Distributed File Storage using Blockchain

May 1st, 2018

Sheharyar Naseer
snaseer@nyit.edu
ID # 1188379

Abstract

As we technologically progress, it is becoming extremely common for both businesses and individuals to use cloud storage services like Google Drive, Dropbox, iCloud and Amazon for storing their files. These services offer them the convenience of storing and retrieving their files from anywhere in the world using the Internet at relatively low costs, but they have introduced their own security and privacy issues because of the way these files storage systems are implemented.

Most file storage services store the users' files without properly securing or encrypting them. Those that do, in most cases, have full read-access to the files internally. This has raised serious privacy concerns, especially during the massive Dropbox hack in 2012 which affected more than 68 million account holders [1], and the iCloud leak in 2014 [2].

In this paper, the author draws inspiration from the *Blockchain* distributed ledger technology, the *Sia* cryptocurrency and various file distribution protocols to explore blockchain as an alternative strategy for file storage that offers security, privacy and reliability. For implementation, we look towards the powerful BEAM Virtual Machine, offering process supervision, node distribution and a strong Actor model based message-passing system out of the box. The author designs and implements a basic prototype of the suggested strategy, and discusses the results and the challenges of the approach.

Literature Review

This literature review covers three different technologies that this project draws inspiration from; which are the *Blockchain* Distributed Ledger technology, *Sia* (Simple Decentralized Storage) and the *Bittorrent* peer-to-peer file sharing protocol.

Let's start with the Blockchain from where it all started. Going back to 90's the first work on implementing cryptographic secured chain of blocks was described in 1991 by Stuart Haber and W. Scott Stornetta [3]. After about a decade and a half the first distributed blockchain was theorized by an anonymous person or a group called as Satoshi Nakamoto in 2008 and during the following year implemented Bitcoin which then served as a public ledger for all transactions [4].

Blockchain is a relatively simple concept based on the Merkle Tree data structure, but when implemented it leads to a whole new set of technologies. The technology is still in its early stages and there are various ways to implement it which completely depend upon the problem statement. *"At the superficial level blockchain allows a network of computers to agree at regular intervals on the true*

state of distributed ledger," [5]. What exactly is a distributed ledger? It is a type of database which is shared, replicated, and synchronized among members of a network. Whereas every record in a distributed ledger has a unique timestamp and cryptographic signature attached to it which identifies it uniquely [6]. Such collection or block can contain different types of data varying from account credentials, transactional records or other arbitrary data. The security of this block is maintained through various cryptographic algorithms and game theory techniques [5].

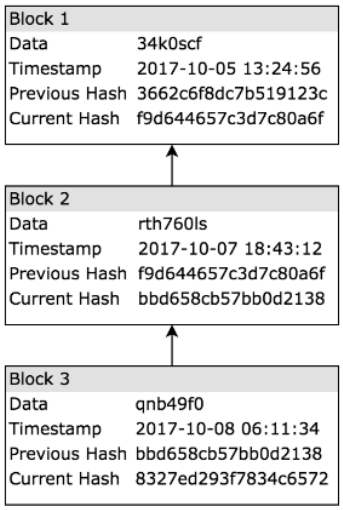

Figure 1 – Example Implementation of Blockchain

Essentially it is a distributed network of all digital events which take place and are shared among various people who take part in it. Transactions which take place in the block are verified by an agreement of majority of the people in the system. It is also immutable, i.e. once the data enters the chain can never be erased [7]. Figure 1 shows an example implementation of a Blockchain where each block in the chain, references the cryptographic hash of the previous one which would change drastically if even a small change would be made in the data, timestamp or anything else. In this way, each entry in the Blockchain automatically verifies and secures data in previous blocks, making it virtually impossible to tamper with.

Bitcoin is a peer-to-peer electronic cash system, one of Blockchain's most famous and the very first successful application was brought to life in 2009 by the brilliant Satoshi Nakamoto [8]. The idea behind this invention was to omit the need of a financial institution which would act as intermediate step in online payments sent from one party to another.

Digital signatures posed as the solution though double-spending was a problem which was then solved by introducing a peer-to-peer network where the network timestamps each and every transaction by hashing them into a rolling chain forming a record that cannot be changed without redoing the proof-of-work [9].

The next important work that's been done in the field of decentralized storage is Sia: Simple Decentralized Storage. The idea of Sia was first put to implementation at HackMIT 2013 [10]. Sia is basically a decentralized cloud storage platform which competes with existing cloud storage services at various levels. In a world where all the datacenters which are owned by single owners or a company Sia allows anyone to earn something by putting their storage space or hard drive on rent and as far as data integrity is concerned Sia uses redundancy and cryptography to ensure that it is maintained [10].

Sia works by storing contracts that are made between various peers and between the owners of the storage and their clients. The data, its type and the price of storing it is defined in that contract. It also requires storage owners to show proof that they are still storing the client's data at regular intervals [11]. Sia uses blockchain to store these contracts and they can be verified publically because of its

blockchain implementation, so there is no need for the clients to personally verify the storage proofs [11].

Sia is the only network where there is complete independence to users to where their data ends up on. More importantly if the owner loses data then they tend to lose the revenue as well as any collateral money that they put up to make sure your data is safe [12].

For file storage and sharing between nodes, we look towards BitTorrent. BitTorrent is a protocol which enables the reliable and quick transfer of data from many to many peers in distributed network. It is one of the most widely used peer-to-peer program ever made. The man behind BitTorrent thought the idea of breaking down files in chunks, encrypting every chunk and storing them in different locations can be used in file sharing. The very first beta version of BitTorrent was released in 2001 [13]. As of 2009 peer-to-peer networks consists of 43% to 70% of all internet traffic [14]. BitTorrent has around 15 to 27 million concurrent users at any time [15].

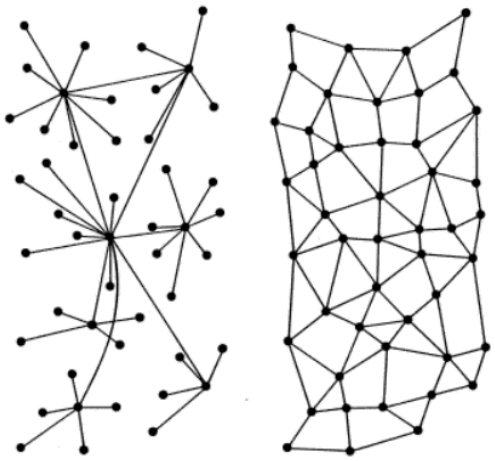

Figure 2 – Decentralized Networks (Left) vs. Distributed Networks (Right) – from "On Distributed Communication Networks" [18]

BitTorrent makes the distribution of large files very efficient by consuming less bandwidth and hence, making it is easier for the originator. It consists of a metafile, popularly known as torrent which contains information or metadata about the host or the publisher and one or more trackers whose function is to co-ordinate the distribution of the shared files. A peer first creates a torrent file referencing the data that is to be shared. This torrent file is made to include addresses of different trackers, that other peers can connect to join the network. All of these peers assist each other in downloading the file and making sure everyone has a copy [16].

It is also important that we make a distinction between Decentralized and Distributed networks. The earliest work to discuss this in terms of technological systems is Paul Baran's "On Distributed Communication Networks" [18] from 1964. Figure 2 visualizes their difference very clearly. Bittorrent and IPFS are decentralized, while Blockchain and Bitcoin are distributed and Sia uses a mixture of both (keeping its ledger distributed while keeping the stored data decentralized).

The computer which hosts the original file is called a seed and the client who wants to download the file needs to request the same from a seed using the BitTorrent client. After requesting the client gets one piece of file and over a period of time remaining pieces from various other peers via peer-to-peer communication. Each computer is downloading some pieces of file from some of these peers while simultaneously uploading other pieces of file to other peers. All the computers which take part in this communication and co-operating in this way are called as *swarm*. How quick this process takes place or the speed of the process depends upon how popular the file is [17].

Proposed System

The author is implementing a Distributed Blockchain File Storage (DBFS) system which aims on targeting three main concerns of the current world which are security, privacy and reliability. The application is designed as a private blockchain [34], allowing it to be much more secure and efficient compared to other types, and putting the reliability of the data in to the hands of the organization running it.

The system will consist of a basic user interface for the end users to interact with the files they will be uploading to the server. The system consists of 2 main components, a client application that performs encryption and signing of data; and a backend node server where all signatures and blocks are verified and validated, and the consensus between all nodes in a network is maintained. Our custom Blockchain implementation would allow us to store data references, its timestamps, executed actions and the cryptographic signatures of the users performing those actions, using a hybrid of both Symmetric and Asymmetric key cryptography. This also provides a powerful way to maintain and check the data consistency and keep track of its changes, making it impossible to alter the data history and its integrity. For efficient delivery of data, we'll use Erlang's virtual nodes and message passing by writing a naïve *P2P* protocol to transfer the data from many to many peers in a distributed network.

Backend Application

For the core backend system, we've created the application [19] in the Elixir programming language [24], on top of the Erlang BEAM Virtual Machine. The backend consists of one Supervisor, one GenServer and three module contexts. The four main modules are *Block*, *Blockchain*, *Crypto* and *Consensus*. The *Blockchain* module provides an interface to work with the Blockchain data structure itself consisting of Linked List of many *Blocks*. The blockchain starts with a zero block, a special data structure to seed the rest of the chain.

```
%DBFS.Block{
  data: %{
    file_name: "Secret Document.pdf",
    file_size: "192517",
    file_type: "application/pdf",
    file_hash: "5962416265CBE4D766F2F474F0A36...",
    file_key:  "ZDMyZTI5YjVkNGZmOTJhMDc4NTQ4Y..."
  },
  type:      :file_create,
  timestamp: ~N[2018-03-14 18:33:58.198337],
  creator:   "2D56ADF32B424547494E205055424C9...",
  signature: "8481DA83D6CB75CBE411031AADE5BBD...",
  prev:      "C0579CA7001A896888AF57895FF8021...",
  hash:      "9099520C15740B91FC4BC5B13E1434F..."
}
```

Figure 3 – Our Blockchain Implementation [19]

The *Block* module defines the block struct, which contains the type of block, creation timestamp, stored data, owner's encoded public key, hash of the previous block, cryptographic signature verifying authenticity of the data and the hash of the entire block. Figure 3 shows this in the form of a native Elixir struct. The cryptographic actions like signing, hashing and verifying blocks are performed using the *Crypto* module which implements RSA Asymmetric key cryptography and the SHA256 hashing function using the open-source *RsaEx* [21] and *ExCrypto* [22] libraries, as well as the *:crypto* and *:public_key* modules in the Erlang standard library [23].

The *DBFS.Consensus* context consists of a few modules responsible for ensuring that all nodes in the network are in sync. It is a simple implementation of the Raft Consensus algorithm [31] which involves electing one of the nodes in the network as the leader, which is then used as the reference point for all of the other nodes in the network. The context also holds a

Status GenServer [25] which describes the current synchronization state. The global network leader state is stored via a distributed *Mnesia* table [32], which is already built in to the Erlang OTP framework. Once a leader is elected, all nodes can compare their chain with the leader to see where it diverges and get the next block & associated file, using built-in RPC calls [33]. There is also a Supervisor responsible for making sure that the application recovers gracefully in case of unforeseen issues and errors and that the GenServer is always running.

The *DBFS.Consensus.Cluster* module implements the actual function calls responsible for sending and retrieving blockchain blocks and their associated files between nodes and updating their internal states.

The permanence of data is handled by a PostgreSQL database via the Ecto database wrapper [26], giving us strict control and validation over the blockchain records (blocks). The database contains only two tables, namely *blockchain* and *blocks*. The *blockchain* table is responsible for caching current state of the blockchain, such as the number of blocks, synchronization status and the hash of the last block. Figure 4 shows the structure of the table below:

Column	Type	Nullable	Storage
id	bigint	not null	plain
data	jsonb		extended

Figure 4 – "blockchain" table structure

This table always holds one record at max which contains an efficient *jsonb* map of the cached state of the blockchain. We also set up an index on this table in case we decide to manage multiple blockchains in the same database in the future:

- `"blockchain_pkey", PRIMARY KEY, btree`

The second table we use is *blocks* which contains the detailed individual data of each block previously shown in Figure 3, including block type, previous hash, cryptographic signature of the creator and associated metadata. The table in Figure 4 shows the structure of the *blocks* table. Like the *blockchain* table it also contains some indexes on different columns to speed up searching and fetching blocks according to different scenarios.

These indexes are:

- `"blocks_pkey" PRIMARY KEY, btree (id)`
- `"blocks_creator_index" btree (creator)`
- `"blocks_hash_index" btree (hash)`
- `"blocks_prev_index" btree (prev)`

Column	Type	Storage
id	bigint	plain
timestamp	naive datetime	plain
type	integer	plain
prev	character varying(255)	extended
hash	character varying(255)	extended
signature	text	extended
creator	text	extended
data	jsonb	extended

Figure 5 – "blocks" table structure

Finally, we have a *dbfs_web* sub-application within the application supervision tree, written in the Phoenix Framework [35]; which exposes RESTful HTTP API calls in JSON, as well as a WebSockets channel that actively transmits network state, allowing client applications to interact with the blockchain.

Web Client

For ensuring that the data is securely stored without having all systems load the entire blockchain locally, we also created a simple and easy to use web-interface [20]

to connect with the blockchain, view status and upload files.

The WebUI is written in Javascript using the popular frontend framework by Facebook; React [27] for the client-side scripting and the Bulma UI framework [28] for the CSS styling of the page elements. Almost all of the cryptographic features of the backend were replicated on the client-side, but instead using popular Javascript cryptographic libraries like *jsrsasign* and *cryptico* [29][30].

All of the dependencies are managed using both NPM [36] and Yarn [37], along with Webpack [38] for compiling all of the code into a useable Javascript application build. The code is organized into three parts: components, which are React DOM elements and their tied logic; services, which use the Axios library [39] for connecting to the backend's API; and core modules that hold the main functionality of the app. The core modules consist of *crypto.js*, which handles encryption, decryption, signing, verification, hashing and more; and *dbfs.js*, which handles block construction and file uploads/downloads.

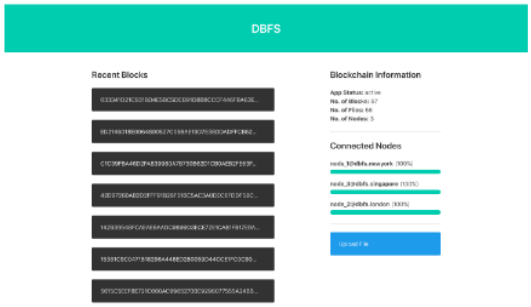

Figure 6 – Initial Screen of the Web Client

The client's initial screen (Figure 6) shows the recently blocks in the chain, along with some statistics of all nodes in the network, and an option to upload a new file. The page also keeps an active WebSocket connection open to the backend's status channel, to display real-time changes to the state of the nodes in the network.

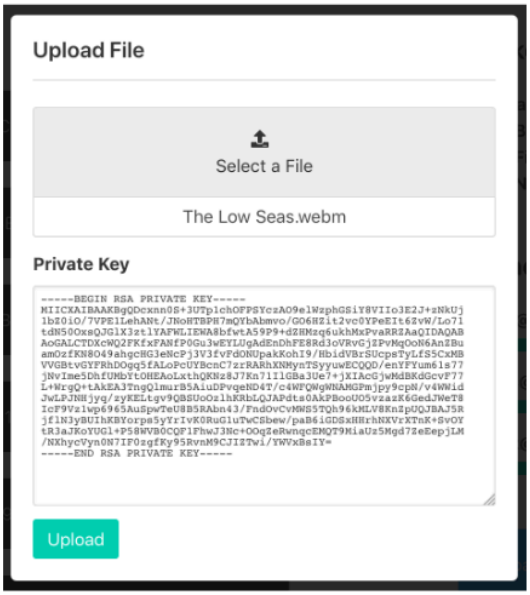

Figure 7 – File uploader asking for users' private key to encrypt the file and sign the block

Clicking on the "Upload File" button opens a modal which accepts a file and the users' private key (Figure 7). After selecting the file and entering the private key, the user can upload the file to the application. Since Asymmetric key cryptography is inefficient for large data, a random 256-bit AES key is generated and is used to encrypt the file, and then the key itself is encrypted using RSA. A new block is constructed with the timestamp, hash of the previous block, the user's encoded public key and the file metadata, including the encrypted AES key. Finally, the block is signed and hashed, before being serialized into a JSON object and sent over to the backend to be inserted into the blockchain via the previously exposed API. Since everything is performed at the client-side and the private keys are never

transferred over the network, this mathematically guarantees the security and privacy of the users' files (as long as the user's private key isn't compromised).

The server receives the object and unpacks it by using the Base64 decoding scheme (even though it can't understand the underlying encrypted data) and recalculates the hash and verifies the signature to make sure no tampering has been done with the file. Once validated, the entire object is finally inserted into the blockchain where it can remain forever without ever being changed.

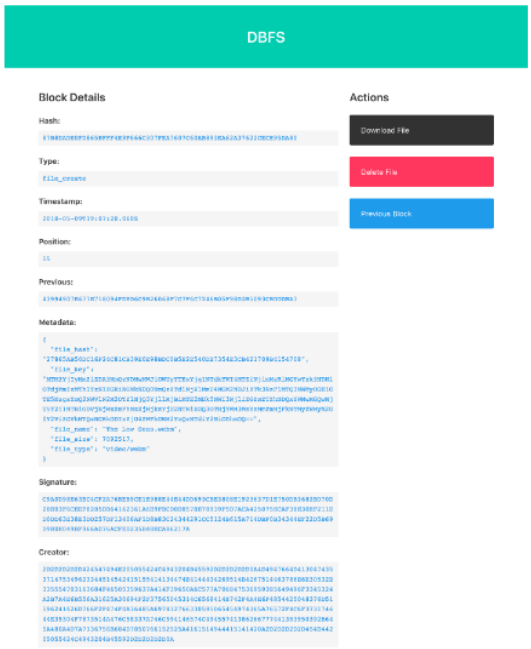

Figure 8 – Block Explorer with file actions

Back in the web client, clicking on a block or manually typing its unique hash, opens the Block Explorer displaying the details of that specific block (Figure 8), along with an option to download the associated file. Clicking on the "Download File" button asks for the users' private key and performs the entire process in reverse; downloading the encoded file, decrypting the embedded AES key, using the key to the decrypt the file within the browser, and invoking the browser's download call for the decrypted file. There is also a "Delete File" button (for files that haven't already been deleted), which when pressed, also asks for the user's private key. When the key is entered and verified, a new block of type "file_delete" is generated and inserted into the blockchain, effective notifying all nodes in the network to purge the associated file from disks.

Running the Code

The application has a lot of moving parts, but the core resides in two main codebases, the backend [19] and the client [20]. The backend should run on every server which should maintain its own copy of the blockchain and the files, but the client can be run on any system which wants to access or interact with the blockchain as long as it's address is specified. For the sake of simplicity, we will run both applications on the same (Linux) system in this example.

First thing we need to do is install the dependencies. The backend application needs Git, ASDF, Erlang, Elixir, Postgres and a few Hex Packages. The client also needs to Git & ASDF, along with NodeJS, Ruby, Sass, Yarn, React, and other NPM libraries.

Here are the steps needed for backend (you might need `sudo` or might choose to have custom properties set for Postgres, depending on your environment):

```
$ apt-get update
$ apt-get upgrade
$ apt-get install git git-core

$ git clone https://github.com/asdf-
  vm/asdf.git ~/.asdf --branch v0.4.3
$ echo -e '\n. $HOME/.asdf/asdf.sh'
  >> ~/.bashrc
```

```
$ asdf plugin-add erlang
$ asdf plugin-add elixir
$ asdf install erlang 20.1
$ asdf install elixir 1.5.2
$ mix local.hex

$ apt-get install postgresql
  postgresql-contrib

$ git clone to.shyr.io/dbfs ~/dbfs
$ cd ~/dbfs
$ mix deps.get
$ mix compile
```

Here are the steps needed to install the dependencies for the web client:

```
$ asdf plugin-add ruby
$ asdf plugin-add nodejs
$ asdf install ruby 2.3.3
$ asdf install nodejs 9.6.1
$ gem install sass
$ npm install -g npm
$ npm install -g yarn

$ git clone to.shyr.io/dbfs-web
  ~/dbfs-web
$ cd ~/dbfs-web
$ yarn install
```

For the first node of the backend, we need to create its instance and initialize the blockchain database. For other nodes, we can simply create their instances without creating the blockchain (They will automatically be synchronized when we start them):

```
$ cd ~/dbfs
$ NODE=newyork PRIMARY=1 mix do
  ecto.create, ecto.migrate,
  ecto.seed
$ NODE=london PRIMARY=0 mix do
  ecto.create, ecto.migrate,
  ecto.seed
$ NODE=singapore PRIMARY=0 mix do
  ecto.create, ecto.migrate,
  ecto.seed
```

To finally start the backend nodes, you need to pass them a port, node name and boot-up config. You can start one node or multiple, depending on the requirements.

```
$ NODE=newyork PORT=3000 elixir
  --name node_1@dbfs.newyork --erl
  "-config sys.config" -S mix
  phoenix.server
```

You can similarly start a London node on port 3001 and Singapore node on port 3002. If you decide to run multiple nodes together, you also need to initialize the distributed Mnesia tables. First enter the REPL command-line of one of the running nodes, and enter the following:

```
iex> DBFS.Consensus.Global.setup
```

Finally, to access the blockchain from the web-client, start it on port 4000. You can optionally do this for every node, as long as you specify a port thousand more than the backend's (4001, 4002, etc.).

```
$ cd ~/dbfs-web
$ PORT=4000 yarn start
$ PORT=4001 yarn start
$ PORT=4002 yarn start
```

These actions should be performed on a different system for each node, but for the sake of simplicity have been done on the same system here. A few helper scripts have also been included in the `priv/scripts` directory of the backend application and can be used to bootstrap it with some (invalid) initial data, which can be very useful for testing.

Process management tools such as `foreman` can also be used for quickly testing multiple nodes on the same machine.

Future Work

There's opportunity for a lot more work and improvements on this project. Some of the other features that can be added to DBFS are:

- *More types of blocks* – we can allow creation of a diverse set of blocks that could allow deletion of previously shared files, publicly-accessible sharing and multi-key user-based sharing of files
- *Advanced implementation of file distribution across nodes* – to allow faster and more efficient transfer of files by breaking them into smaller chunks (like the original BitTorrent protocol)
- *Improved Conflict Resolution* – to resolve conflicts and differences in "Blockchain Branching" when the blockchain data on one node completely diverges from another
- *Nearest Node Resolution* – So the Web Client can automatically find the nearest and fastest node for quicker file transfers
- *Better Key Handling* – By using compression, caching, removal of unnecessary data, and support for encrypted ECDSA Keys
- And much more!

Clearly, there's much more possible in the world of Blockchains and File Storage.

Conclusion

Distributed File Storage using Blockchain was certainly an exciting project to work on, but there's only so much that can be accomplished by an individual in the span of 4 months. The author implemented a very naïve version of the blockchain by drawing inspiration from other extremely popular projects and protocols, and though there were a lot of issues in implementation initially, the final application still turned out to be a great success.

There is still room for a lot of improvements, but we leave that for the future work on this topic.

References

[1] BBC News (2016) – *Dropbox Hacked*

http://www.bbc.com/news/technology-37232635

[2] Wikipedia (2014) – *iCloud leak of photos*

https://en.wikipedia.org/wiki/ICloud_leaks_of_celebrity_photos

[3] Blockchain Whitepaper

https://www.blockchain.com/whitepaper/index.html

[4] The Economist (2015) – *The great chain of being sure about things*

https://www.economist.com/news/briefing/21677228-technology-behind-bitcoin-lets-people-who-do-not-know-or-trust-each-other-build-dependable

[5] MIT Sloan (2017) – *Blockchain explained. MIT Sloan Assistant professor Christian Catalini*

http://mitsloan.mit.edu/newsroom/articles/blockchain-explained/

[6] IBM developerWorks (2016) – *Blockchain basics: Introduction to distributed ledgers by Sloane Brakeville and Bhargav Perepa*

https://www.ibm.com/developerworks/cloud/library/cl-blockchain-basics-intro-bluemix-trs/index.html

[7] Sutardja Center for Entrepreneurship & Technology, Berkeley University of California (2015) – *BlockChain Technology*

http://scet.berkeley.edu/wp-content/uploads/BlockchainPaper.pdf

[8] The New Yorker (2011) – *The Crypto-Currency by Joshua Davis*

https://www.newyorker.com/magazine/2011/10/10/the-crypto-currency

[9] Bitcoin.org – *Bitcoin: A Peer-to-Peer Electronic Cash System by Satoshi Nakamoto*

https://bitcoin.org/bitcoin.pdf

[10] Sia – *Sia Blockchain Technology*

http://www.sia.tech/

[11] Sia: Simple Decentralized Storage – *whitepaper by David Vorick and Luke Champine*

https://www.sia.tech/whitepaper.pdf

[12] News.ycombinator.com – *A cryptocurrency operated file storage network*

https://news.ycombinator.com/item?id=13723722

[13] Department of Telematics, NTNU (2005) – *Peer-to-peer networking with BitTorrent*

http://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf

[14] Wikipedia – *BitTorrent*

https://en.wikipedia.org/wiki/BitTorrent

[15] MLDHT – *BitTorrent mainline DHT measurement*

https://www.cl.cam.ac.uk/~lw525/MLDHT/

[16] Summary of BitTorrent protocol by guest lecturer *Petri Savolainen*

https://www.cs.helsinki.fi/webfm_send/1330

[17] Explainthatstuff (2017) – *BitTorrent*

http://www.explainthatstuff.com/howbittorrentworks.html

[18] Baran, P. (1964) – *On Distributed Communication Networks.* IEEE Transactions on Communication Systems.

https://www.rand.org/content/dam/rand/pubs/papers/2005/P2626.pdf

[19] Naseer, S. (2018) – *DBFS Source Code* on GitHub

https://github.com/sheharyarn/dbfs

[20] Naseer, S. (2018) – *DBFS Web Client Source Code* on GitHub

https://github.com/sheharyarn/dbfs-web

[21] Noskov, A. (2016) – *RsaEx Elixir Library* on GitHub

https://github.com/anoskov/rsa-ex/

[22] Austin, J. (2015) – *ExCrypto Elixir Library* on GitHub

https://github.com/ntrepid8/ex_crypto/

[23] Ericsson (1999-2018) – *crypto* and *public_key* Erlang modules

http://erlang.org/doc/man/crypto.html
http://erlang.org/doc/man/public_key.html

[24] Valim, J. (2013) – *Elixir Programming Language*

http://elixir-lang.org/

[25] Elixir Core Team (2016) – *GenServers*

https://hexdocs.pm/elixir/GenServer.html

[26] Elixir Ecto Team (2015) – *Ecto* project on GitHub

https://github.com/elixir-ecto/ecto

[27] Facebook (2013) – *React Frontend Framework*

https://reactjs.org/

[28] Thomas, J. (2016) – *Bulma UI Framework*

https://bulma.io/

[29] Urushima, K. (2012) – *jsrsasign* Javascript Library on GitHub

https://github.com/kjur/jsrsasign

[30] Terrel, R. (2012) – *cryptico* Javascript Project

https://github.com/wwwtyro/cryptico/

[31] Ongaro, D. (2012) – *Raft Consensus Algorithm*

https://raft.github.io/

[32] Ericsson (1997-2018) – *Mnesia*

http://erlang.org/doc/man/mnesia.html

[33] Ericsson (1997-2018) – *RPC*

http://erlang.org/doc/man/rpc.html

[34] CoinSutra (2017) – *Private Blockchains*

https://coinsutra.com/private-blockchain-public-blockchain/

[35] McCord, C. (2014-2018) – *Phoenix Framework*

http://phoenixframework.org/

[36] NPM Inc. (2010) – *npm.js*

https://www.npmjs.com/

[37] Facebook, Google, Exponent (2016) – *Yarn Package Manager*

https://yarnpkg.com/

[38] Webpack (2012) – *Webpack*

https://webpack.js.org/

[39] Axios (2014) – *Axios*

https://github.com/axios/axios