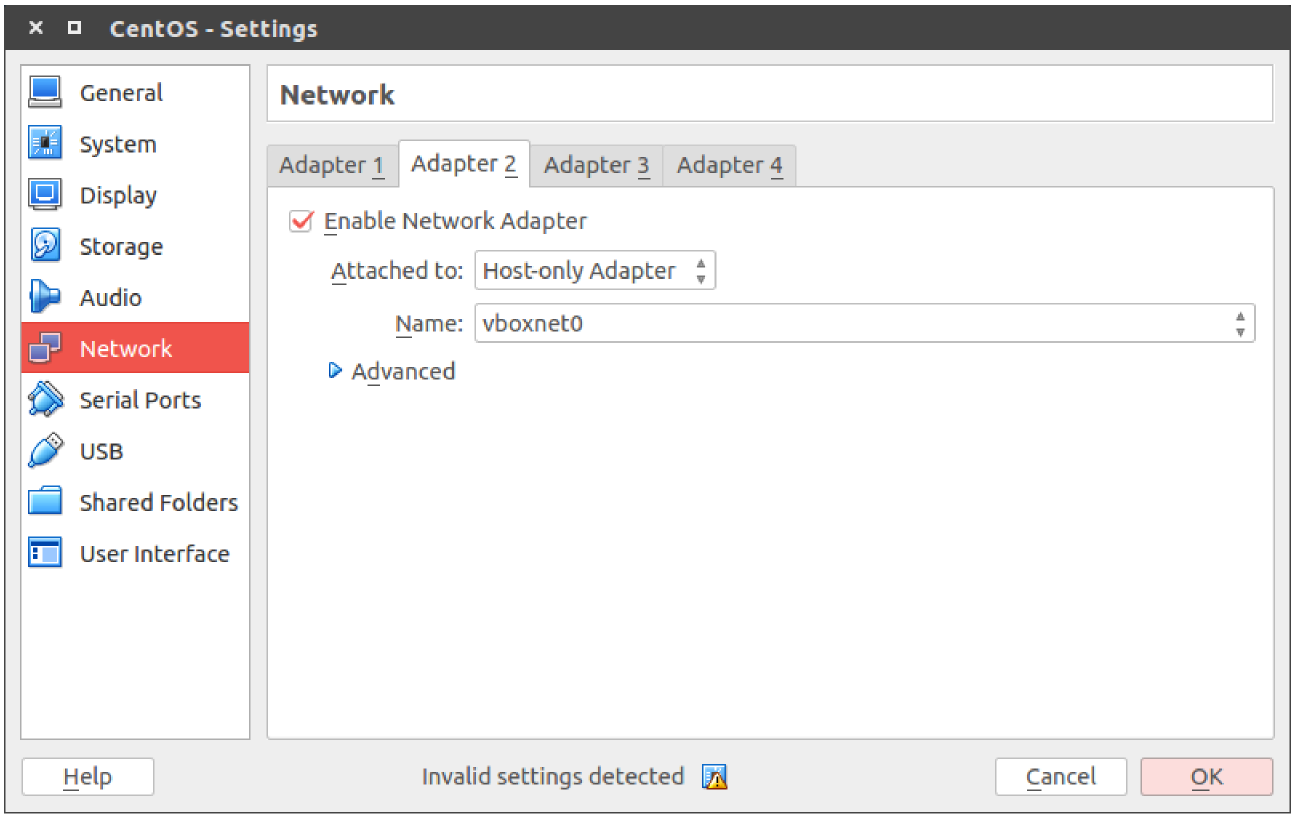
☑ Enable Network Adapter

Attached to: NAT

Name:

Invalid settings detected

Figure A.1: Virtual Machine setup: network adapter 1



Network

Enable Network Adapter ✓

Attached to: Host-only Adapter

Name: vboxnet0

Advanced

Invalid settings detected

Figure A.2: Virtual Machine setup: network adapter 2

By default, sftp and ssh daemons are launched at startup, therefore we can connect to the virtual machine using:

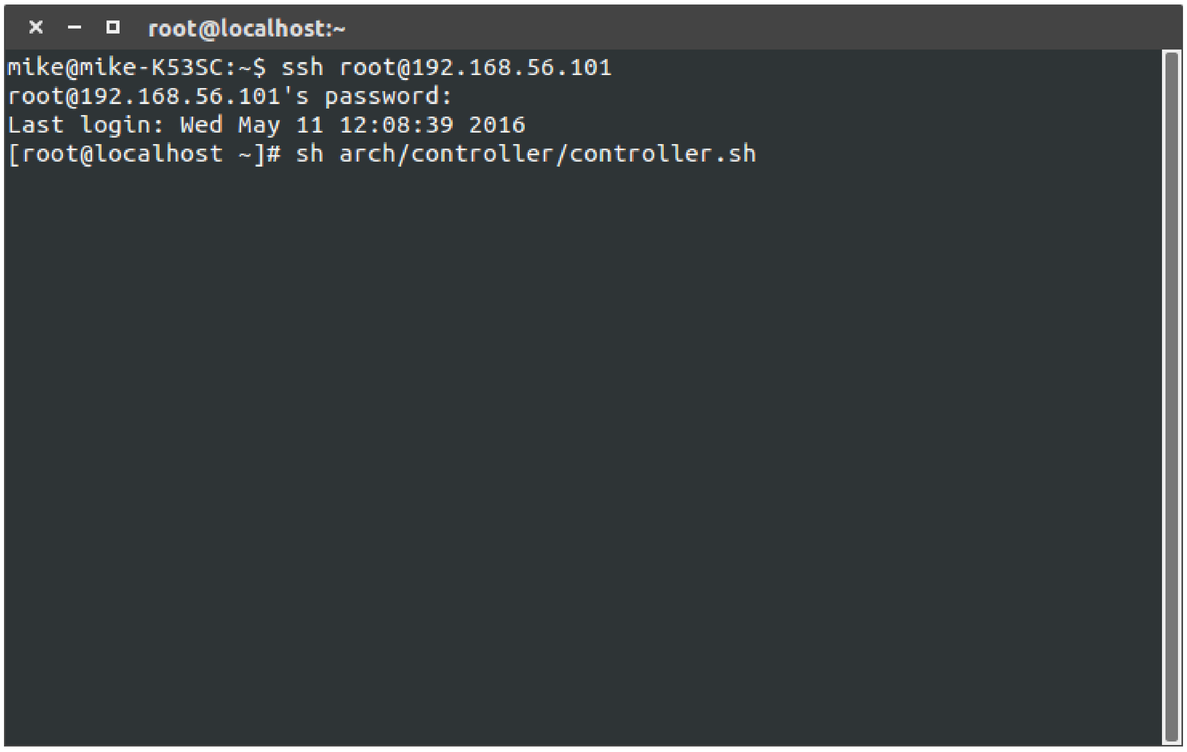
```
sftp root@<ip_address>
```

Or:

```
ssh root@<ip_address>
```

A.2 Start Background Service

Using ssh, we are able to start the virtual machine controller as shown in the picture. The controller is now waiting for the input to be delivered, once it finds the file *done* in the *sftp* folder, it launches *executor.sh* which is in charge to build the dockerfiles and launch the input task.



```
mike@mike-K53SC:~$ ssh root@192.168.56.101
root@192.168.56.101's password:
Last login: Wed May 11 12:08:39 2016
[root@localhost ~]# sh arch/controller/controller.sh
```

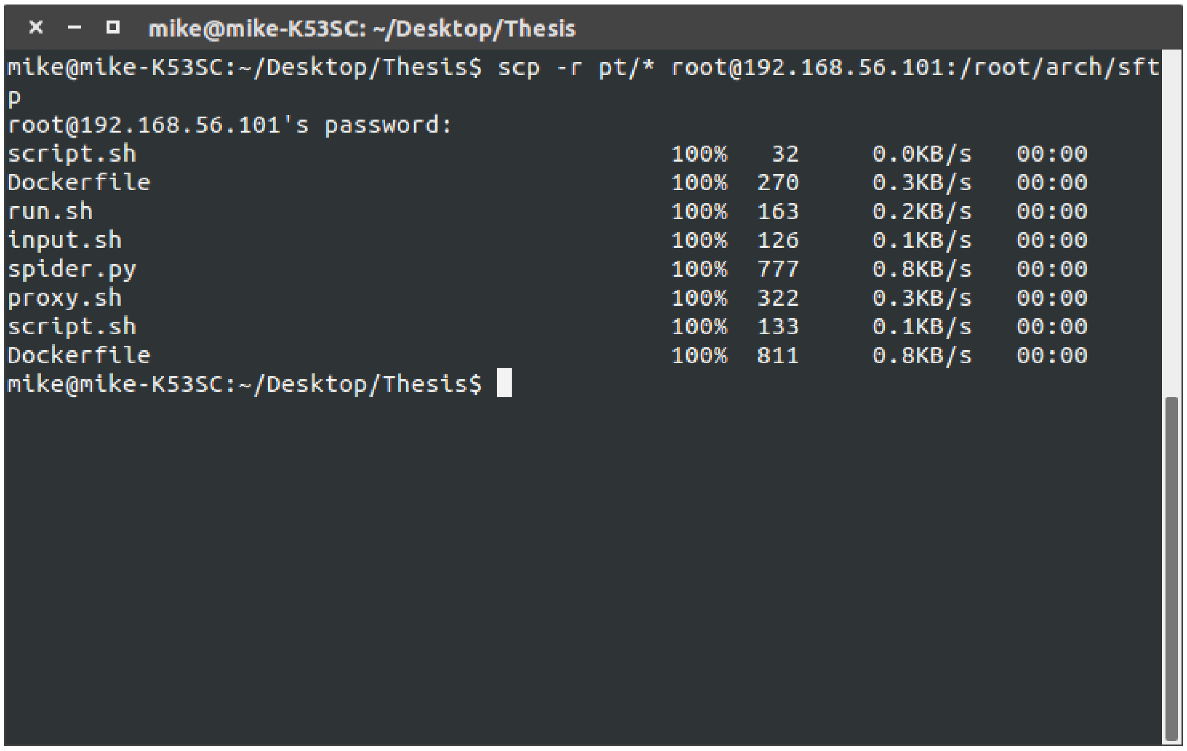
Figure A.3: Virtual Machine setup: starting background service

A.3 Input Delivery

Now, using sftp or ssh we can deliver the following items:

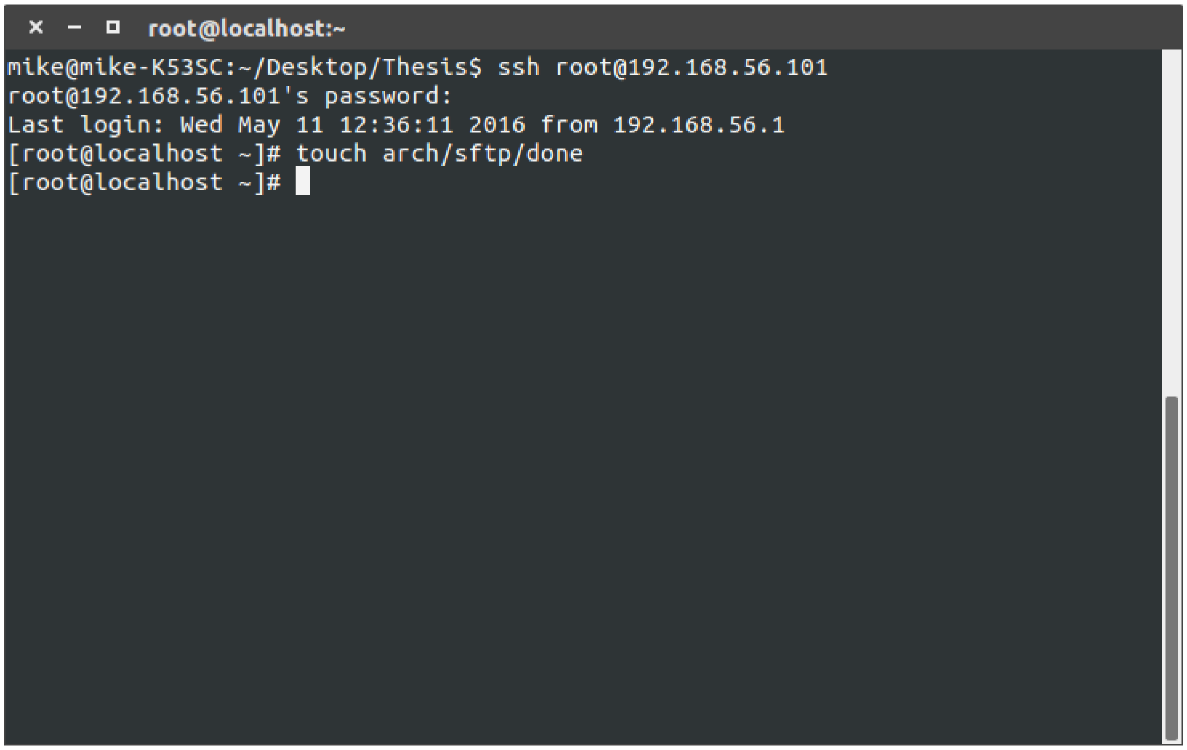
- Gateway container
- Logic container
- Input files
- "done" file

The pictures clearly shows how to accomplish this procedure. In figure A.4 we can see how the input files are delivered. Only after this has been accomplished it is possible to deliver the *done* file, since we have to be sure that all the files have been successfully delivered before starting the task. Figure A.5 shows the creation of the *done* file on the virtual machine, while figure A.6 shows that the crawling task has been successfully launched.



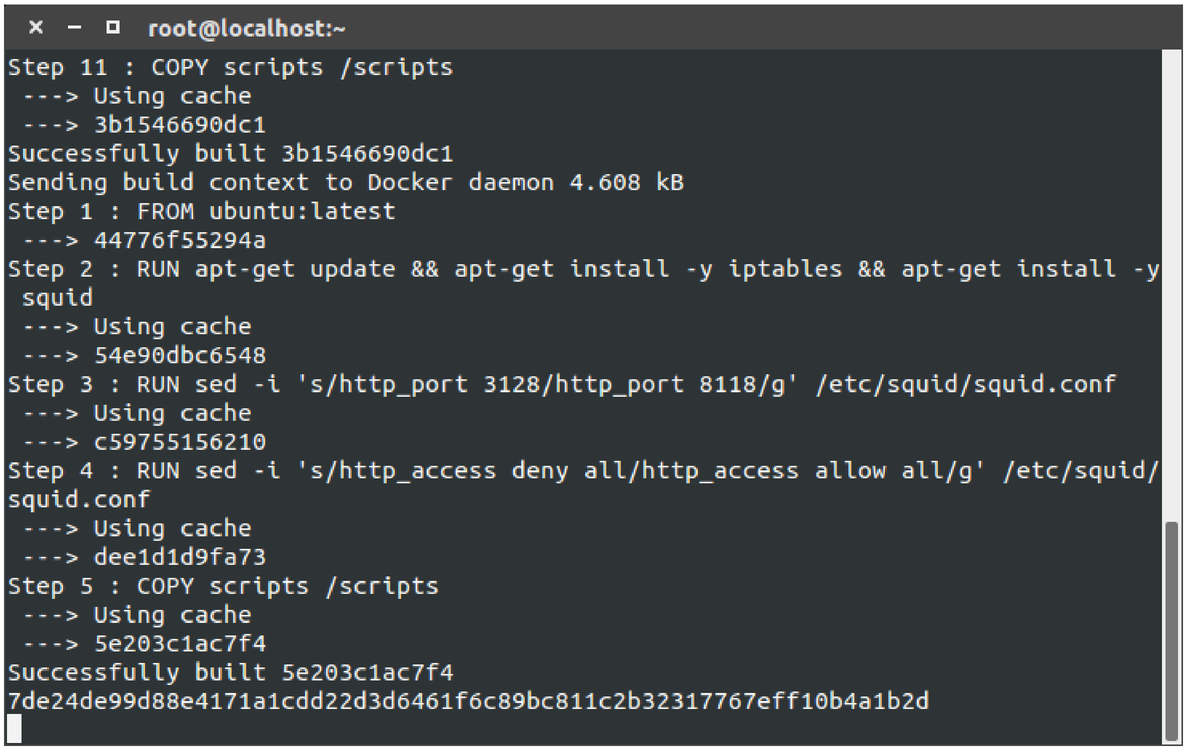
```
mike@mike-K53SC: ~/Desktop/Thesis
mike@mike-K53SC:~/Desktop/Thesis$ scp -r pt/* root@192.168.56.101:/root/arch/sft
p
root@192.168.56.101's password:
script.sh                                     100%   32     0.0KB/s   00:00
Dockerfile                                    100%  270     0.3KB/s   00:00
run.sh                                        100%  163     0.2KB/s   00:00
input.sh                                      100%  126     0.1KB/s   00:00
spider.py                                     100%  777     0.8KB/s   00:00
proxy.sh                                      100%  322     0.3KB/s   00:00
script.sh                                     100%  133     0.1KB/s   00:00
Dockerfile                                    100%  811     0.8KB/s   00:00
mike@mike-K53SC:~/Desktop/Thesis$
```

Figure A.4: Virtual Machine setup: input files delivery



```
mike@mike-K53SC:~/Desktop/Thesis$ ssh root@192.168.56.101
root@192.168.56.101's password:
Last login: Wed May 11 12:36:11 2016 from 192.168.56.1
[root@localhost ~]# touch arch/sftp/done
[root@localhost ~]# 
```

Figure A.5: Virtual Machine setup: delivery of "done" file

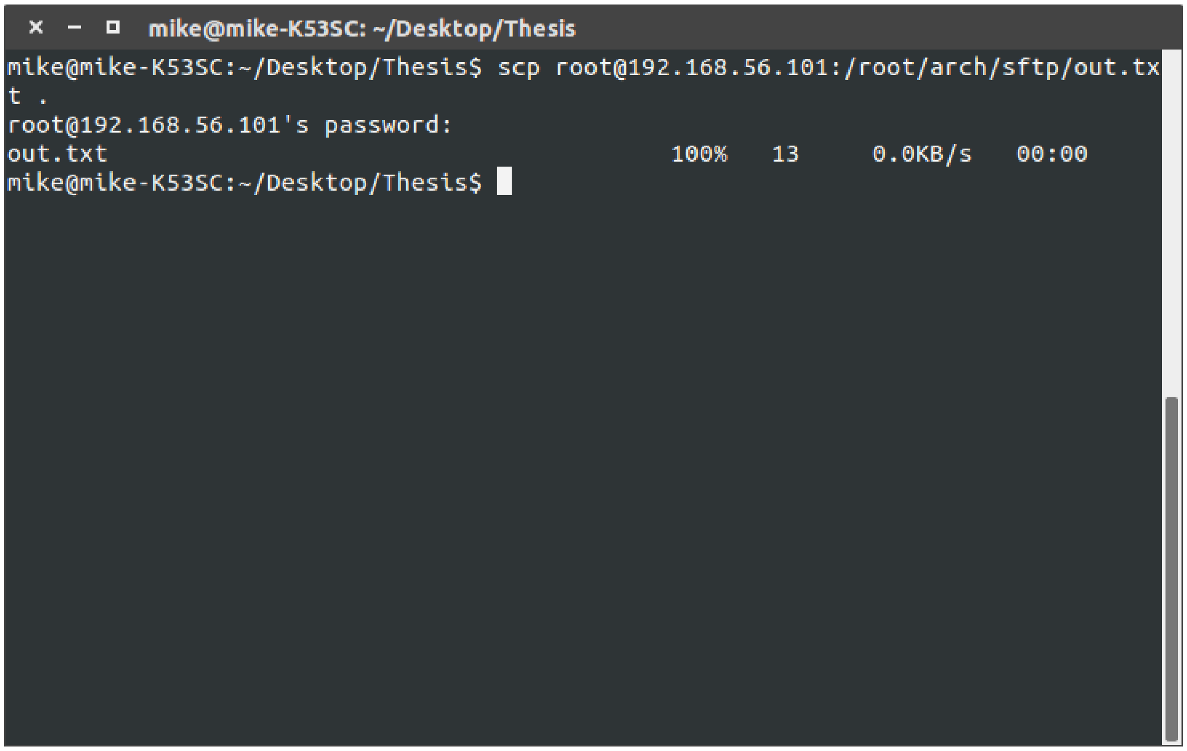


```
root@localhost:~
Step 11 : COPY scripts /scripts
 ---> Using cache
 ---> 3b1546690dc1
Successfully built 3b1546690dc1
Sending build context to Docker daemon 4.608 kB
Step 1 : FROM ubuntu:latest
 ---> 44776f55294a
Step 2 : RUN apt-get update && apt-get install -y iptables && apt-get install -y
 squid
 ---> Using cache
 ---> 54e90dbc6548
Step 3 : RUN sed -i 's/http_port 3128/http_port 8118/g' /etc/squid/squid.conf
 ---> Using cache
 ---> c59755156210
Step 4 : RUN sed -i 's/http_access deny all/http_access allow all/g' /etc/squid/
squid.conf
 ---> Using cache
 ---> dee1d1d9fa73
Step 5 : COPY scripts /scripts
 ---> Using cache
 ---> 5e203c1ac7f4
Successfully built 5e203c1ac7f4
7de24de99d88e4171a1cdd22d3d6461f6c89bc811c2b32317767eff10b4a1b2d
```

Figure A.6: Virtual Machine setup: crawling task running

A.3.1 Output Retrieval

It is now possible to retrieve the output directly from the *sftp* folder. This step is shown in figure A.7 using scp.



```
mike@mike-K53SC:~/Desktop/Thesis$ scp root@192.168.56.101:/root/arch/sftp/out.tx
t .
root@192.168.56.101's password:
out.txt                                       100%   13     0.0KB/s   00:00
mike@mike-K53SC:~/Desktop/Thesis$ 
```

Figure A.7: Virtual Machine setup: output retrieval

A.4 Accessing the VM through Tor

Assuming we want to connect to our architecture from an Ubuntu machine (the operations for a different distribution are very similar), the first step is to download *Privoxy*:

"Privoxy is a non-caching web proxy with advanced filtering capabilities for enhancing privacy, modifying web page data and HTTP headers, controlling access, and removing ads and other obnoxious Internet junk. Privoxy has a flexible configuration and can be customized to suit individual needs and tastes. It has application for both stand-alone systems and multi-user networks."[13]

So, in our terminal

```
sudo apt-get update && sudo apt-get install privoxy
```

Now, we start it (init.d):

```
sudo /etc/init.d/privoxy start
```

Or (systemd):

```
sudo service privoxy start
```

By default, Privoxy listens on port 8118, therefore we launch ssh with the following options:

```
ssh -L 8118:localhost:8118 <username>@<ip_address>
```

A.5 Architecture: Organization

The architecture has the following structure:

```
Arch
├── sftp
│   ├── logic
│   │   ├── Dockerfile
│   │   └── scripts
│   ├── gateway
│   │   ├── Dockerfile
│   │   └── scripts
│   └── input
├── controller
│   ├── controller.sh
│   └── executor.sh
└── shared
```

- *sftp*: folder accessible from a remote position, used to pass the containers and the tasks to be executed by the crawler. This folder content is removed as soon as the crawling task is launched.
  - *logic*: contains the dockerfile and the scripts needed to set the logic container up.
  - *gateway*: contains the dockerfile and the scripts needed to set the gateway container up.
  - *input*: contains the task logic to be launched by the crawler.

- *controller*: folder containing the scripts needed to transparently launch the crawling task as soon as they arrive.
  - *controller.sh*: the daemon which continuously monitors the *sftp* folder waiting for a task to arrive.
  - *executor.sh*: launched by *controller.sh*, this script is in charge to launch the crawling task and produce the output.
- *shared*: shared folder between the host machine and the containers. Reserved to the system.

Appendix B

Architecture: Scripts

Controller code (to be launched with root privileges):

```
#!/bin/bash

# Check if all files are in "shared" folder
# and launch executor.sh
sftp_dir=/home/mike/docker/sftp
done_file=/home/mike/docker/sftp/done
logic_folder=/home/mike/docker/sftp/logic
gateway_folder=/home/mike/docker/sftp/gateway
input_folder=/home/mike/docker/sftp/input
executor_sh=/home/mike/docker/controller/executor.sh
while true
do
```

```
  for entry in "$sftp_dir"/*
  do
    # As soon as the confirmation file arrives,
    # launch executor.sh, then delete input files
    # in sftp folder
    if [ "$entry" = "$done_file" ]; then

      sh "$executor_sh"

      # Delete input files
      rm -rf "$logic_folder"
      rm -rf "$gateway_folder"
      rm -rf "$input_folder"
      rm "$done_file"
    fi
  done
done
```

Executor code:

```
#!/bin/bash

logic_folder=/home/mike/docker/shared/logic
gateway_folder=/home/mike/docker/shared/gateway
input_folder=/home/mike/docker/shared/input

```

```
# Get Input
cp -a /home/mike/docker/sftp/logic
  /home/mike/docker/shared
cp -a /home/mike/docker/sftp/gateway
  /home/mike/docker/shared
cp -a /home/mike/docker/sftp/input
  /home/mike/docker/shared

# Run Dockerfiles
docker build -t arch/logic
  /home/mike/docker/shared/logic/
docker build -t arch/gateway
  /home/mike/docker/shared/gateway/

# Launch Gateway (in background)
docker run -itd --privileged --name gateway
arch/gateway
bash -c "sudo iptables -t nat -A POSTROUTING -o eth0
-j MASQUERADE; bash"

# Launch Logic Container
docker run --privileged --rm --link gateway
--name logic
-v /home/mike/docker/shared/input:/shared arch/logic
```

```
bash -c "route del default; route add default gw
gateway eth0; sudo sh scripts/script.sh"

# Kill Gateway
docker stop gateway
docker rm gateway

# Remove images
docker rmi arch/gateway
docker rmi arch/logic

# Produce output
mv /home/mike/docker/shared/input/*
   /home/mike/docker/sftp

# Remove input files
rm -rf "$logic_folder"
rm -rf "$gateway_folder"
rm -rf "$input_folder"
```

Logic container Dockerfile:

```
FROM ubuntu:latest
RUN apt-get -y update && apt-get install -y ping
COPY scripts /scripts
```

Gateway container Dockerfile:

```
FROM ubuntu:latest
RUN apt-get -y update && apt-get install -y iptables
COPY scripts /scripts
```

B.1 Network Tests: Scripts

```
#!/bin/bash
# Run with sudo!

array=( http://speedtest.ftp.otenet.gr/files/
test100k.db
http://speedtest.ftp.otenet.gr/files/test1Mb.db
http://speedtest.ftp.otenet.gr/files/test10Mb.db )

# Testing Open Targets without Tor
echo Testing Without Tor:
for var in 0 1 2 3 4 5 6 7 8 9
do
  echo Iteration $var

  echo Packet Loss: speedtest.ftp.otenet.gr
  ping -q -n -c 10 speedtest.ftp.otenet.gr |
  grep "packet loss" | cut -d " " -f 6

  for i in "${array[@]}"
  do
    echo Bandwidth: $i
        wget -O /dev/null $i 2>&1 | grep -o "[0-9.]\+
        [KM]*B/s"

        echo Latency: $i
        time wget -q -O /dev/null $i 2>&1 | grep
        elapsed
        echo \n\n
  done

  echo Sleeping 6 minutes
  sleep 6m
done

# Testing Open Targets through Tor
echo Testing With Tor
for var in 0 1 2 3 4 5 6 7 8 9
do
echo Iteration $var

  echo Packet Loss: speedtest.ftp.otenet.gr
  (proxychains ping -q -n -c 10 speedtest.ftp.
  otenet.gr)
  | grep "packet loss" | cut -d " " -f 6

  for i in "${array[@]}"
  do
        echo Bandwidth: $i
        (proxychains wget -O /dev/null $i) 2>&1
        | grep -o "[0-9.]\+ [KM]*B/s"

        echo Latency: $i
        (proxychains time wget -q -O /dev/null $i)
        2>&1 | grep elapsed
        echo \n\n
  done

  echo Sleeping 6 minutes
  sleep 6m
done
```

Appendix C

Network Tests: Diagrams

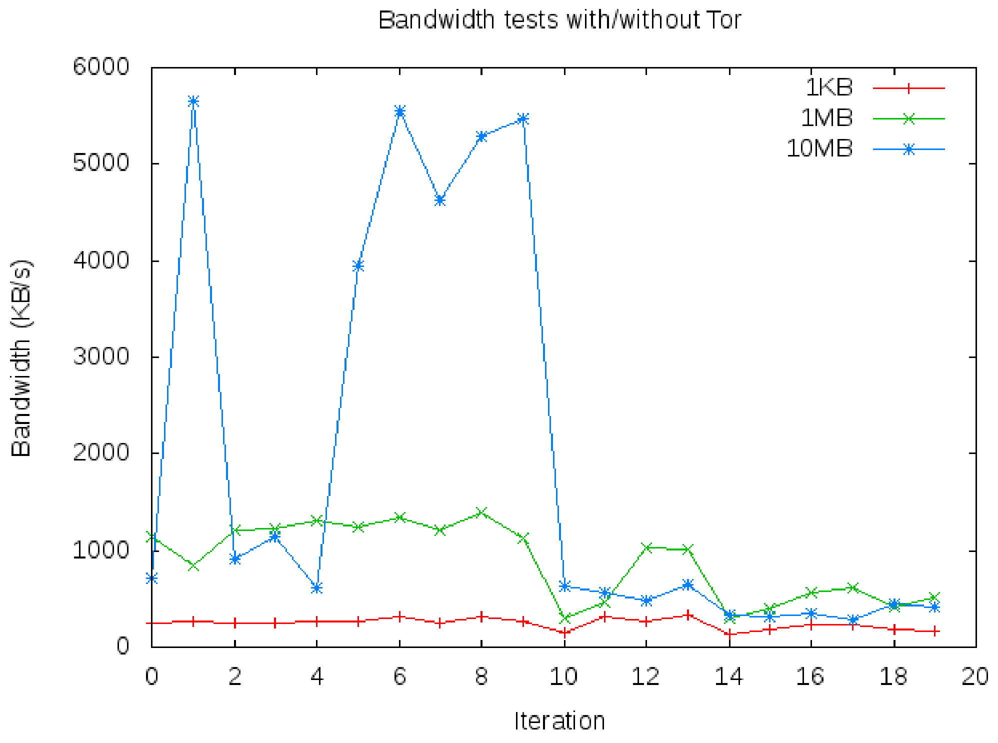


Figure C.1: Bandwidth Test - Day 1 - 10 a.m.

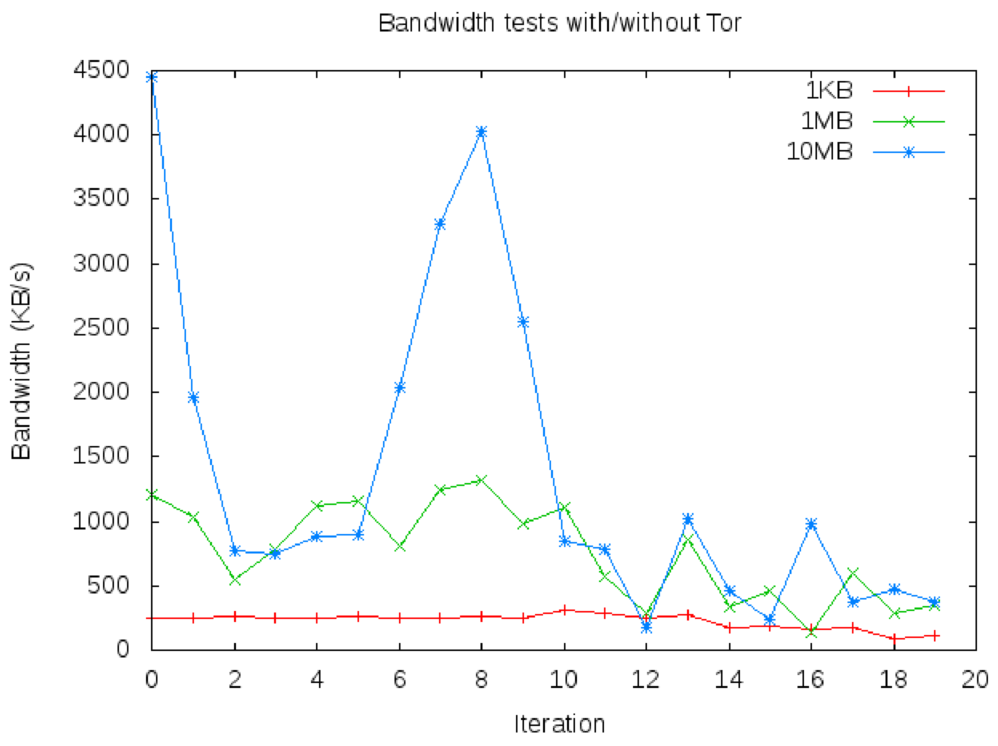


Figure C.2: Bandwidth Test - Day 1 - 12 a.m.

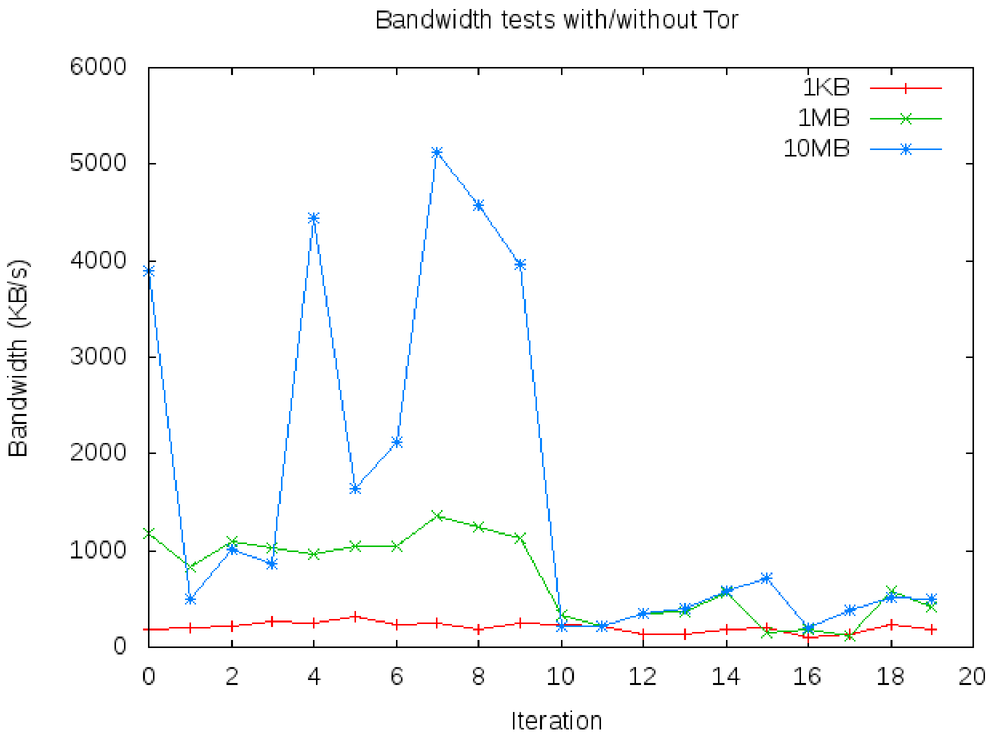


Figure C.3: Bandwidth Test - Day 1 - 14 p.m.

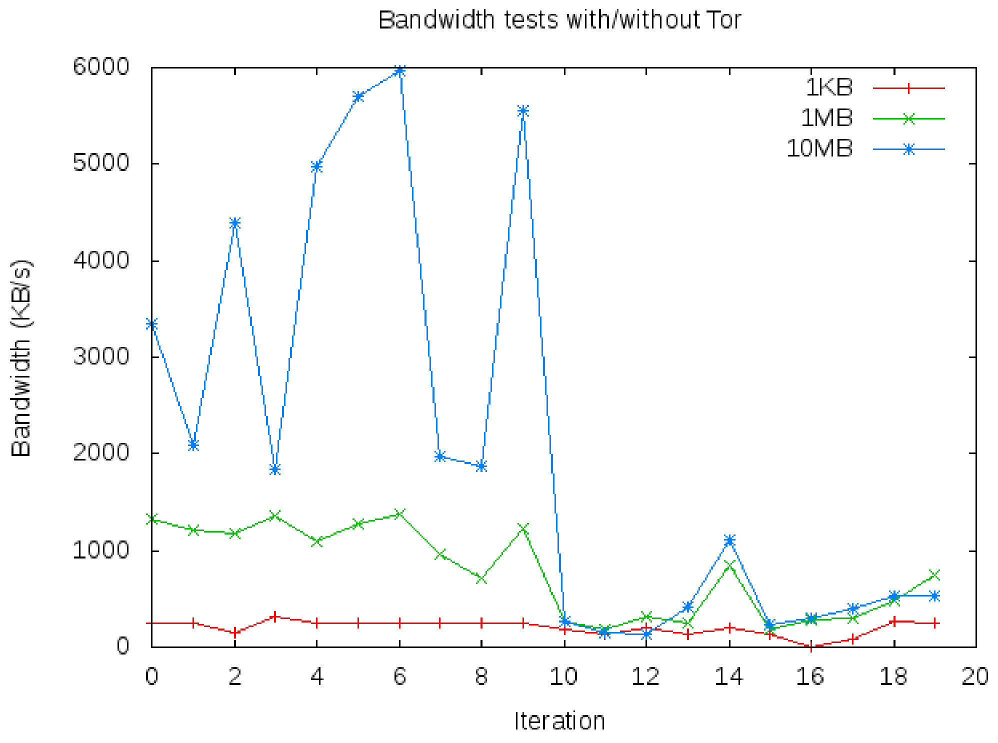


Figure C.4: Bandwidth Test - Day 1 - 16 p.m.

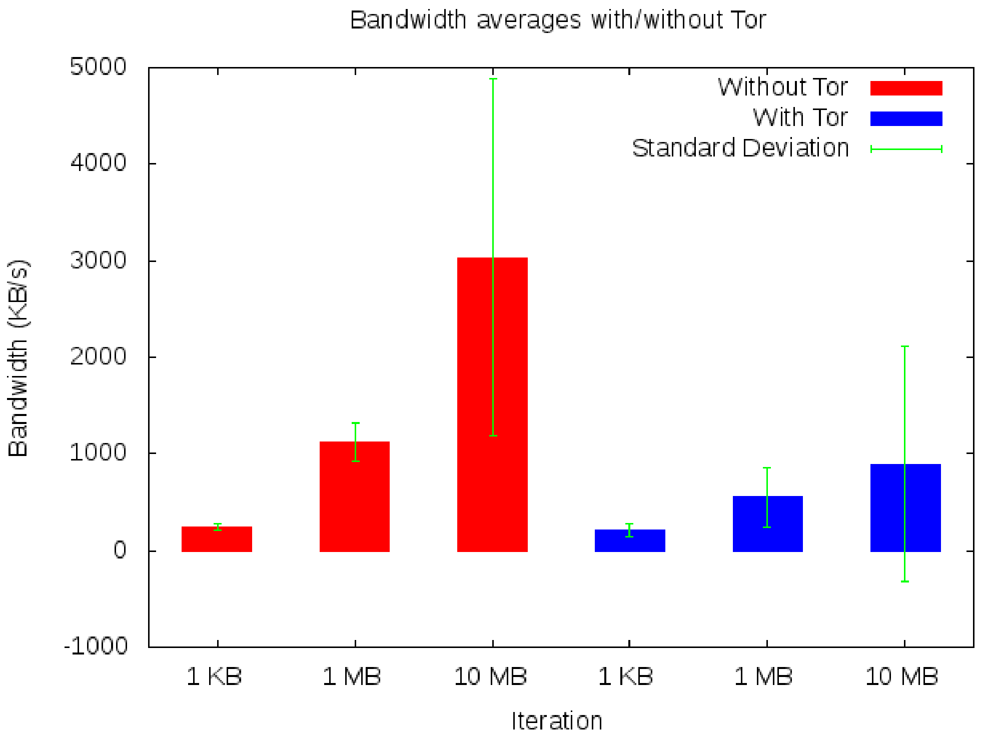


Figure C.5: Bandwidth Test - Day 1 - Average and Standard Deviation

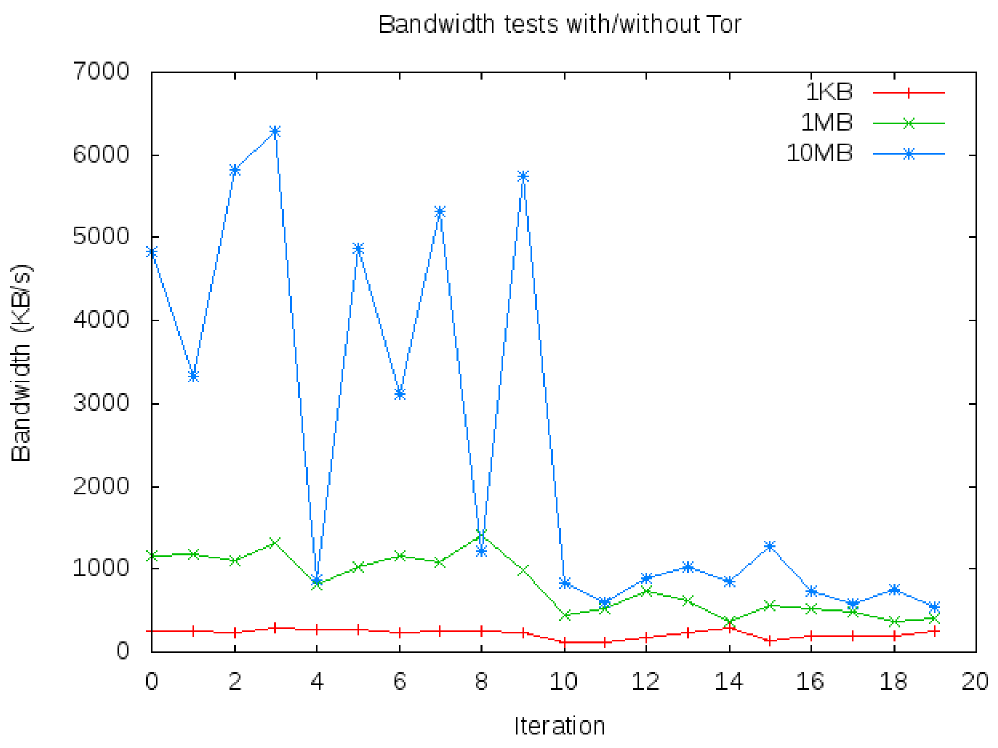


Figure C.6: Bandwidth Test - Day 2 - 10 a.m.

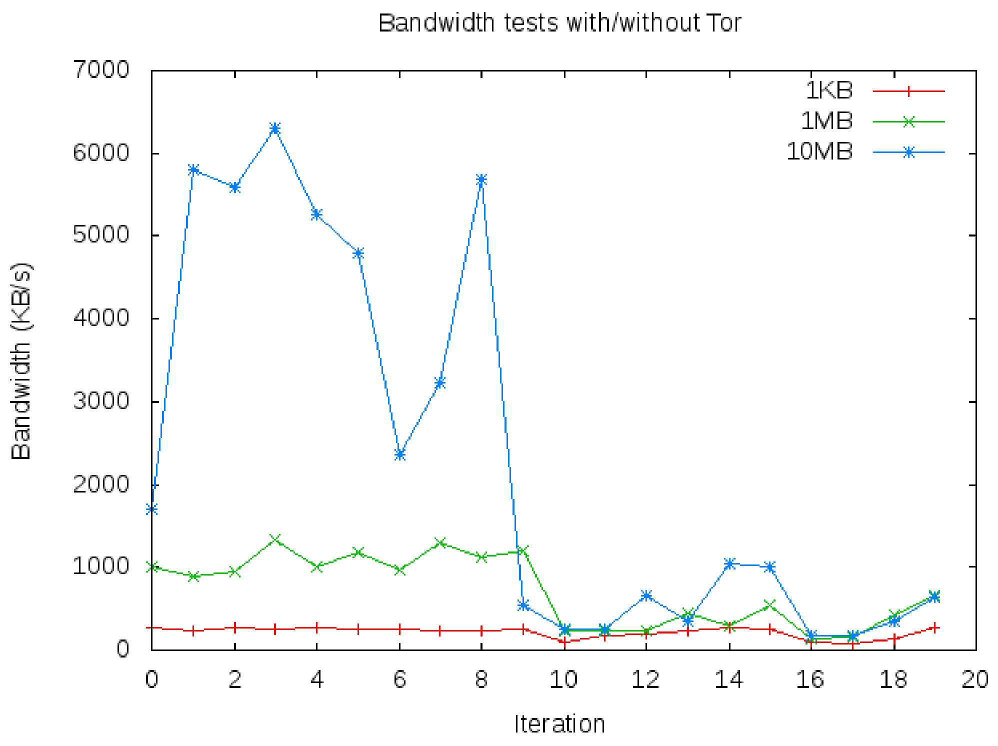


Figure C.7: Bandwidth Test - Day 2 - 12 a.m.

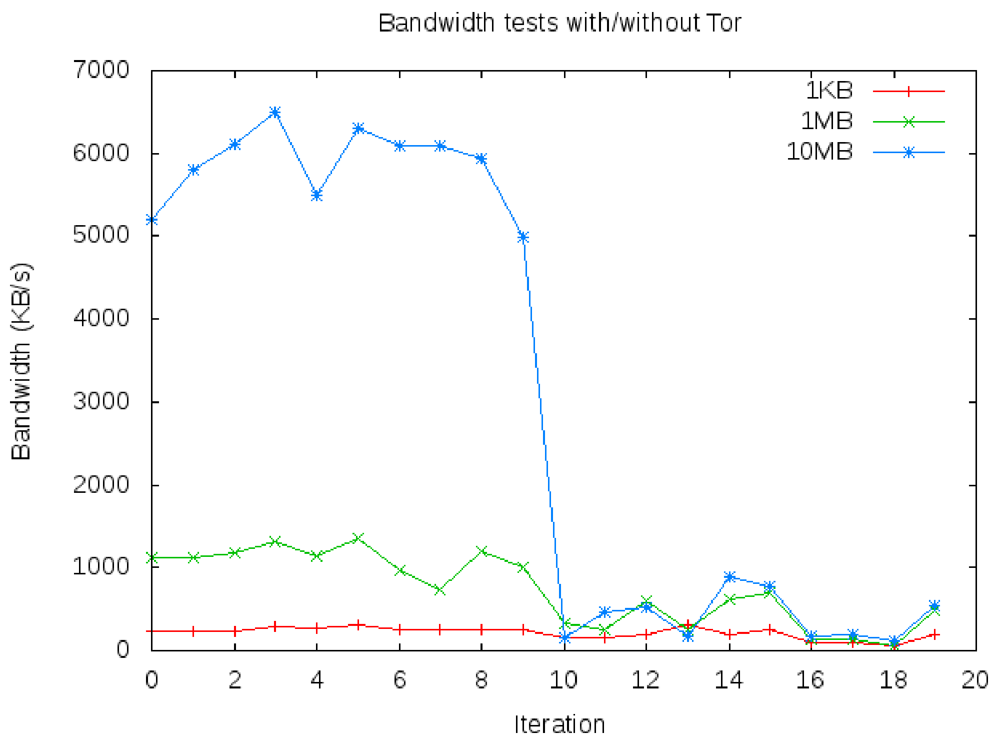


Figure C.8: Bandwidth Test - Day 2 - 14 p.m.

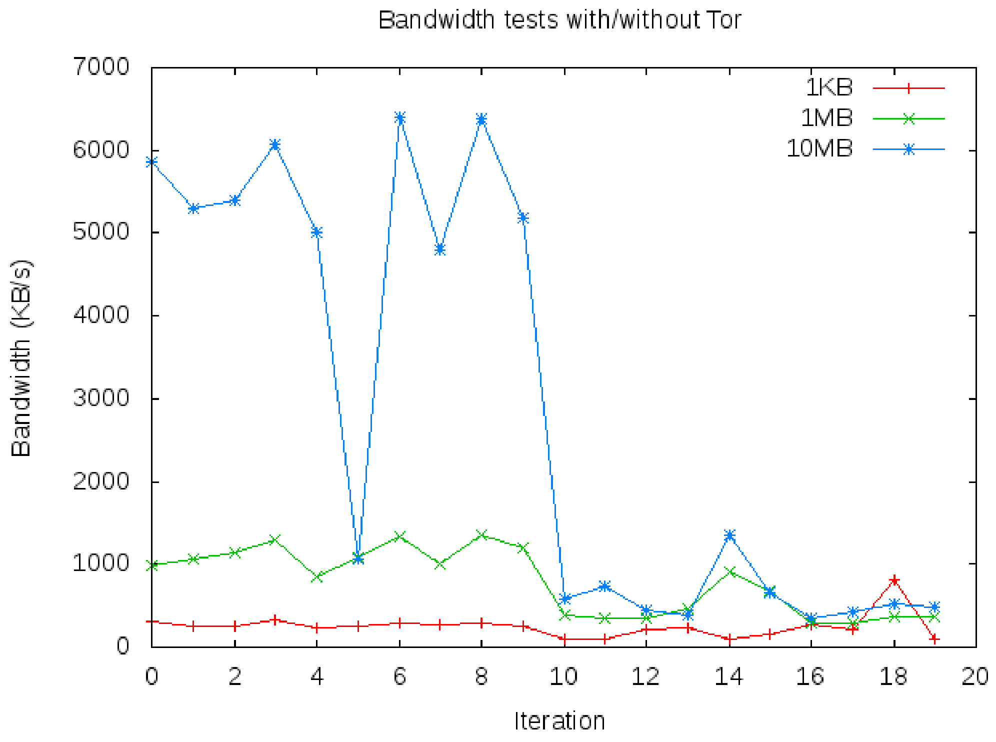


Figure C.9: Bandwidth Test - Day 2 - 16 p.m.

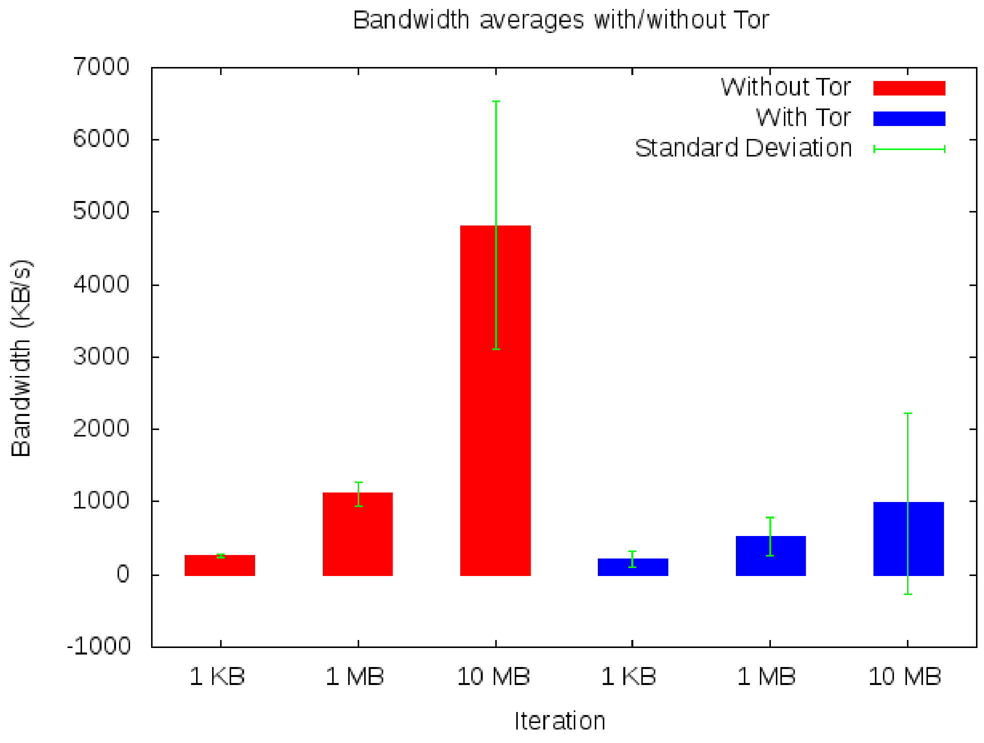


Figure C.10: Bandwidth Test - Day 2 - Average and Standard Deviation

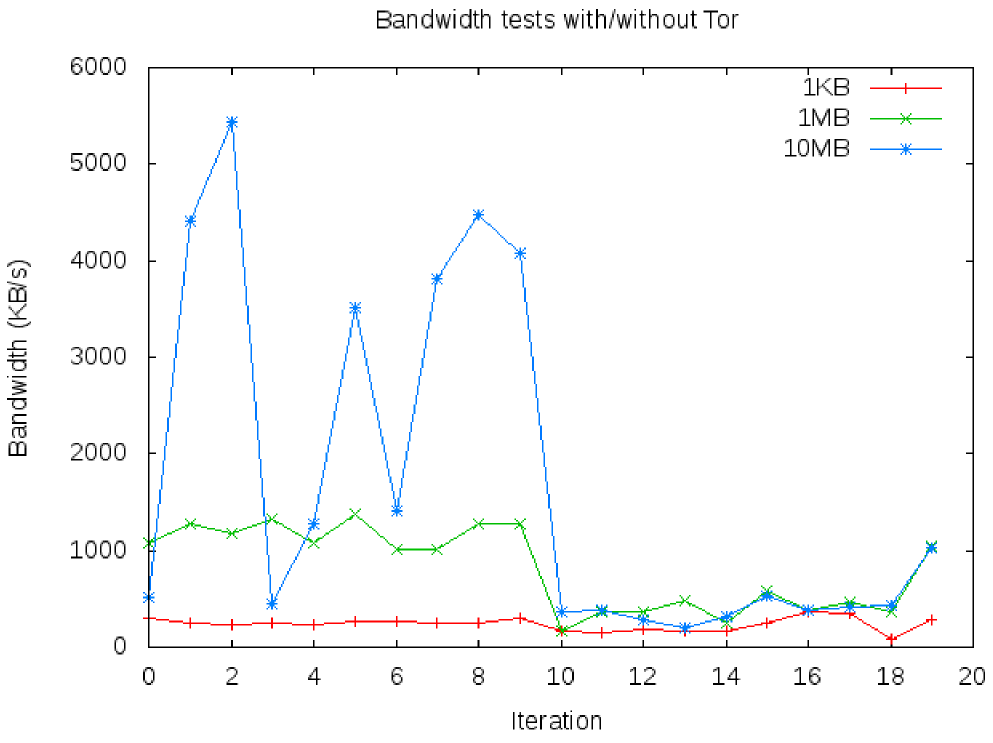


Figure C.11: Bandwidth Test - Day 3 - 10 a.m.

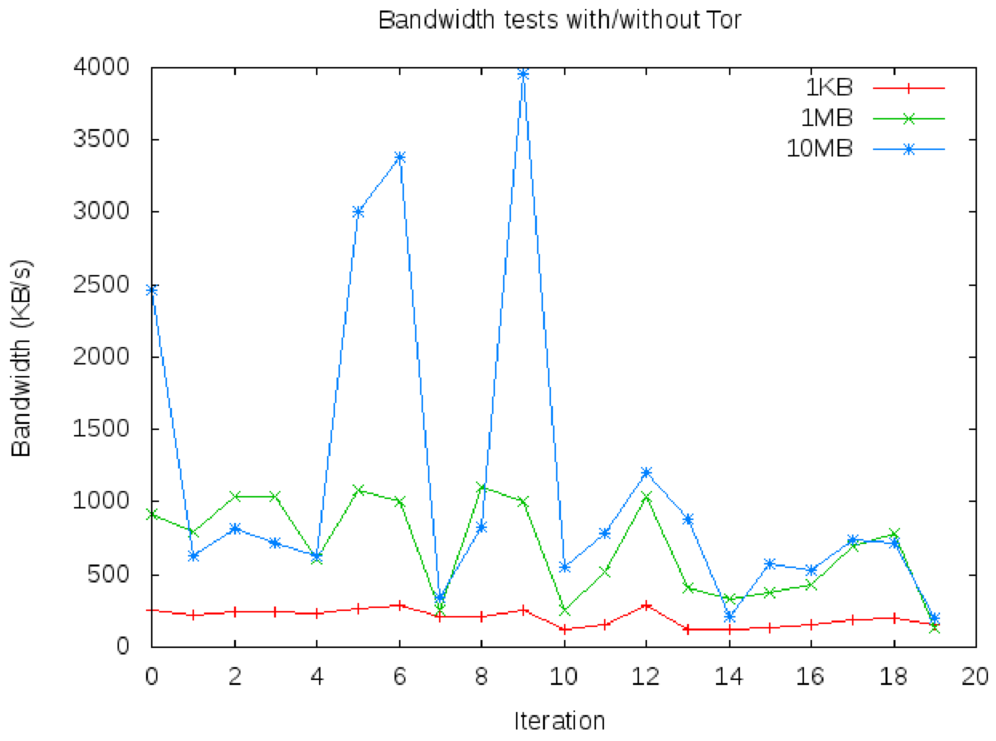


Figure C.12: Bandwidth Test - Day 3 - 12 a.m.

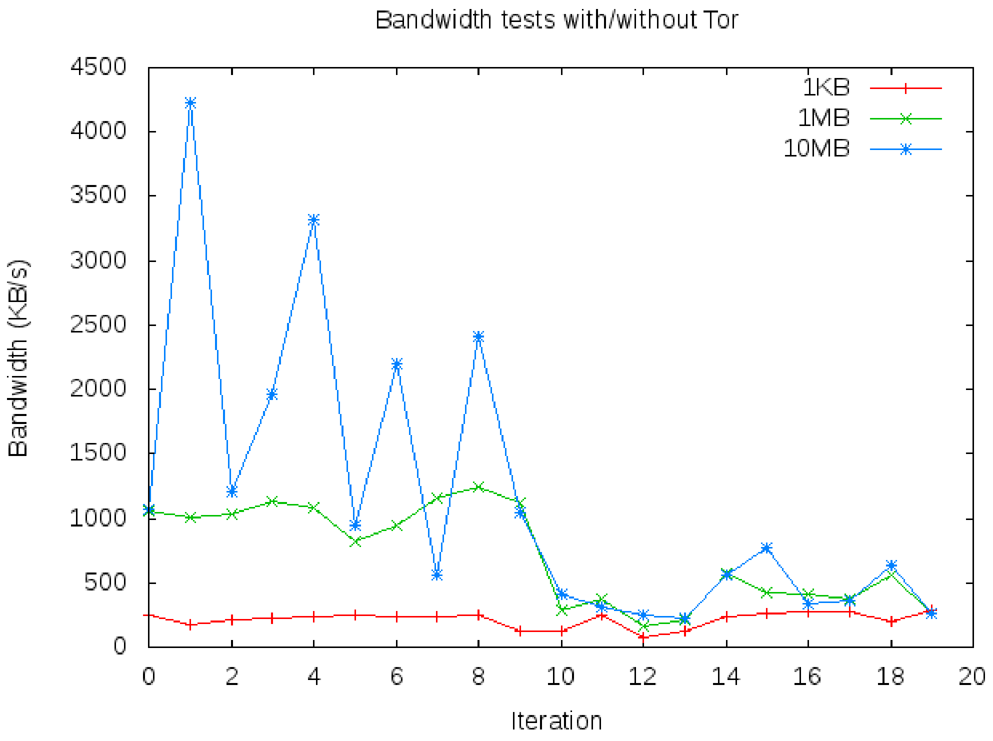


Figure C.13: Bandwidth Test - Day 3 - 14 p.m.

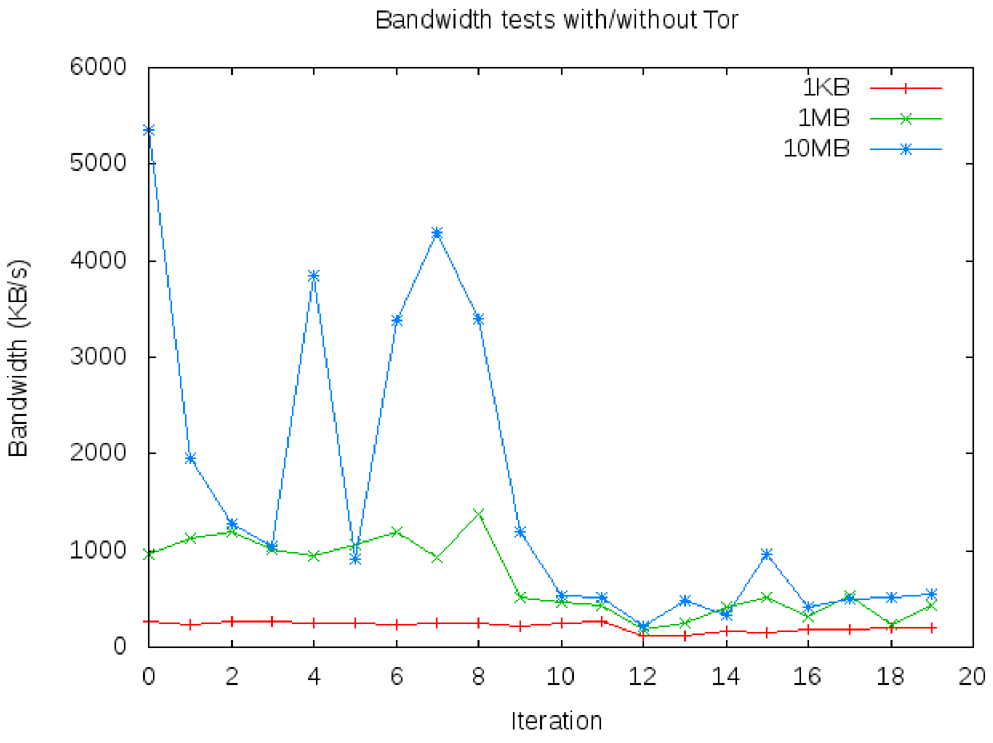


Figure C.14: Bandwidth Test - Day 3 - 16 p.m.

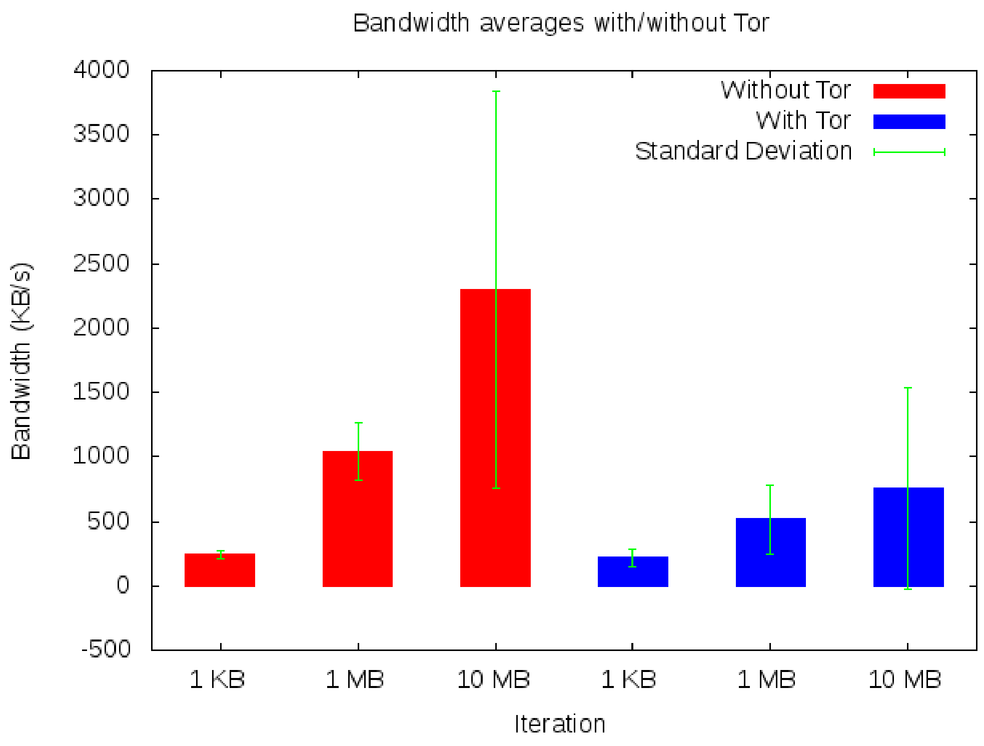


Figure C.15: Bandwidth Test - Day 3 - Average and Standard Deviation

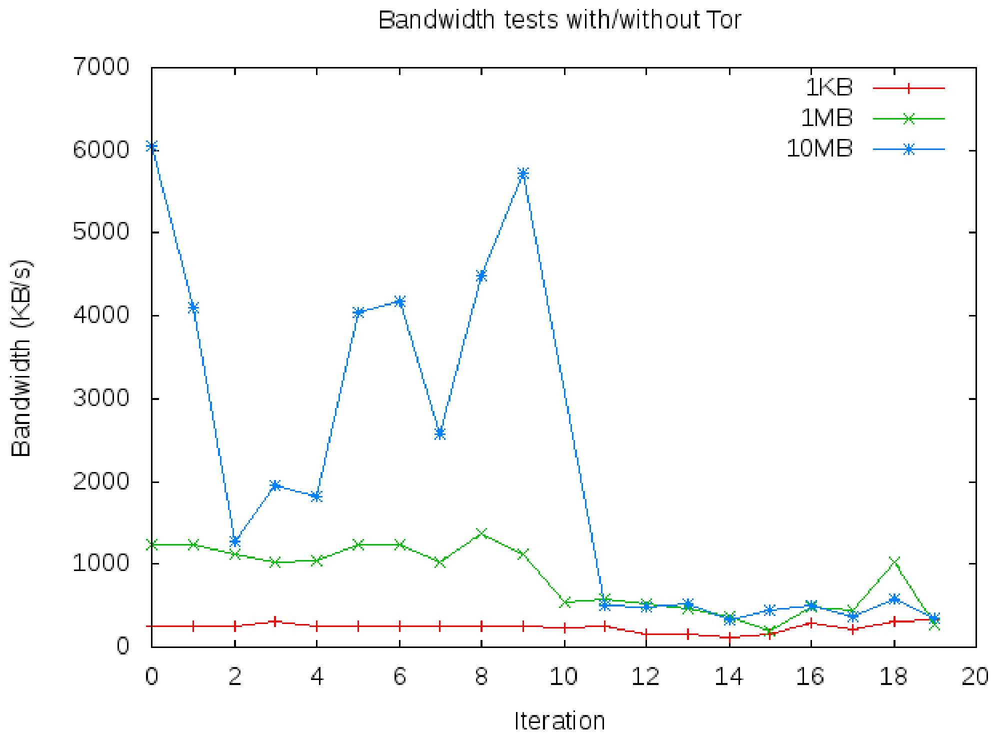


Figure C.16: Bandwidth Test - Day 4 - 10 a.m.

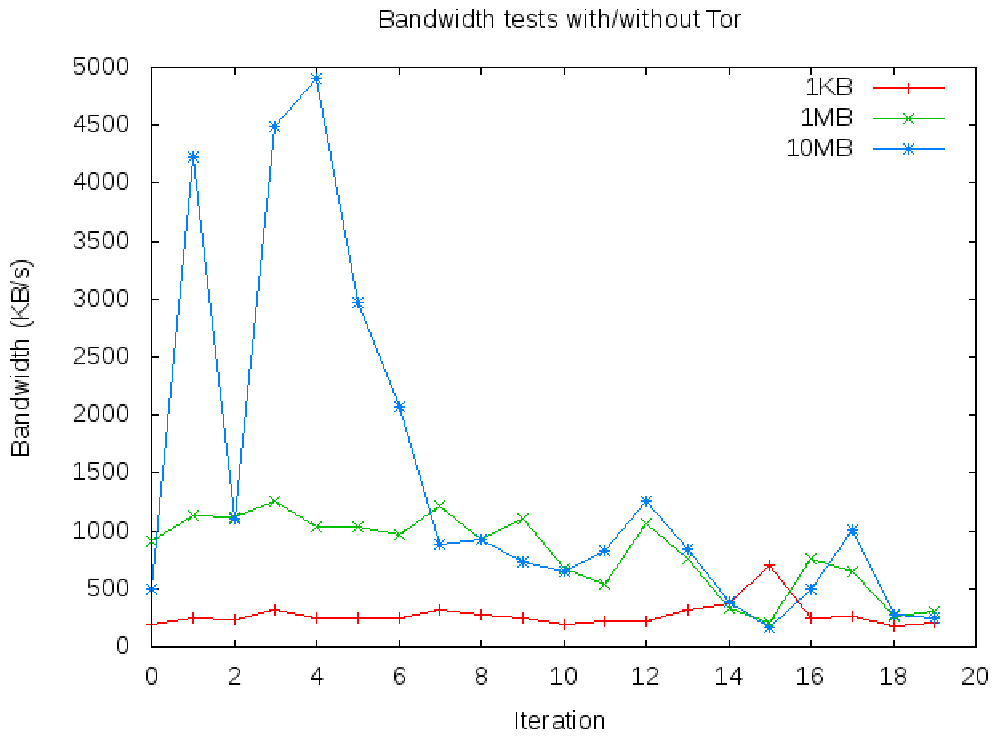


Figure C.17: Bandwidth Test - Day 4 - 12 a.m.

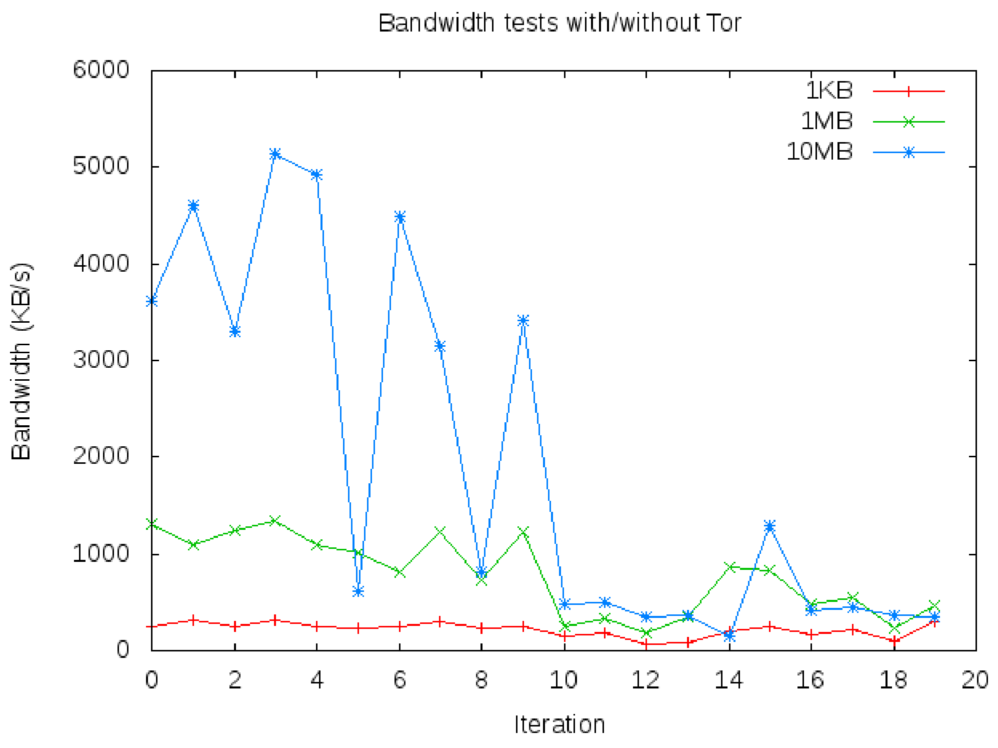


Figure C.18: Bandwidth Test - Day 4 - 14 p.m.

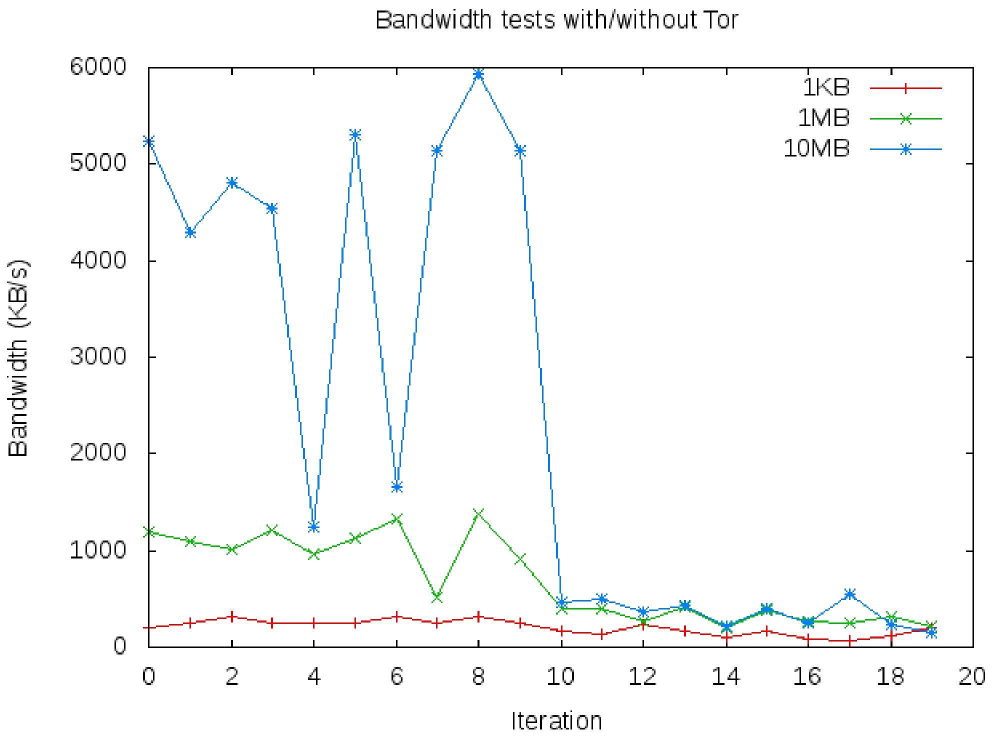


Figure C.19: Bandwidth Test - Day 4 - 16 p.m.

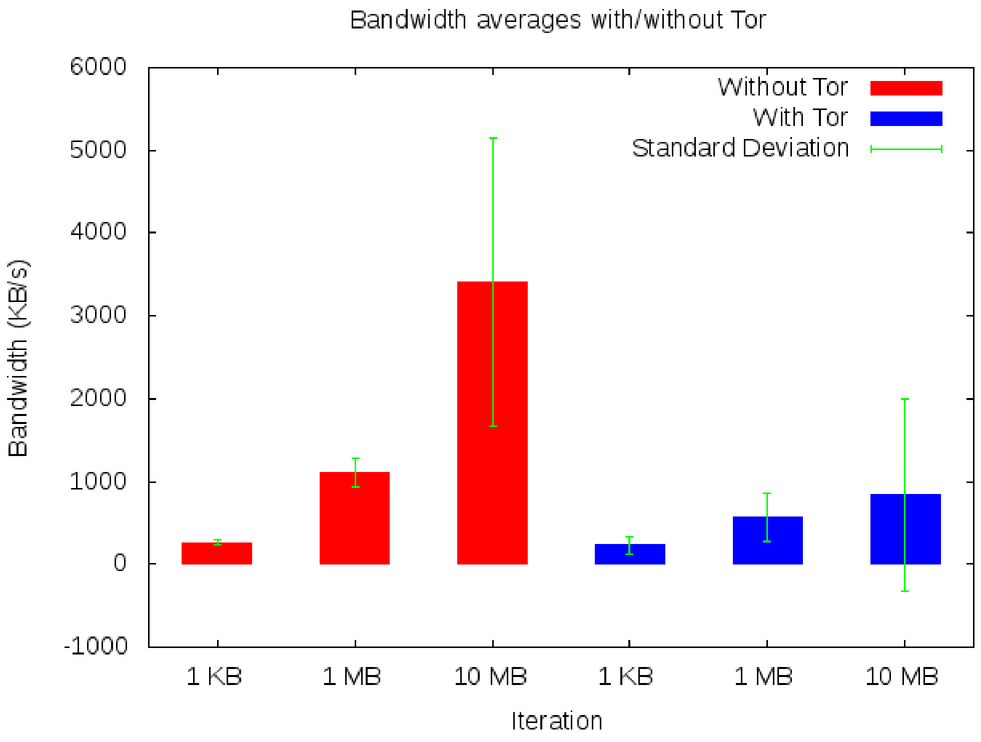


Figure C.20: Bandwidth Test - Day 4 - Average and Standard Deviation

Bibliography

[1] Wikipedia, *Anonymous P2P*, *https://en.wikipedia.org/wiki/Anonymous_P2P*

[2] Kevin Bauer, Joshua Juen, Nikita Borisov, Dirk Grunwald, Douglas Sicker, and Damon McCoy, *On the Optimal Path Length for Tor*

[3] Roger Dingledine, Nick Mathewson, Paul Syverson, *Tor: The Second-Generation Onion Router*

[4] Roger Dingledine, *Tor security advisory: "relay early" traffic confirmation attack*, *https://blog.torproject.org/blog/tor-security-advisory-relay-early-traffic-confirmation-attack*

[5] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, Srinivas Devadas, *Circuit Fingerprinting Attacks: Passive Deanonymization of Tor Hidden Services*

[6] Roger Dingledine, Steven J. Murdoch, *Performance Improvements on Tor or, Why Tor is slow and what we're going to do about it*

[7] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, Paul Syver-

son, *Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries*

[8] Nathan S. Evans, Roger Dingledine, Christian Grothoff, *A Practical Congestion Attack on Tor Using Long Paths*

[9] *Squid FAQ: About Squid*, *http://wiki.squid-cache.org/SquidFaq/AboutSquid*

[10] *Scrapy Official Documentation*, *http://doc.scrapy.org/en/latest/intro/overview.htm*

[11] *Whonix Official Website*, *https://www.whonix.org/*

[12] *Tails Official Website*, *https://tails.boum.org/*

[13] *Privoxy Official Website*, *https://www.privoxy.org/*