DS Final Project Report
Dwebsites

Hugo Decroix, Simon Hayoz, Mauro Staver

January 2022

1 Introduction

The current internet is heavily reliant on big companies and their infrastructures. This means that those authorities can potentially censor content and the internet users are dependent on the robustness of their infrastructures. Decentralization comes to the rescue! This project uses Peerster primitives to develop an alternative infrastructure which offers decentralized hosting and lookup of websites.
With these additions to the Peerster system, a node is able to publish new websites, update the one they have created before and also browse the "Peerster web" in the same way as the normal web. In addition to these websites, a search engine is built into the system, allowing users to search for website in the Peerster system based on their name or content. All these operations are indistinguishable from the way the web currently behave to a lambda user as websites and search engine are displayed in a normal web browser using a simple interface to connect and navigate websites.

2 Background

2.1 Mutability and ownership

As the Dwebsites system handles modifying an existing website, the datasharing system implemented in Homework 2 was modified to handle updating files. These modifications are inspired by IPFS. IPFS is a decentralized storage system which uses content-based addressing similar to Peerster. In order to allow the content to be updated without changing the address through which that content is fetched, IPNS was introduced. IPNS [1] (Inter-Planetary Naming System) is based on SFS (Self-Certifying File System) and uses mutable pointers which point to an immutable file/folder. Each pointer has a fixed address, namely the hash of the public key associated with that pointer. As only the owner of a website should be able to update it, PKI is used, only owners of the private key associated with some pointer can change the object to which that pointer points

to. Similar pointer objects are used to allow the mutability and ownership of files and folders in Peerster.

2.2 Search Engine

The idea behind the search engine is to create a list of website that have in their content or title the searched term, and then sort them by relevance. The most famous algorithm for that kind of problem is Google's PageRank [2]. The algorithm assigns a score to each website based on the number of websites that have a link to that website and on a relevant these websites already are. Of course, the actual algorithm used by Google is more complex and fine tune, but the main idea stay the same. In this project, a basic PageRank algorithm is implemented, but in a decentralized network. For that, the PageRank graph will be stored and updated in a blockchain so that every node can have access to the ranking in a decentralized way. For string matching with content of website, the search algorithm with do an expansion ring with budget search similar to the SearchAll of homework 2, it will check on locally stored website content of each node that expansion ring visit. This method give different results based on what budget is used and on how many locally stored websites the neighbors nodes have, but it is a good tradeoff between scalability and accuracy of the search.

3 System Model, Architecture & Functionalities

3.1 Work distribution

Even if each member had to interact with the work of others, the following work distribution has been done :

- Mauro Staver : Data storage, Mutability and Ownership
- Simon Hayoz : Website, Folders, HTTP Gateway and Web redirects
- Hugo Decroix : Search Engine and PageRank

3.2 Data storage

Peerster uses content addressing, meaning that the address/key of stored data is simply its hash. This has the benefit of allowing users to fetch data from anyone who has a copy of it. Users can verify that the content is genuine by simply computing its hash and comparing it to the hash of the data they are looking for. In order to efficiently find who has the data associated with some key, we implemented a Kademlia[3] style DHT. The DHT stores key/value pairs, where a key is the hash of the data and the value is a list of addresses of nodes which store that data locally. Think of it as a huge table that stores who have what data.

The DHT implements a 160-bit key space and a recursive lookup algorithm which finds the K-closest nodes to some given key defined by the XOR distance metric, as described in the Kademlia paper. In our implementation, we used k = 4, and for the lookup algorithm, the degree of parallelism alpha = 3. A simple caching logic is used: a successful key query will store the found value on the node closest to the key, which did not have the value already.

Fetching data by its key is then accomplished by first querying the DHT, which is an O(log n) operation which returns a list of peers, and then picking a random peer from that list and fetching the data from it using a backoff-strategy if the peer is unresponsive. After the data has been fetched, the user can choose whether he wants to store this data locally, and if so, the DHT is updated with the information that he stores the data associated with the given key.

In order to upload a file to the Peerster network, the file is first split into chunks, each chunk is then hashed to obtain its key. Hashes of all chunks are merged into a metafile string which is hashed itself to obtain its key, the metahash. At the end all (chunkKey, chunk) pairs, as well as the (metahash, metafile) pair are stored in a local data store and the DHT is updated with the (chunkKey, peerAddress), and (metahash, peerAddress) key/value mappings.

To download a file addressed by its metahash, a peer has to first fetch the metafile and then gather and assemble all chunks by querying the DHT and fetching chunks from returned peers.

3.3 Mutability and Ownership

Due to content-based addressing, files stored on Peerster are immutable. This is a nuisance because every time a user wants to edit a file, he has to publish the edited file as a new file and share the new metahash with all interested users. This makes it virtually impossible to support blogs or any other frequently-updated content. In order to mitigate this problem, we will introduce a new type of objects which can be stored on the Peerster network: Pointer Records.

A Pointer Record is essentially a modifiable pointer to some other file stored on Peerster. It will serve two purposes:

1. allow new versions of a file to be accessed by the same address as the old one
2. allow only the owner of the Pointer Record to modify which file the record is pointing to

More concretely, a Pointer Record is a struct with the following fields:

- value: metahash of the pointed file
- sequence: current version number of this record
- public key: public key associated with this record
- signature: signature of the above fields (concatenated) created using the associated private key

The idea is simple: a Record's address is the hash of the public key associated with it, not the hash of the entire content of the record. This way, the Record can always be accessed at a fixed address, while allowing the pointed content, denoted by the value field, to be changed. Because of their small size, Pointer Records are stored directly in the DHT.

To create a new Pointer Record, the user simply creates an RSA key pair, initializes all struct fields and stores the private key locally to allow future modification, i.e., to allow changing the address to which the Record points to. After creation, the Record is stored on the DHT and can be associated with a human-readable name using the Tagging mechanism already built into Peerster.

Now anyone can fetch the pointed content by first fetching the Record, verifying that the Record is genuine and has not been tampered with, and finally fetching the content denoted by the address in the Record's value field. To verify that the Record is genuine, we assert that the hash of the public key matches the Record's address, and that the signature is valid.

Updating the pointed address is done by storing a new Pointer Record in the DHT, which is created by using the same private key as the previous one, but with the updated value field and sequence number increased by one.

When fetching a Pointer Record from the DHT, due to the distributed nature of the system: at any given moment in time, there could exist multiple different versions of the same Pointer Record. If this it the case, we consider the Pointer Record instance with the highest sequence number to be valid. This also implies that the DHT must conduct a full search query, i.e., it must not return after the first Record has been found because we have no guarantee that it is the newest one.

In order to keep the DHT query logic the same for all objects stored in the DHT, we simply turn off caching when querying for Pointer Records and return the first Record found. This is not as robust as running the full query, but works very well under the assumption that the K nodes closest to the key of the searched Record were alive during its newest publishing, because then the old record simply gets overwritten.

3.4 Website

As websites are distributed in Peerster, all websites but the search engine in this system are statics. Indeed, a distributed system do not have a single back-end server and cannot dynamically modify web-content and files. Thus, a website can be simplified to a collection of files. This notion is represented by a folder. The structure of such folders is the same as for a normal website: looking for *www.example.com* would return the *index.html* file at the root of the folder representing *example.com/*, and you can display any *.html* files in the folder by going to the following url: *www.example.com/any/path/to/file.html*. In Peerster, a website's folder is always named *name.domain/*, for instance *www.example.com* is named *example.com/*. This system also support relative link to content such as images, scripts, css files inside html and will use them accordingly. Figure 1 shows the basic structure of *www.example.com*.

Figure 1: Example of a folder structure of a website

3.5 Folders

The current version of Peerster supports only upload and download of files and not directories. In order to allow seamless management of websites, the support for folders has been added by modifying the Peerster's datasharing system and the Pointer Record struct introduced above.
The modifications add the following fields to the Peerster's Pointer Record struct:

- name: the name of the current file or folder, this is solely used in folder's reconstruction as the links to different addresses below are not enough to reconstruct the folder if the name of the different files are unknown
- links: a list of addresses

A directory will have the value field empty, and the links field will contain addresses of subfolders or files contained within this directory. This architecture allows us to store files and directories of arbitrary structure under one unified system using content-based addressing.

3.6 HTTP Gateway

A Peerster user cannot use the DWebsites system directly in a browser, as current browser fetch website from urls directly on the normal internet. To let users use the Peerster system instead, a proxy can be attained by the user and will redirect all his requests to the Peerster system. The Peerster proxy will listen for HTTP requests, process them and return an HTTP response directly to the users' browser. When a user goes to the local page *.../gui/web/website.html* a simple homepage with a search bar, browse bar and a manage panel will be served. The search bar allows users to search for resources using regular expressions. The manage panel is a place where users can easily publish content, create named Pointer Record objects, or update records for which they hold the key for. All keys management is done by the Peerster client and the users interact with the system using a simple UI. Finally, the browse bar let the user enter websites "URL" directly and also let the user cache fetched DHT locally for faster response time of already visited websites.

3.7 Web redirects

As websites are tied together through links and due to the HTTP Gateway, links have to be adapted so that if a user clicks a link to *www.example.com* in a page, it does not redirect him to *http://www.example.com* but rather to *localhost:proxy_port/www.example.com*. To do so, The proxy gateway updates all external links in all html file of a website just after downloading it and before it is displayed to the user. Doing so ensure that a user is always redirected to the website on Peerster system on click. Moreover, doing so on download also ensures that websites on the Peerster system are untouched and exactly like the users have created them. This would let more freedom for new developers to totally change the HTTP gateway to something else, for instance a browser that would directly query the Peerster system instead of the normal internet.

3.8 Search Engine

The search engine works with two main blocks. It first does a query that return a list of website that match the searched regular expression in their title or content. Then, from a blockchain consistent PageRank ranking, it sorts the list by relevance.

The search algorithm is a modification of the SearchAll method used in Homework 2. The expansion ring and budget system is the same, with the search request being send to random neighbors by splitting evenly the budget. The difference is that the request does not expect file info and chunk hash for a name, but a list of match website. For that, the node receiving a request will proceed to that standard action to keep the expansion ring going, but to create the response, it will check for string matches in its local naming store for website in title, if so add the website to the reply list. The search also provide option in content search, for that the node will iterate all .html file of each locally stored website and check for string matches in the text content of the file. This is done by doing a tokenized iteration over all the text bodies of the html file.

Finally, after a timeout, the searching node can fetch and aggregate the results from all the replies and sort the matching websites by their ranking from the PageRank ranking. The result is then displayed in order on the GUI with proper links that can work with the HTTP Gateway described above.

The Figure 2 show a typical activity diagram of the search pipeline.

3.9 PageRank

The goal of the PageRank algorithm is to rank the website by relevance. It works by counting the number and quality of links that points to each website. For its computation, the algorithm needs a directed graph, with the nodes being the websites and the directed edges the links from one website to another. Moreover, the computation is of order $O(n + m)$ with n the number of nodes and m the number of edges in the graph. Therefore, it is not scalable to compute the ranking for each search request nor to reconstruct and query the graph.

To avoid, each node have their own copy of the graph and is keep consistent across all node by a blockchain mechanism with a Paxos consensus similar to what is used for name consistency. From that point, each time a website is added/modified by a user, a walk is done on all html files of the website to fetch all the potentials links to other website, all these new edges are then added to the blockchain. Each time an edge is added, the nodes need to compute the new ranking.

For that, the iterative page rank algorithm is used, all the website start with an even normalized ranking and each iteration the ranking is updated with the following formula: $PR(p_i; t+1) = \frac{1-d}{N} + d \sum_{p_j \in P_i} \frac{PR(p_j;t)}{L(p_j)}$ with p_i the i-th website in the graph, t the iteration, N the number of websites, P_i the set of all website excluding p_i and $L(p_i)$ the number of outgoing links from website p_i and d the damping factor set at 0.85 with correspond to the probability that the imaginary web user continue "surfing" by clicking another link. Finally, the algorithm when a convergence threshold is reached by comparing the absolute sum of the difference between each iteration ranking. In the implementation, the convergence threshold is set at 0.01.

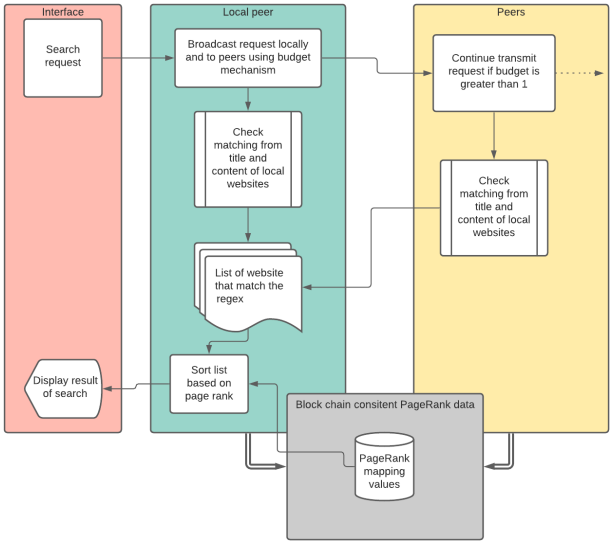

Figure 2: Search engine activity diagram

4 Evaluation

4.1 Hot keys

Consider the scenario where N nodes all try to download the same file from the network at the same time. Considering that K other nodes store the data

locally, the expected number of data fetch requests that each of the storing nodes will receive should be $\approx \frac{N*(fileSize/chunkSize)}{K}$. This is because when a node initiates the download procedure, it finds, for each chunk key, a list of peers which store that chunk key and then randomly samples one peer out of the list and sends him a data fetch request. After each downloaded chunk, the node can store it locally and accordingly update the DHT with its information. The idea is that popular content gets populated throughout the network and does not cause bottlenecks as many users try to fetch it because the requests will be spread out.

We have run an experiment where N nodes download the same file from the network one after the other, mimicking a rising trend. At the beginning, only one node has the file stored locally, and as nodes download it one after the other, we expect that the requests get more and more spread out. The empirical data for N = 10 nodes can be observed in Figure 3 where each bar represents the number of data fetch requests that the node received from each querying node. The results clearly show a downward trend in the number of requests per node as the content gets more popular, thus eliminating bottlenecks and proving our analysis correct.

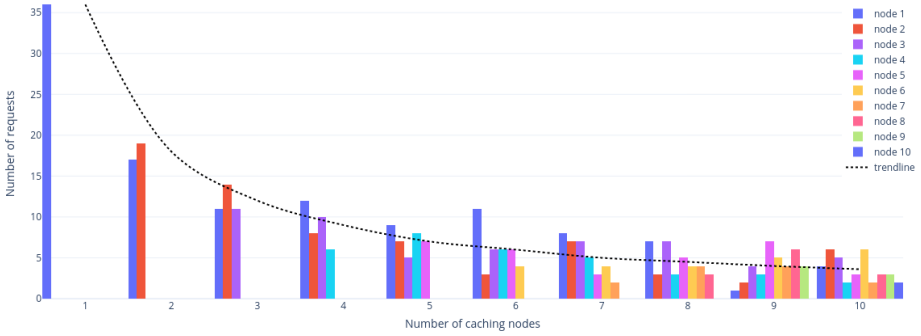

Figure 3: Request distribution for hot keys

4.2 Website load time

In this section, we will compare normal web pages' load time and Peerster web pages' load time. Based on [4], the average time for a desktop web page to load is 10.3 seconds. On a real decentralized system (i.e., with nodes not on the same localhost), we can consider that the time to retrieve all data needed on the Peerster system is slower, as you have to go through multiple nodes to get the content of the website. Moreover, the current Peerster way of retrieving a web page will actually get the whole folder containing the whole website, which is slower than the HTTP requests of the normal web that will retrieve a unique file. On the other hand, this means that Peerster could actually be better (depending on the nodes geography) if the site is very small. Indeed, for

instance a website with a unique page, multiple images, scripts and css files that are use in the same html file:

- On Peerster: all files will be retrieved before displaying the website
- On the normal web: the first HTTP request will retrieve the html file, then one request per other files will be made

This example shows that it could be possible in very specific cases for the normal web to be slower than Peerster web. Peerster also use a local cache with the different folders and when a new request is made, it will just compare the sequence number it has locally and the one on the Pointer Record. If they are the same, then it can use the version it has in its cache, thus being nearly as fast as HTTP request with cache.
Finally, we compared loading a website locally with a proxy server serving the file directly and on Peerster web locally (multiple nodes on the same computer). We used the Chrome extension Page load time to measure it. Here are the results for two different sites, displaying the same html file:

Number of files and size	Peerster [ms]	normal redirect [ms]
4 files/8.2 KB	152	109
411 files/20.5 MB	474	110

Table 1: Loading time in ms for different website's size

We can see in the table above the effect of full folder download, as the normal redirect website is not affected at all, whereas Peerster loading time is multiplied by ≈ 3.

4.3 PageRank's computation performance

One of the potential scaling issue with the PageRank algorithm is the computation required to compute the ranking from the graph, this operation is done on every node each time a new edge is added to the graph. In theory, the expected complexity of the iterative implementation is of, $O(n+m)$ with n the number of nodes and m the number of edges in the graph. To evaluate the implementation, a comparison of computation time on the same computer for different graphs with increasing number of nodes. Different measure are done with a different average of outgoing links per website, these links are created at random for each node such that the average link per node over all node is the one tested. The Figure 4 display the results in logarithm axis scale on both axis. As expected, we see that the expected linear complexity is achieved on the implementation.

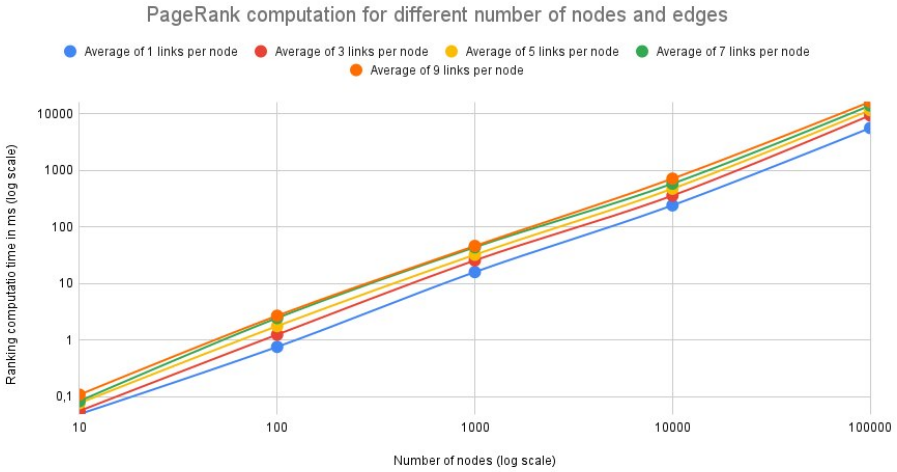

Figure 4: PageRank's computation time for different number of nodes and edges

We also see from the results that the computation time on a standard laptop is under 1 second when there are less than 10 000 nodes in the graph, which is a reasonable value, but over 10 000 nodes the computation time less negligible. Note that the convergence threshold can be fine-tuned, a time limit can be added to trade off between accuracy of the ranking and performance. Finally, a potential improvement is to parallelize the computation for very large graphs.

References

[1] "Ipns," https://github.com/ipfs/specs/blob/master/IPNS.md.

[2] "Pagerank," https://en.wikipedia.org/wiki/PageRank.

[3] "Kademlia," https://en.wikipedia.org/wiki/Kademlia.

[4] "Website time load statistics," websitesetup.org.