A scalable crawler on the TorNetwork

Antoine Masanet
Advisor: Tiziano Piccardi
Data Science Lab, École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
antoine.masanet@epfl.ch

ABSTRACT

The Tor network, despite being used by over 2 million montlhy users [1], remains to this day largely unknown. In this bachelor project, I built a web crawler on the Tor Network that recursively gathers pages from onion services. After a 20 days crawl, more than 500 000 onion services were discoverd and over 6 million pages were fetched for a total of 85 GB of compressed data. Using a sample of this data, I built a graph representing the topology of this network and extracted some key characteristics from it.

KEYWORDS

web crawler, Tor network, dataset, network topology

ACM Reference Format:
Antoine Masanet, Advisor: Tiziano Piccardi. 2021. A scalable crawler on the TorNetwork. In *Proceedings of Dlab '21: Dlab Presentation (Dlab '21).* ACM, New York, NY, USA, 5 pages. https://doi.org/None

1 INTRODUCTION

The Tor Network is a free and open-source software that enables its users to perform anonymous queries on the internet. This technology is based on a volunteer overlay network with nodes successively adding a new layer of encryption to the query. The virtual circuit created ensures that no one on the network can identify both the source of the query as well as the query itself. In a similar fashion, web services on the Tor network, called onion services, can also hide their locations making it almost impossible for government agencies to track them. As a consequence of this provided anonymity, a lot of sensitive content and websites can be found on these hidden services such as terrorism networks, drug dealing, security breaches... This lead to many studies analysing the popularity of websites [2] or assessing the security of the network [3] but very few analyzed the topology of the network. A 2017 study, claiming to be the first to do such a thing, modeled a sample of the Tor network, fetching 5000 domains and a million pages [1]. This uneven crawl suggested that it was hard to discover new domains or that the crawler used was not Depth First Search oriented as it fetched an average of 200 pages per domain. The goal of this project is therefore to build a balanced DFS web crawler in order to gather as much data as possible on the Tor Network. The large sample obtained could later be used for a variety of study such as to build and study a wider graph of the network as well as using the pages content to classify the communities obtained.

[1]https://metrics.torproject.org/userstats-relay-country.html

2 CHALLENGES

They are multiple characteristics a good crawler should have and I tried to address all of these challenges in this project.

- Fast and scalable
- Crash resistant
- Prevent sites from blocking it
- Store pages in an efficient and easily accessible manner
- Allow access to sites requiring login
- Revisit websites that are only available at specific times of the day
- Real time crawl monitoring
- Balanced crawl

3 STORM CRAWLER

The first crawler built to achieve this task was based upon the *StormCrawler* SDK[2]. This collection of resources enables the creation of a distributed web crawler using *Apache Storm*. The advantage of this approach is that it offers many different self contained components that can communicate with each others using *Apache Storm* tuples. This makes it possible for different components to be running on different machines. Furthermore, once a topology is launched, failing node are automatically restarted, making it crash resilient. Finally, its continuous stream processing makes it 60% faster than a similar batch processing *Apache Nutch* alternative [3]. On top of the *Apache Storm* topology is an *Elasticsearch* index that stores the content of the fetched pages and the urls in the queues. Each crawling thread then pulls URLs from the index, ensuring that all URLs from the same host goes to the same fetcher. Using those tools, encapsulating it in a docker cluster and adding a crawl visualization tool (*Kibana*) enabled the creation of a fast, crash resilient and easily deployable crawler on the Tor Network. Despite those advantages, the crawler had significant drawbacks such as using a DFS schema and not allowing customization such as adding login necessary to access certain websites. As a consequence, we decided to build a new custom implementation for crawling the Tor Network. Nevertheless, this crawler could very easily be reused for any project that would like to crawl a single website[4].

[2]http://stormcrawler.net
[3]https://dzone.com/articles/the-battle-of-the-crawlers-apache-nutch-vs-stormcr
[4]https://github.com/epfl-dlab/TorCrawler

4 CUSTOM CRAWLER

After 9 weeks of battling with the StormCrawler SDK with little progress, we decided that it would be simpler to build our crawler in Java from scratch in order to have it customized and optimized for the task at hand [5].

4.1 Fast and scalable

To solve this challenge, the whole crawling process can have a customizable number of threads with each thread sharing concurrent data structures and writing the fetched pages to independent files. The version running on the cluster runs on 50 threads. This is essential as queries on the Tor Network tend to have a very long latency (each thread fetches on average 1 page every 10s), greatly reducing the crawlers throughput.

4.2 Crash resistant

We wanted the crawler to be able to recover in case of an unexpected crash of the server it is running on. In order to do this, the queues cannot only be stored on RAM but must be persisted on disk in real-time. Two choices where possible, either using an external database system such as ElasticSearch or Berkley DB or directly storing the queue in a file. I chose the latter option using square/tape QueueFile [6] that offered a fast, transactional, file-based FIFO queue. This transactional queueu ensured that the file containing the queue would remain consistent in case of a crash. Finally, this queue does not natively support multi-threading so I had to wrap it in a concurrent blocking queue.

4.3 Prevent sites from detecting and blocking the crawler

An issue that was encountered in the first versions of the crawler on the clear web was *429 Too Many Requests* or *403 Forbidden HTTP* error produced by some websites that most likely detected the crawler and blocked access to my IP. To moitigate this, I used three different techniques to make the crawler as discrete as possible. The first one is using a Tor proxy that rebuilds the virtual circuit every 2 min making it appear like the query comes from another exit relay[7]. The second one is adding in the *Jsoup* query the same user agent and headers the *Tor Browser* sends to fetch a webpage. Finally, the last one consists in having a long interval between queries to the same domain implemented by the round-robin queue that will be seen later.

4.4 Efficient and easily accessible page storing

To minimize the space the pages take, all the data is written in *gzip* compressed format achieving a 10x compression rate. Furthermore, there is a limit of 65 536 characters for the content of pages. Finally, the data is written in a *JSON* format that uses Google's *Gson* library for serialization and deserialization.

[5]https://github.com/epfl-dlab/CustomTorCrawler
[6]https://github.com/square/tape
[7]https://github.com/zet4/alpine-tor

4.5 Allow access to usually blocked websites

Some sites on the Tor Network, such as forums, require a login to be accessed, preventing a simple crawler from accessing them. To overcome this issue, the crawler provides a file dedicated to cookies that can be used to manually enter cookies for certain websites. Those cookies coming from manual login will then be used by the crawler when it accesses those websites.

4.6 Revisit inaccessible websites

In the clear web, most websites have an uptime of 99,9% and it is very rare that a crawler would visit a website when it is temporally down. On the other hand, some onion services are only available at certain periods of the day making them likely to be inaccessible when the crawler tries to access them. A solution to this issue could be to push failed URLs at the end of the queue. However, most of the failed page queries correspond to URLs that lead to hosts whose service is no longer running. As a result those stale URLs would continuously be pushed back in the queue greatly reducing the crawler's efficiency. The approach I took was to store all the URLs that lead to a failed website in a dedicated compressed file to be able to start a new crawl later using these URLs as a seed.

4.7 Monitoring

A key feature to a good crawler is being able to easily monitor the crawler's progress without needing to have the background process session open. For this matter, I created a singleton global class that is concurrently updated by all the fetching threads and contains the crawl's current status. A separate thread is then responsible for persisting this data to a file every 10 second allowing us to monitor the crawl's progress in real time.This file then enables us to see if they are any imbalance in the crawl's algorithm (a thread is fetching more/less pages than the others) or if a thread was interrupted by a unforeseen error. Finally, in case of a crash, this specific file can be reused by the crawler to recover some key informations such as the number of threads it was previously using.

Here are the different statistics provided by the status file:

- thread count
- queue size
- total number of pages correctly fetched
- total number of URLs leading to errors
- number of domains discovered
- number of URLs discovered
- percentage of non empty subqueues (to be explained later)
- crawl time
- for each thread, the number of pages it fetched

4.8 CrashRecovery

Launching the executable with the argument "recover" is all that is required to resume a stopped crawl, provided that the required file for the recovery (queues and status file) exist. This can be achieved by reloading the status file and all the individual queue file.

4.9 RoundRobinQueue

The main challenge in this crawler was finding the best suited data structure for the queue of URLs to fetch pages from. Here are the main criterias it needs to satisfy: can be concurrently accessed, achieves a balanced BFS crawl, be recoverable and be fast. The first implementation for this queue was a concurrent blocking queue that stopped adding a domain's URLs to the queue as soon as the number of pages fetched from this domain reached a certain threshold. This implementation satisfied 3 of the previous points and prevented the crawler from over-crawling a specific website. However, this data structure had three major issues. The first issue is that for a threshold of size S, the crawler could fetch all the S pages in a row resulting in a high traffic for a short period of time towards a specific website. The second issue is that the crawler would stop by itself after the threshold is reached for all discovered domains. Finally, at any point in time they would be no balance between the number of pages fetched per domains except that they are all between 0 and S. In order to solves those issues, I built a round robin blocking queue that ensures a fair crawling of all discovered hosts. This queue implements the standard interface of Java's *BlockingQueue* so any thread can easily take and put URLs into the queue without having to worry of its underlying implementation. Here is the lifecycle of a URL in this queue:

When a URL is put into the queue, it is assigned to a subqueue corresponding to its domain. Therefore, they are as many subqueues as domain discovered.

When a URL is to be taken from the queue, the data structure verifies if they are any URLs left in the *CurrentRound* queue. If they are, it just takes a URL from this queue; otherwise, it refills the *CurrentRound* queue by taking one URL from each subqueue (domain).

This algorithm ensures that for each round, exactly one page is fetched from each discovered domain which ensures an even domain-wise BFS crawl. Furthermore, as the crawl's boundary gets bigger, queries to a specific domain are spaced out by a very large time interval ensuring that the crawler has a minimal impact on onion services and does not get blocked. Furthermore, the subqueues that are all using the persistent *ObjectQueue* class in order to be easily recoverable.

It is to be noted that the *RoundRobinQueue* class could be reused for other projects to store any element of type <T> as long as the user provides a function that maps this <T> to a <String> that will be used to determine in which subqueue the element will be stored.

Advantages:

- works concurrently
- even domain-wise distribution of URLs
- fast: inspired by Java's own implementation of *LinkedBlockingQueue*[8]
- recoverable

Drawbacks:

- lots of files are created (one file per domain discovered)

At first, I thought that after a certain amount of time, most queues would be empty, resulting in empty files taking unnecessary space. However, after a 20 days crawl, the round robin queue had 35% of its subqueues non-empty. As a result, I decided not to destroy a subqueue when it becomes empty as the cost of recreating the queue and the corresponding file would be higher than that of keeping empty files (empty queue files take up 4KB of space).

[8]http://fuseyism.com/classpath/doc/java/util/concurrent/LinkedBlockingQueue-source.html

5 DATASET

5.1 Result

After a 20 days crawl with 50 threads, using the hidden wiki as the starting seed, here are the current results of the crawl:

- 500 000 domains discovered
- 6 million pages fetched
- 19 millions URLs found
- 8.5 million URLs left in the queue (3GB)
- 85GB of data

By extrapolating these results to a 6 month crawl assuming that the Tor Network is big enough for the crawler to continue crawling at this rate, we would reach a total of 36 million pages fetched and 510GB of content. It is harder to estimate the number of domains that will be discovered as it will most likely not grow linearly with time.

5.2 Data Format

The entire crawl's data is stored in the *data* file. The pages fetched are stored in the *pages* folder which contains a list of *json.gzip* files. Once the file reaches a threshold size (250MB) a new file is created. After decompression, each line of the file is a *JSON* object with keys: "pageURL", "linkURLs", "content", and "title". It is to be noted that particularly long pages with more than 65K characters will be truncated The *crawl.status* file gives the global statistic about the crawl. The *urlFetchError* folder contains the *gzip* files storing the URLs whose pages generated fetch errors. The *persistentRoundRobinQueue* folder contains the unnderlying files of all the subqueues.

5.3 Graph

From the previously gathered data, we decided it could be interesting to create the network graph it represents. I order to do that, I wrote a python script that extracts the "pageURL" and "linkURLs" from the data and merges all the *gzip* files into a single file. Then, this file is used to build in RAM a Map(domain,Map(domain,linkCount)) that represents the directed graph with weighted edges. Finally, I used the python *igraph*[9] library to build the graph of the crawl and saved it in a *GraphML* format[10]. I chose this format because it allowed the nodes to have attributes (here, the name of the domain it corresponds to) contrary to other formats that emphasize on data compression. In this network, each node corresponds to a domain that has directed edges towards the domains the URLs found on its pages link to. Those edges are weighted by the number of links that direct to this domain. The previous python script was ran on a small sample of the crawl (1743 nodes and 2890 edges) and the graph obtained takes 414KB to be stored. Therefore, by extrapolating it to the current data of 500 000 nodes and around 830 000

[9]https://igraph.org/python/
[10]http://graphml.graphdrawing.org

weighted edges, the final graph would therefore take up a space of approximately 120MB.

Here are interesting characteristics obtained from this sample graph:

- Graph diameter: 8
- Average path length: 3.43
- Average shortest path length: 3.18
- Average out-degree: 1.66
- Graph density: 0.000952
- More than half of the websites do not redirect to other domains

Domain Name	pageRank	domain content
expyuzz4wqqyqhjn.onion	0.0064	Tor project
3g2upl4pq6kufc4m.onion	0.0061	DuckDuckGo
exchangenpea6tk5.onion	0.0054	cryptocurrency exchange platform

Table 1: Top 3 domains by pageRanks

Domain Name	out-degree	domain content
wiki5kauuihowqi5.onion	632	links wiki
hdwikicorldcisiy.onion	286	links wiki
wikitjerrta4qgz4.onion	258	links wiki

Table 2: Top 3 domains by out-degree

Domain Name	in-degree	domain content
www.facebookcorewwwi.onion	19	Facebook
3g2upl4pq6kufc4m.onion	11	DuckDuckGo
rutorzzmfflzllk5.onion	9	russian forum and market place

Table 3: Top 3 domains by in-degree

Furthermore, I generated two plots the Tor network graph using the Large Graph Layout (LGL) (Fig. 1) and the Atlas2 layout (Fig. 2).

5.4 Data interpretation

Caveat: the sample of the crawl used to model the topology of the network might not be representative of the characteristics obtained from studying the whole graph. From the obtained graph and characteristics, it seems that the overall topology of the Tor network consists of two main types of nodes. *Wiki nodes* that redirect to a large number of onion domains and are responsible for those circular patters in the graphs. *Standard nodes* that do not redirect to other domains and constitute the edge of the graph.

6 CONCLUSION AND FURTHER STUDIES

This project aim was to build a BFS crawler on the Tor Network. In doing so it already gathered more than 6 million pages from it and can be expected to gather around 36 million pages in a 6 month crawl or more if more resources are allocated to it. From this data, lots of interesting studies could be realized regarding its topology as well as looking at the page's content. An example of such a study would be using a combination of Natural Language

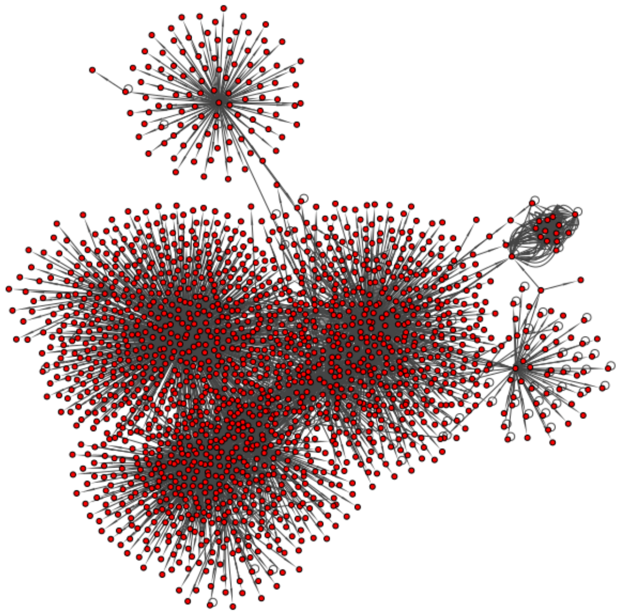

Figure 1: The LGL plot of the topology

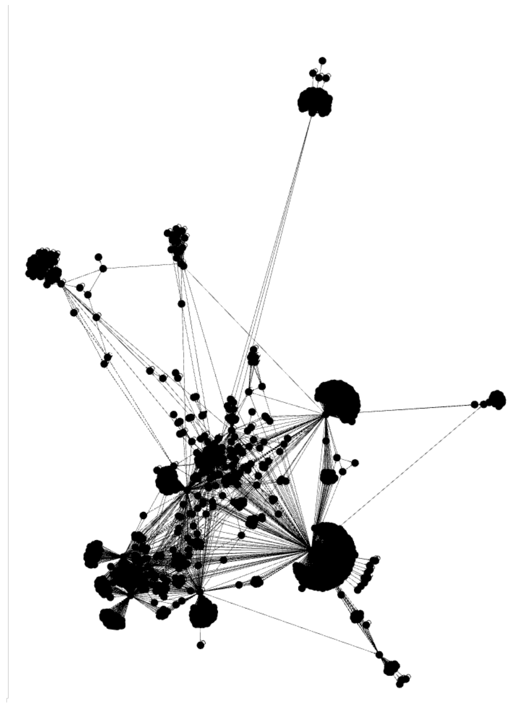

Figure 2: The Atlas2 plot of the topology

Processing techniques on the content and networks links to try and classify the domains.

ACKNOWLEDGMENTS

To my mentor, Tiziano Piccardi, who boldly suggested that I build my own crawler 4 weeks before the end of the term ;)

REFERENCES

[1] Massimo Bernaschi, Alessandro Celestini, Stefano Guarino, and Flavio Lombardi. 2017. Exploring and Analyzing the Tor Hidden Services Graph. (July 2017). https://doi.org/10.1145/3008662

[2] A. Biryukov, I. Pustogarov, F. Thill, and R. Weinmann. 2014. Content and Popularity Analysis of Tor Hidden Services. In *2014 IEEE 34th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 188–193. https://doi.org/10.1109/ICDCSW.2014.20

[3] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. 2013. Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2508859.2516651