Decentralizing indexing and bootstrapping for online applications

Schutz, Pierre

2021

Schutz , P , Gal , S , Chatzopoulos , D & Hui , P 2021 , ' Decentralizing indexing and bootstrapping for online applications ' , IET Blockchain , vol. 1 , no. 1 , pp. 3-15 . https://doi.org/10.1049/blc2.12001

http://hdl.handle.net/10138/356240
https://doi.org/10.1049/blc2.12001

Received: 28 September 2020 | Accepted: 22 December 2020

DOI: 10.1049/blc2.12001

ORIGINAL RESEARCH PAPER

Decentralizing indexing and bootstrapping for online applications

Pierre Schutz[1] | Stanislas Gal[2] | Dimitris Chatzopoulos[3] | Pan Hui[3,4]

[1] Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland

[2] ETH Zürich, Zurich, Switzerland

[3] The Hong Kong University of Science and Technology, Hong Kong, China

[4] University of Helsinki, Helsinki, Finland

Correspondence
Pan Hui, The Hong Kong University of Science and Technology, China.
Email: panhui@cse.ust.hk

Abstract

Peer-to-peer (P2P) networks utilize centralized entities (trackers) to assist peers in finding and exchanging information. Although modern P2P protocols are now trackerless and their function relies on distributed hash tables (DHTs), centralized entities are still needed to build file indices (indexing) and assist users in joining DHT swarms (bootstrapping). Although the functionality of these centralized entities are limited, every peer in the network is expected to trust them to function as expected (e.g. to correctly index new files). In this work, a new approach for designing and building decentralized online applications is proposed by introducing DIBDApp. The approach combines blockchain, smart contracts and BitTorrent for building up a combined technology that permits to create decentralized applications that do not require any assistance from centralized entities. DIBDApp is a software library composed of Ethereum smart contracts and an API to the BitTorrent protocol that fully decentralizes indexing, bootstrapping and file storing. DIBDApp enables any peer to seamlessly connect to the designed smart contracts via the Web3J protocol. Extensive experimentation on the Rinkeby Ethereum testnet shows that applications built using the DIBDApp library can perform the same operations as in traditional backend architectures with a gas cost of a few USD cents.

1 | INTRODUCTION

Peer-to-peer (P2P) file-sharing architectures were first developed 20 years ago and since then they have been significantly improved. First-generation architectures (e.g. Napster) are heavily centralized since their function depends on servers that store indices of the shared files [1, 2]. Although second generations (e.g. Gnutella) eliminate the need for centralized servers and provide connections only between the users [3–5], searching for a file takes time, especially if it does not exist in the peers the searching peer can reach. BitTorrent protocol [6], via the proposal of torrent files, removes the file searching process from the P2P network via introducing a centralized entity, called tracker, that stores metadata about the shared files. Trackers themselves do not have copies of the files, they only track the up/downloaders and make sure they can connect to each other. Users interested in a file, first locate its torrent in a torrent exchange website and then use a torrent client to find the peers who have stored the file locally.

The employment of trackers introduced various vulnerabilities that motivated the development of trackerless versions ([7, 8]) that employ distributed hash tables (DHTs) [9–11]. When DHT is enabled, it connects to a bootstrap peer and gets information about a set of DHT nodes and users to build up a small group of connected peers. Those peers are then used to get new peers. Although no tracker is required at any time, the users need to find the torrent files of the files they are interested in before using the DHT.

Blockchain-based systems, that appear after the development of Bitcoin [12] further decentralize file-sharing systems while a fraction of them enhances participants privacy too [13, 14].

In this work, we develop DIBDApp, a software library that integrates a Kademlia DHT [11] and utilizes Ethereum smart contracts [15] and the BitTorrent protocol [6] using the Java based Web3J tool [16]. In contrast to IPFS [17], that enables versioned and decentralized distribution of files, DIBDApp does not consider all files as part of the same generalized data structure. On the contrary, it employs smart contracts to minimize the storage requirements of each P2P node since they only need to store the files they are interested in exchanging with others.

Also, unlike existing P2P architectures that are assisted by the functions of blockchains [18, 19], the majority of the developed smart contracts implements read-only functions whose

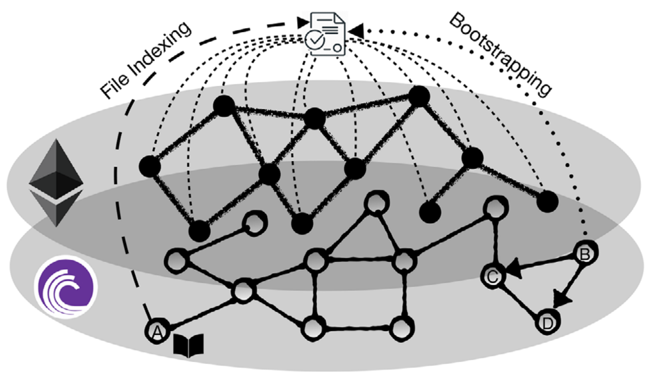


FIGURE 1 Via the DIBDApp library, an Ethereum smart contract assists node 'A' to index a file she wants to share with a P2P network while, at the same time, provides connectivity information to node 'B' in order to connect to nodes 'C' and 'D' and join the P2P network

execution via the Web3J library is near-instant and not limited by the consensus protocols of the Ethereum blockchain. Figure 1 depicts the cases where a peer wants to publish a file or a new peer wants to join the network and needs information to connect to other peers.

1.1 | Example

We consider Instagram, a traditional centralized online media sharing service, to illustrate how DIBDApp can be used to design a fully decentralized equivalent. We show how one can rely on DIBDApp to deploy such a service without having to host any centralized authority nor to administrate one, delegating all the tasks of centralization to the decentralized networks of Ethereum and of BitTorrent. We describe an Instagram service that provides simple features: (i) publish pictures with a description and tags, (ii) browse pictures by tags and authors, and (iii) follow other authors. Additionally, we discuss how DIBDApp can be employed in the implementation of a decentralized fake news detection mechanism [20] and in a decentralized federated learning protocol [21].

The motivation behind the design of DIBDApp is to reach a higher level of decentralization for online services. This goal can be questioned since losing centralization comes with trade-offs. One may think that removing any form of direct control from authority over a network will introduce a threat to the information made available by the network and bring in a potential risk of cybercrime victimization. Thus, the failure-resistance a decentralized network brings needs to be considered together with its censorship-resistance. Nevertheless, high decentralisation promotes democracy and freedom of speech [22].

1.2 | State-of-the-art

Before listing the contributions of this work, we discuss the two more closely related projects, IPFS and Filecoin, and compare them with DIBDApp.

The InterPlanetary File System (IPFS) is a peer-to-peer distributed file system that seeks to connect all computing devices with the same system of files [17]. DIBDApp differs from IPFS since it is built as a toolset to directly develop services and applications, implementing the building blocks of online applications in a decentralized way. IPFS is a protocol aiming to replace HTTP with a decentralized storing feature. DIBDApp's security and architecture also fundamentally relies on the Ethereum blockchain, where IPFS implements its own decentralized network. In IPFS, files must be tracked with a Git-like version control system, where files' data (but not metadata) are considered immutable in DIBDApp. Services built on top of IPFS (e.g. IPLS [23]) can be integrated with DIBDApp using the developed API.

Filecoin [24], Sia [25], Swarm [26] and Storj [27] are decentralized file storage system with similar functionality to DIBDApp but none of them provides file-sharing functionalities. Filecoin is a decentralized storage network that turns cloud storage into an algorithmic market. Filecoin is also blockchain-based but aims to host an actual market, where peers can rent hard drive storage to others, utilizing proof-of-replication storage and registering deals on the blockchain. The underlying file storing system is IPFS. Its purpose is different than DIBDApp's since it is a market in itself, while DIPDApp is a library, meant to be used as a technical tool in development.

1.3 | Contributions

Traditional backend services for online applications offer four main functionalities:(i) file reading, (ii) data indexing, (iii) file writing and (iv) data access policies. P2P desktop and mobile applications can employ DIBDApp for assisting nodes to join the P2P network (via the bootstrapping functions) and to find shared files without accessing any centralized entity (using the indexing functions). In detail, DIBDApp allows P2P applications to:

1) bootstrap (i.e. to connect with the other nodes in the P2P network),
2) upload the files they want to share,
3) browse existing files in the P2P network and,
4) download an existing file, by orchestrating interactions between Ethereum smart contracts and P2P nodes.

The rest of the paper is organized as follows: In Section 2 we discuss the required background before introducing the DIBDApp in Section 3; in Section 4 we present the implementation details of DIBDApp; in Section 6 the experimental results; in Section 8 we provide an overall discussion; finally, in Section 9 we present the concluding remarks.

2 | BACKGROUND

Before introducing the design of the DIBDApp library, the implementation details, representative use cases and discussing the performance, we introduced the required background for understanding the function of DIBDApp.

2.1 | Distributed hash tables

Hash tables are widely used in networked systems due to their ability to search, insert and delete data in constant time, on average. Distributed Hash Tables (DHTs) provide efficient ways to retrieve values associated with keys, similar to traditional hash tables. DHTs have the advantage to eliminate the need for a centralized authority that stores a complete hash table. DHTs are distributed in multiple nodes and every node can retrieve all the (key, value) pairs by communicating with the other peers. Kademlia [11] is a well-known and widely used protocol. The most popular uses are the BitTorrent protocol and the node discovery protocol in Ethereum network. Its wide acceptance is justified by the following features: (i) the number of configuration messages is minimized, (ii) it uses parallel and asynchronous queries to synchronize the nodes, (iii) it is not sensitive to node failures, (iv) the integrated lookup algorithm that finds files in other nodes is highly scalable.

2.2 | BitTorrent

BitTorrent protocol [6] is the most popular file-sharing protocol due to its ability to scale with the number of peers. Primary versions incorporate trackers, (i.e. online servers) to (i) facilitate the communication between peers by storing torrent files and metadata and (ii) to assist peers on searching the files they want to download. Popular trackers, such as "thepiratebay.org" or "demonoid.com" index hundreds of thousands of torrent files. These functionalities are based on the number of downloads or votes and let people download trusted files and increase users safety.

Another primary role of a tracker is to keep track of where file chunks are stored on the P2P network, and which peers can make them available for download. More than a decade ago, in 2008, BitTorrent started supporting trackerless torrent downloading using the Mainline DHT [28]. Mainline is an extension of Kademlia and is used by clients to find peers by mapping the hash of a torrent file to the list of peers that store the given file. In this way, a client can directly locate the peers who can share the file. Nevertheless, Mainline does not support file indexing which still requires a tracker to store torrent files. Also, in order to join the DHT, peers need to bootstrap via a known peer, which is often a tracker. Therefore, this solution only decentralizes a subset of the trackers' functions. Magnet links [29] allow clients to download files directly using the DHT without relying on torrent files. They store a set of parameters, including a torrent hash and other details like the specification of the file to download inside a torrent. This method avoids storing torrent files and as a result reduces the amount of data needed to download a file via the BitTorrent protocol.

2.3 | Ethereum

Blockchains [12, 30], one of the latest advances in permissionless distributed systems, are immutable and append-only distributed ledgers maintained by open distributed systems of Internet-connected computers which are incentivized to dedicate part of their resources to the network and can disconnect and reconnect at any time. Blockchains are composed of blocks of data that are stored in sequence. Each block contains data generated by an address, and information that is necessary for the operation of the blockchain. Although the most popular blockchain data type is transactions that transfer credit between addresses, blockchains have also been used to store other types of data [14, 31]. Transactions have also been used to store scripts, healthcare data, shared files, votes, ownership titles and others. In addition to storing data, blockchains also support decentralized applications, called DApps, that are composed of smart contracts and can process the stored data and protect sensitive information [14].

The limited scripting capabilities of the Bitcoin blockchain motivated the development of Ethereum in 2014 [15]. Ethereum, via its virtual machine (EVM), provides an open platform that allows anyone to build and use decentralized applications, like DIBDApp, that are composed of smart contracts. Each smart contract contains one or more methods and variables that define its state. After its creation, a smart contract acquires an address and its code cannot be changed. The methods of a smart contract can be called by users or other smart contracts that transfer a sufficient amount of gas on the contract's address in order to execute it. The gas amount depends on the type of code the method contains. Whenever the destination of a transaction is the address of a contract, the code of the called method is executed before adding the transaction in a new block and storing it in the blockchain. The methods included in smart contracts are categorized to "write" and "read-only". Write methods are called via a transaction to the contract which is broadcasted to the network, processed in exchange of gas and eventually added to a block in the blockchain. This process takes at least one block duration of time to confirm that the transaction has been validated. Read-only methods are named calls and read the state of the contract on the blockchain without any transaction broadcasting nor gas consumption and returns their return values instantly.

3 | DIBDAPP LIBRARY

DIBDApp exposes an API, as presented in Table 1, to P2P application developers who want to implement a decentralized backend. The API offers four methods: `upload`, `download`, `browse` and `bootstrap` that are listed in Table 1. In order to expose the required functionality without using any centralized entity, DIBDApp relies on two decentralized networks: the Ethereum blockchain network and a DHT-based BitTorrent network. For the implementation of the API, we design two classes to describe the users who participate in the P2P network

TABLE 1 P2P applications can use `bootstrap` to connect to the BitTorrent network and `upload`, `download` and `browse` to share, download and find files, respectively, by being assisted Ethereum smart contracts instead of centralized entities

Name	Parameters	Returns	Constraints	Description
`upload(User user, User author, File file, String name, List<String> keywords, String description)`	Author, file and keywords to index the file on the blockchain as well as a description and the user that calls the contract.	File ID that denotes its position on the blockchain if success, and null in case of failure.	The author needs a populated DHT	Adds a new file in the network and publish its metadata on the blockchain.
`download(User leecher, String magnet, File directory)`	The leecher to search in her DHT, a magnet URI to recover the file in the network, path to store the file	void, the file is downloaded in the gived location.	The user needs a populated DHT	Downloads a file from the network if it is available
`browse(User user, int startIndex, int windowSize, List<String> keywords, Address author)`	Keywords, id, and/or author to search + a window size and start index to select the number of answers + the user that calls the contract.	List<FileMetadata> object containing details of the matching files.	None	Searches for files indexed on the blockchain using keywords or author addresses.
`bootstrap(User user)`	User object with Ethereum wallet, IP, port number	True if the operation succeeds	Ethereum Wallet	A user joins the DHT by finding a bootstrap node on the blockchain.

(`User`) and the exchanged files (`FileMetadata`). The basic variables of these two classes are listed below.

```
class User(String name, DHT dht, Credentials ethereumWallet, int port, String ip)
class FileMetadata(String name, String magnet, String authorAddress, String description)
```

Furthermore, to complement the functions of the API, we deployed two smart contracts on the Ethereum blockchain: `BootstrapTracker` and `FileTracker`.

Considering the development of a decentralized Instagram-like application, an application developer can employ DIBDApp for the backend services. In detail, the process of publishing a picture and sharing it with other users can be handled via the `upload` function using the tags of the picture as keywords, its description and the picture itself as a file. The subscription of a user to another user can be implemented using the `upload` function with the subscription information as metadata without seeding an actual file. In order to allow the users of the service to browse pictures given an author or a tag, the `browse` and `download` function would make it possible to retrieve such files and then to display them after downloading them using BitTorrent. When they connect to the service, peers would join the network using the `bootstrap` function. Therefore, the whole service is relying entirely on purely decentralized architectures. Before listing the implementation details in the next section, we describe in more detail the indexing and bootstrapping processes.

3.1 | Indexing

In contrast to traditional backend services that store data on centralized servers, the data of a service developed using the DIBDApp library is "stored" in the BitTorrent network and located by peers using its DHT. The `download` function of DIBDApp is used to guarantee this feature. In order to call the `download` function, the user needs a magnet uniform resource identifier (URI) that references it such that the file can be recovered in the network. This leads to the other component of the "reading" function: indexing. DIBDApp implements the indexing functionality using the Ethereum blockchain. A Smart Contract called `FileTracker` plays the role of the "metadatabase" of the online service. `FileTracker` uses mappings to store file metadata (including magnet URIs) and indexes them with keywords. DIBDApp's `browse` function reads these data and combines multiple calls to `FileTracker` to answer the query in the most precise way. The file reading process is detailed in Figure 2. Whenever an application wants to share a file, the `upload` function is called to upload file metadata and files on the Ethereum and BitTorrent networks. This function allows to start seeding the file in the BitTorrent network, adds it to the DHT, publishes the related metadata on the `FileTracker` contract (some tool functions allow to edit the description or add a keyword to an already existing file). The uploading protocol is detailed in Figure 3.

3.2 | Bootstrapping

In order to join the BitTorrent network using its DHT without a tracker or a centralized bootstrap node, the DIBDApp library uses another Smart Contract called `BootstrapTracker`, which stores permanently updated information about a subset of users of the BitTorrent network to help new users to be bootstrapped in the network. They are other users of DIBDApp that previously called the bootstrap function. Indeed, once a user has been correctly linked to the network, it may be used to

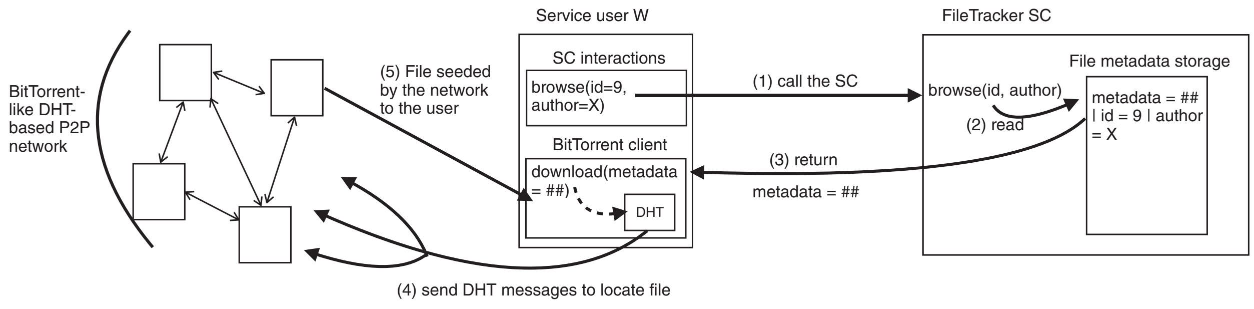


FIGURE 2 Search and download function of DIBDApp library. A user can use the browse function to search a file on the blockchain using its metadata, and use the DHT of BitTorrent network to find seeds that can share the wanted file

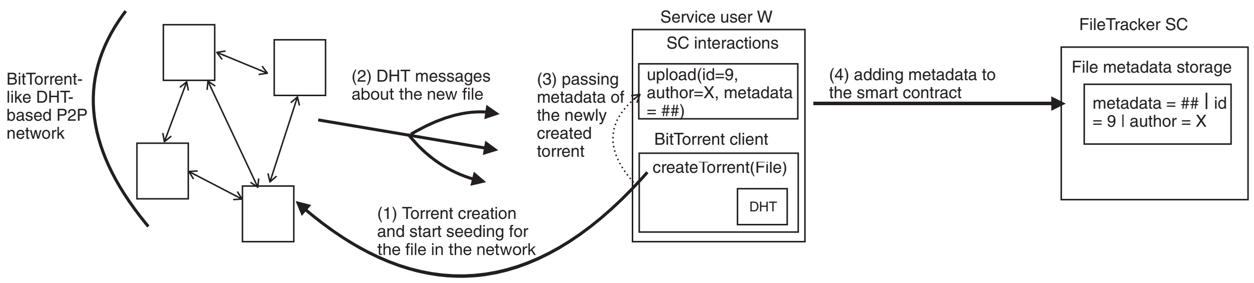

Service user W
SC interactions
upload(id=9, author=X, metadata = ##)
BitTorrent client
createTorrent(File)
DHT
FileTracker SC
File metadata storage
metadata = ## | id = 9 | author = X
BitTorrent-like DHT-based P2P network
(2) DHT messages about the new file
(3) passing metadata of the newly created torrent
(4) adding metadata to the smart contract
(1) Torrent creation and start seeding for the file in the network

FIGURE 3 Upload function of DIBDApp library. A user creates a torrent for a file she wants to share and start seeding it, making it available in the BitTorrent network. The file's metadata is then added on the blockchain to make the file public and accessible

bootstrap new ones. The bootstrapping protocol is detailed in Figure 4.

4 | IMPLEMENTATION

DIBDApp is composed of: (i) API Functions, (ii) a BitTorrent client, and (iii) a Blockchain DApp. The API functions allow developers to use the library. The BitTorrent client assists on file exchanging over the BitTorrent network using the BitTorrent protocol. The DApp on the Ethereum blockchain is used to store public information that is used by all users, that is, the bootstrap nodes, a file index, and file metadata.

4.1 | API functions

The four API functions, listed in Table 1, are the tip of the DIBDApp iceberg. They are the only part that a developer or user needs to know to use all the features of DIBDApp. We list below the implementation details of these functions.

`upload()`: The upload function adds the metadata of a file on the blockchain and makes the file available in the BitTorrent DHT. In addition to `upload()` DIBDApp exposes two secondary functions called `editDescription()` and `addKeyword()` which are available for the owner of a file that can use a file ID to change the description of the file or add a keyword.

`download()`: The download function calls the BitTorrent client and downloads the file from the BitTorrent network after locating it using the DHT by giving a magnet URI (which plays the role of an identifier for the file on the DHT). The user needs to have called the `bootstrap` function before in order to have a working DHT that can search for files over the network. She also need to have called the `browse` function to find the file she wants to download by looking at its metadata to get the file's magnet URI.

`browse()`: The browse function calls the `FileTracker` smart contract browsing function depending of the given arguments. It will return a list of file metadata that match with the arguments given as parameters. File metadata is composed of a name, a magnet URI, the Ethereum address of the account that uploaded the file, and a file description. The parameters of the browse function is a list of keywords but the `FileTracker` contract data structures are mappings from one keyword to a set of files. Therefore sorting and filtering of the indices is done by the Java code by computing the set intersection with the results of multiple smart contract calls (as shown in Algorithm 1).

`bootstrap()`: The bootstrap function calls the `get_node()` function of the `BootstrapTracker` smart contract to gain access to the information of some already-in-the-network users. It will then use this information to join the network and populate the DHT of the BitTorrent client. Then, it will register itself to the smart contract as a bootstrap node using the implemented `join` function.

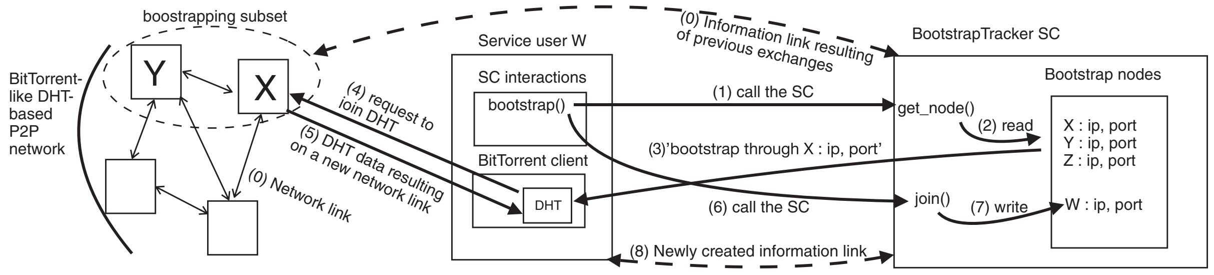


FIGURE 4 Bootstrap function of DIBDApp library. A user asks for bootstrap nodes on the blockchain calling getNode function, and populates her BitTorrent DHT using one of them. She finally becomes herself a bootstrap node by calling the join() function

ALGORITHM 1 Pseudocode of file browsing on the Java side

```
1: procedure BROWSE($user, start, window, id, author, keywords[]$)
2:     create a list $result$
3:     $contract \leftarrow loadFileTracker(user)$
4:     // Handle a query using file id.
5:     if $id \neq null$ then
6:         $result[0] \leftarrow getMetadata(contract, id)$
7:     else
8:         create the sets $aIds$, $kIds$, and $resIds$
9:         // Handle a query using file author.
10:        if $author \neq null$ then
11:            $aIds \leftarrow getIds(contract, author)$
12:        end if
13:        // Handle a query using keywords.
14:        if $keywords.length \neq 0$ then
15:            $kIds \leftarrow getIds(contract, keywords[0])$
16:            for $i \leftarrow 2$ until $keywords.length$ do
17:                $kIds \leftarrow kIds \cap getIds(contract, k)$
18:            end for
19:        end if
20:        // Merge results.
21:        $resIds \leftarrow aIds \cap kIds$
22:        for $j \leftarrow start$ until $min(resIds.size, start + window)$ do
23:            $result[start - j] \leftarrow getMetadata(contract, resIds[j])$
24:        end for
25:    end if
26: end procedure
```

4.2 | BitTorrent client

BitTorrent network has proved in the last decade to be an efficient way to share files in a decentralized manner. It is also a well-known option to share effectively large files. That is why it became the most popular peer-to-peer file-sharing protocol.

The BitTorrent clients integrated with DIBDApp provide tools for two functions in the library: the `download` and `upload` fucntions. In order to do so, we used the two most famous Java BitTorrent clients: bt [32] and tTorent-lib [33].

4.2.1 | Bootstrapping

To use the BitTorrent protocol, we need to populate the node's DHT with other peers to find the files we want to

ALGORITHM 2 Pseudocode of bootstrapping

```
1: procedure BOOTSTRAPPING($user$)
2:     $contract \leftarrow loadBootstrapTracker(user)$
3:     $windowSize \leftarrow getWindow(contract)$
4:     // For each node on the bootstrap list, try to get her DHT.
5:     $i \leftarrow 0$
6:     while $user.getDHT() = null$ and $i < windowSize$ do
7:         $ip, port \leftarrow getNode(contract, i)$
8:         $user.bootstrap(Node(ip, port))$
9:         $i \leftarrow i + 1$
10:    end while
11:    if $i = windowSize$ then
12:        print("Bootstrap failed, try again later")
13:    else
14:        // Become itself a bootstrap node.
15:        $join(contract, user.getIpPort())$
16:    end if
17: end procedure
```

download. Once it is done, by communicating with other peers, we will be able to update our DHT and have a better search tool for files in the BitTorrent network. To do so, multiple calls to the `BootstrapTracker` Ethereum smart contract are done in order to provide a list of bootstrap nodes to the BitTorrent client. These nodes will share their DHT to populate the new user's one with the shared content. This will help her to have a base on which she will be able to update the content using the Kademlia node lookups. For each returned node, the bootstrap function of the mainline DHT is called until she gets a populated DHT. This process is presented in Algorithm 2. Once the new user has a populated DHT, she will call again the `BootstrapTracker` smart contract using the `join` function. By doing so, she will share her IP and port on the blockchain and become herself a bootstrap node.

4.2.2 | Downloading

Downloading a file using DIBDApp is very similar to downloading a file using a traditional BitTorrent client (e.g. Vuze, uTorrent, and others). The only difference comes from the way you find the file, the tools you need to find it as well as finding the peers that can share it with you. When a user wants to download a file, she first looks for it using the `FileTracker` smart contract and via `browse` function. Once she finds the file she wants, she queries the DHT using the magnet URI from the file's metadata. This URI plays the role of the key in the DHT and will link to peers that can share the file. Locating the nodes is done in the same way as when updating the DHT: it searches for the closest nodes for the provided key (magnet URI). Those nodes have, by definition of distance in the Kademlia DHT, more information about who owns the searched file. Therefore, to make this request, the user needs to have her DHT populated. Once the seeds are located, the user can start downloading the file from them. The bt client is used for the download function.

4.2.3 | Uploading

Uploading a file using DIBDApp is a two-sided process between the BitTorrent client-side and the Ethereum blockchain. For the BitTorrent side, we need to add a new entry to the DHT. This entry will contain a mapping between the file torrent hash (contained in a magnet URI), and the nodes that own and seed the file. Therefore, initially, a torrent file is created from the user's file using the tTorrent library. Some metadata such as the creation date or the author's name are added to the torrent file. Once this torrent is generated, the hash is added to the DHT linked to the user's node by calling the Kademlia (or mainline DHT) `STORE` function. After doing this, DIBDApp will add the file's metadata to the `FileTracker` smart contract using the `upload` function to share it to everybody.

4.3 | Ethereum DApp

Blockchain technologies provide a solution to the problem of storing public information and sharing it with everybody without relying on centralized authorities. By being freely accessible, robust and censorship-resistant, it is a great candidate to replace a central tracker in peer to peer interactions and file indexing. To store the public data of DIBDApp, we used two Ethereum smart contracts called `BootstrapTracker` and `FileTracker` that form the DIBDApp decentralized application (dApp) and operate as a decentralized tracker. The `BootstrapTracker` smart contract manages the user's connection to the BitTorrent network by providing the bootstrap nodes needed to join the DHT network. The `FileTracker` smart contract manages the indexing of uploaded files by storing metadata to find the information necessary to download the file using BitTorrent. It, therefore, assists DIBDApp users to share and download files.

The role of the `BootstrapTracker` smart contract is to help a new user to join the BitTorrent DHT network. To do so, it stores a list of online users that can share the DHT called to bootstrap nodes. Practically, the `BootstrapTracker` smart contract, combined with the BitTorrent mainline DHT, implements the connectivity functionalities offered by BitTorrent centralized trackers or simply replaces DHT bootstrap nodes. The goal of this smart contract is to return to the client a list of the peers that will assist her on bootstrapping (Algorithm 2). This functionality is implemented in two functions

TABLE 2 `BootstrapTracker` functions

Name	Description
`join(string _node)`	Adds a user that already has a populated DHT to the list of nodes. In order to join the list, the FIFO replacement policy is applied: when a user joins the bootstrap nodes list, she replaces the oldest node on the list.
`getNode(uint256 _index)`	Returns the node at the position of the given index on the list.

that are described in Table 2. The list of nodes stored in the smart contract is of a fixed size. The nodes are defined by their IP and the port number used for the DHT. Finally, the bootstrap nodes replacement policy is first in first out (FIFO) and each new online user becomes a bootstrap node until she is replaced by another. In this way, the time a disconnected node is still in the bootstrap list is minimized. There is a tradeoff between the size of the list of the bootstrap nodes and the gas needed. The higher the size of the list, the higher the gas needed to store the IP and the port number of the nodes. However, the higher the number of the nodes on the list, the higher the probability of finding at least one that is online and can assist a new user with her bootstrapping. Assuming N nodes in the list with node i to be offline with probability p_i, the probability of all of them to be offline is $f = \Pi_{i=1}^{N} p_i$. For $N = 10$ and $p_1 = p_2 = \cdots = p_{10} = 0.5$, $f = 1/2^{10} < 1\%$.

The role of the `FileTracker` contract is to help users find a file previously uploaded to the network. To do so, it stores metadata about uploaded files. Then, when someone is searching for a file, this information is used to recover it. The two parts of `FileTracker` are Uploading and Browsing. The function that implements the core functionality for uploading a file is the `upload` function. It allows a user to add a new file, or more precisely its metadata, to the smart contract storage. These metadata are composed of the magnet URI of the files, a list of keywords, the author's address, a file name, a file ID and the description of the file. The smart contract never stores the actual data of a file because this would be extremely costly. This smart contract only helps the user to find the files previously uploaded on the network without storing this information on a centralized server and therefore plays the file indexing role of a BitTorrent tracker. The `upload` function uses the SHA-256 hash function and a mapping between the hash of each keyword and the ID of the file. It uses another mapping between the address of the author and the ID of the file. Therefore, all the files corresponding with an author or a keyword can be further retrieved. It is also possible to retrieve a file using its ID. All this metadata being pushed to a list when files are uploaded. Two more functions called `addKeyword` and `editDescription` complete the uploading functionality. The `addKeyword` function allows the author of a file to add a new keyword to her file if she wants to help people to find it more easily. It will, therefore, add the file ID to the hash of the new keyword in the aforementioned keyword-file ID mapping. The `editDescription` function is used in the same way but to modify the description initially passed to the `upload` function. Those functions are gas-consuming and are called via transactions by the DIBDApp client.

The Browsing part has a lot of functions that read the data stored on the smart contracts. Therefore they are used through blockchain "calls", with near-instant answers and no gas cost. These functions are using the previously described mappings and list to provide the information users need. They allow getting the name, the keywords, the description, the author or the magnet URI corresponding to a file using its ID. It also provides the file IDs corresponding to a keyword or an author helping users to search for a file. The only crucial information to download a file using its metadata is the magnet URI that is used by the BitTorrent client to search the file in the DHT. All other information is here to index the file or provide further details about it.

5 | DESIGNING A DECENTRALIZED INSTAGRAM

The developer of a decentralized Instagram needs to define the format of the strings that are used as keywords when calling the upload function. One of the ways would be to pass the tags as keywords in a format "tag=mocha" (DIBDApp includes an upload function that automates this process by taking a JSON object as a parameter instead of a list of keywords). This will allow the developer to include other ways of browsing files, such as the location of a file, by uploading files with keywords of the form "location=Mocha". (Using directly "mocha" as a keyword for the initial tag feature would make it impossible to later differentiate the pictures taken in Mocha, Yemen and a Starbucks picture with the tag #mocha.) In this way, updating the decentralized Instagram to include videos would also be easy, adding the keyword "type=video".

Multiple new features can be added and a centralized database can be imitated by following this format. Users can be authenticated using their Ethereum addresses. When a user publishes a picture, its metadata is uploaded on the `FileTracker` contract, and the picture becomes searchable by its tags and by its author. The developer then implements a feature that finds pictures with a tag (or an author) given by its user (via `browse`) and displays them on its screen after downloading them (via `download`). When a user follows Giannis Antetokounmpo, the upload function is called with the keyword "type=follow" and "followed=giannis_an34" as a description but no magnet URI is passed because this information does not index an actual file stored on the BitTorrent network. When the user clicks on its personal feed, the browse function with "type=follow" as a keyword and the user herself as an author is called, and all the descriptions are unwrapped to look for the last pictures of the followed users. The application developer can, in the same way, allow users to rename themselves. A bijection between the author and metadata describing its username could be uploaded to display it/allow it to be searched by its username instead of the Ethereum address. The functions `addKeyword`

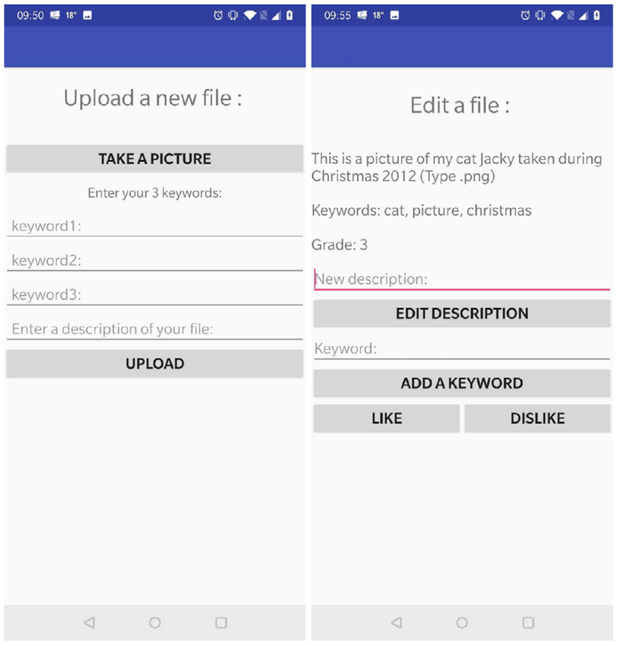


FIGURE 5 Example of a decentralized Android application that is using DIBDapp and allows users to upload and edit their photos while sharing them with their followers

and `editDescription` can also be used to allow the users to edit their publications and modify their pictures tags.

Figure 5 shows two example Android activities implemented to test DIBDapp on Instagram-"like" functions. Via the left activity, the user can take a picture, add a few keywords, enter a description of the picture and upload it. Via the right activity, a user can add more keywords and edit the previous description. Due to space limitations, we do not present more activities. In conclusion, the user can publish pictures with tags and descriptions, look for pictures given specific tags and authors and follow the publications from chosen authors, without relying on a central authority at any point. The user has full control over its content and cannot be censored without actually updating the code of the DIBDApp library and forcing other users to download the new code. Nevertheless, the developer can keep some control over its service. For example, it can customize the front page of it by displaying a file she is the author of, which means this file cannot be overwritten and she has total control over it.

6 | PERFORMANCE EVALUATION

Blockchain-based systems introduce several characteristics that need to be considered. First of all, adding a transaction to the blockchain takes time and, therefore, when writing on the blockchain, we need to evaluate the time needed to validate the operation. A transaction can take a multiple of a block's duration to be mined. Additionally, we need to evaluate the gas cost introduced by transactions to smart contracts. This evaluation is important for the `upload` and `bootstrap` functions that need to write data on the blockchain. Some functions of DIBDApp

TABLE 3 Gas needs and cost of deployment of the two smart contracts

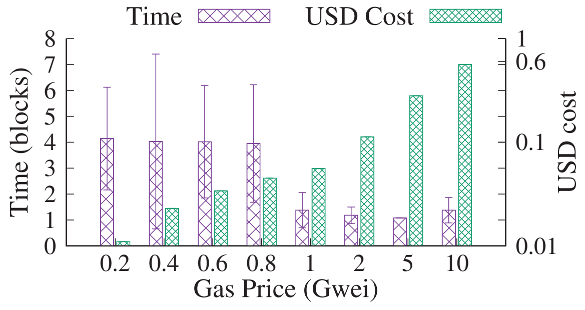
Name	Gas needs	Cost in USD (gas price: 0.05 GWei)
`BootstrapTracker`	325,003	0.0038350354
`FileTracker`	1,150,979	0.0135815522

also rely on the blockchain without using transactions but only reading data. When the volume gets high, fetching can take time, which is measured below.

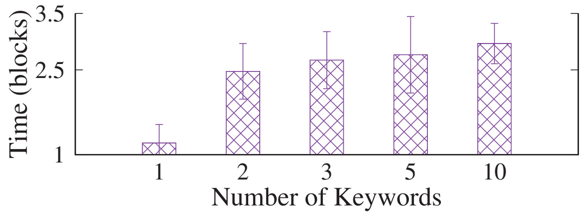
Set up: We evaluate the performance of DIBDApp by introducing code snippets collecting data in the Java code of the DIBDApp client. We use the Remix IDE [34] to develop the DIBDApp DApp with Solidity programming language [35]. We test the DIBDApp DApp by deploying the developed smart contracts in the Rinkeby Ethereum test network [36]. This network is used by developers on the Ethereum blockchain to test their applications without needing to use the main network and pay for the gas needs of their contracts. The collecting snippets and Etherscan [37] allow us to know the time transactions take and the gas they consume. We repeated each experiment ten times, and in every plot, we present the mean measured value and the standard deviation. Ethereum identifiers are created using the Ethereum wallet [38], an open-source software maintained by the Ethereum community used to send transactions, deploy smart contracts, create wallets and other. The wallets were funded with testnet Ethers using the Rinkeby Authenticated Faucet [39]. We considered two sets of gas prices (in Gwei, 1 ETH $= 10^9$ Gwei) during the experiments: "high" = [0.2, 0.4, 0.6, 0.8, 1.0, 2.0, 5, 10] and "low" = [0.05, 0.1, 0.15, 0.2]. Before examining the gas needs of each function, we measure the deployment cost of the developed smart contracts and show the results in Table 3. We use the cost of Ether on 24 May 2019 to calculate the deployment cost in USD.

The processing of an Ethereum transaction consumes gas, and gas is not free. The price the user is willing to pay for the gas works as an incentive and impacts the time needed for a transaction to be processed and added to the blockchain. Therefore, the time that the program takes to execute a transaction of the Ethereum blockchain depends on the price paid by the user. Based on that, we design one set of experiments to measure the time of execution for all transactions as a function of the gas price paid to perform them. It is important to evaluate the time needed to write data on the blockchain because this process dominates the executing time of DIBDApp. Since these times are aligned with the blocks in which the produced transactions will be added, we measure them in numbers of blocks. The average block time in the Ethereum blockchain is 15 s.

Reading data from the blockchain does not require any transaction. It is therefore not gas-consuming and independent of the block time. Thus, testing the reading time of every call to the smart contract is not necessary. However, when the number of calls is massive, for example, to retrieve a very large number of files, it may be interesting to see how the metadata-fetching time evolves with the number of instances.



(a) Different Gas prices.



(b) Different number of keywords.

FIGURE 6 Gas and time needs of the `upload` function

Actions associated with BitTorrent and its DHT are not measured as the protocols have already been extensively evaluated and are considered efficient and reliable [11]. Since the `download` function is purely BitTorrent-based, we do not present its performance in this section.

6.1 | Indexing performance

The performance of the indexing capabilities of DIBDApp depends on the performance of `upload` and `browse`.

6.1.1 | Upload function

This function relies on transactions since it writes data on chain. It is used to find the average time it takes to upload a new file on the network. Adding the file on the BitTorrent network only means registering it in the DHT and seeding it to the network. It is irrelevant to measure such actions. The time taken to add metadata on the blockchain is the actual time-dominating procedure of this feature. The only variable that can vary a lot when uploading a file is the number of keywords associated with it. The time for 1, 2, 3, 5, and 10 keywords is measured for gas price = 1.0 Gwei. The time to upload a one-keyword file is also measured against gas price. We see in Figure 6a that for a time-requirement of the order of one block, 1.0 Gwei is a correct gas price for the `upload` function with one keyword. It has to be noted that `upload` can be expensive, with transaction costs of the order of ten cents. Figure 6b shows that beyond 2 keywords, the execution time does not increase significantly.

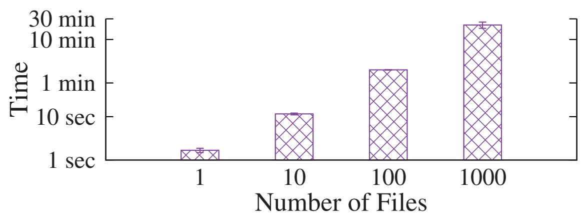


FIGURE 7 Needs of `browse` for different number of files

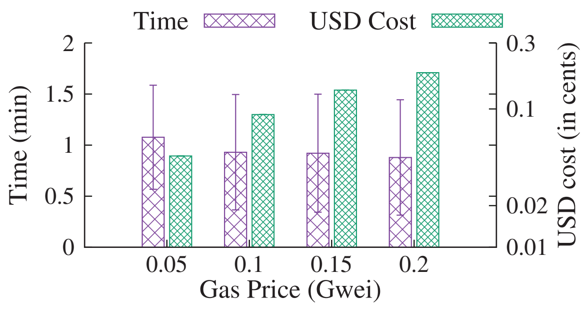


FIGURE 8 Time needs and USD costs for the smart contract join function (in bootstrap API function)

6.1.2 | Browse function

The browse function returns file metadata corresponding to passed parameters. It is a read-only function, however, when the files that match the parameters are numerous and the window size is large enough, fetching the metadata for each of them can be time-consuming (but still gas-free). The time it takes to fetch file metadata with the browse function is measured against the number of matching files as powers of ten (1, 10, 100, 1000). We observe in Figure 7 that the time to fetch metadata increases linearly with the number of files that match the lookup criteria.

6.2 | Bootstrapping performance

The bootstrapping feature of our system interacts with the blockchain in two different actions. First, it queries the bootstrap nodes, doing multiple calls until someone populates the DHT. Then, once one of these nodes has been used to be bootstrapped in the DHT, the user registers itself as a bootstrap node in the smart contract. The user does not have to wait for this interaction to be completed and the time it takes is therefore partially irrelevant (this action is pure network altruism so its completion can take a lot of time without any consequences on user experience). This time can be arbitrarily decreased to cut the costs of the execution. A plot of the time of execution for very small gas prices and the associated price of execution in USD is shown in Figure 8. It can be seen on the figure that for a waiting time of only around 1 minute, the cost of the `join` function is of the order of the hundredth of cent of USD. Costs can, therefore, be heavily reduced. It is important to understand that this minute of waiting time is not the time the user waits for the network but conversely: it is the time the network waits for the user to declare itself as a bootstrap node. The network is not impacted by this time being long because it does not wait for only one node but for any node that has been bootstrapped, which makes irrelevant the time that separates their bootstrapping from their self-declaration.

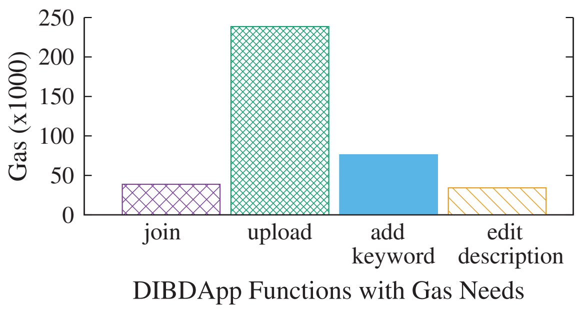


FIGURE 9 Gas consumption of DIBDApp transaction-based operations on smart contracts

6.3 | Overall gas consumed and actual cost

After measuring the time needed for each of the aforementioned transaction-based and read-only functions, we measure the gas needed for the transaction-based function and present them in Figure 9. Functions with minimal changes in the states of the smart contracts (e.g. the `bootstrap` function that relies on the simple `join()` function of the `BootstrapTracker` smart contract) have low requirements. On the other hand, functions that store information (e.g. the `upload` function that writes on the `FileTracker` smart contract) consume much gas. Gas needs multiplied by the gas price shows how much Ether needs to be paid.

6.4 | Lessons learned

6.4.1 | Joining the network

Based on Figure 8, we can say that it is very cheap to join the network, which means the "most altruistic" action of a node using DIBDApp is designed in a way that its costs are as low as possible. Also, necessary calls to join the network through bootstrapping are read-only `get_node`, which means it is free of charge and really quick to be bootstrapped in the network of a DIBDApp-developed service.

6.4.2 | Keywords

Adding a keyword to a file requires gas and given that it has to be called for every keyword of a file, this may be an issue. This comes from the fact than Solidity is not able to receive a variable-size array of variable size types yet. It may be fixed in the future. However, the design of DIBDApp succeeds to avoid the heavy tradeoff this could be regarding execution-time.

Figure 6b shows that the time of adding keywords remains of the same order for several keywords.

6.4.3 | Files

The uploading performance depends on the time requirements the developer has. If it needs to be performed quickly, that is, within one block, it can be costly as Figure 9 shows. Uploading consumes a lot of gas and a gas price of more than 1.0 Gwei could be needed to achieve the aforementioned time-requirement. Figure 6a shows the USD cost that the `upload` function can reach for too high gas prices when quickness is expected by the caller.

6.4.4 | Search

The browsing time requirements are linear to the number of the browsed files, as depicted by Figure 7. This can be an issue if the volume of files is massive, thus the parameters startIndex and windowSize of the browse function allow the caller to regulate the amount of data to be fetched.

7 | USE CASES

In addition to the example of decentralized Instagram, we discuss two more use cases with very different foci, fake news and federated learning and explain why their decentralized versions do not require trust on any authority.

7.1 | Fake news

The continuously increasing rate of Internet penetration, which is associated with the proliferation of devices with Internet connectivity, empowers anyone to disseminate information, using social media platforms, that can reach a potentially huge audience. However, this comes with a disadvantage since anyone can try to spread fallacious information and influence others to further share this information without being sure about their credibility.

Fake news, that is, news articles that are intentionally and verifiably false, can affect the public's perception. For example, C. Silverman and J. Singer-Vine in 2016 [40] show that "fake news headlines fool American adults about 75% of the time." It goes without saying that the type of information impacts the complexity of verifying their truthfulness significantly.

DIBDApp can be trivially applied in a decentralized framework that allows anyone who has evidence regarding the trustworthiness of a news article or any published information, in general. More specifically, in the scenario where someone identifies an article as "fake" the considered framework can create a keyword and invite anyone interested to contribute on clarifying whether the article is fake or not. Similarly, in the case of a published photo that has been processed and altered before publication, DIBDApp can be trivially used to invite anyone with contradicting or supporting evidence.

The motivation behind implementing a decentralized fake news detection framework is based on that fact that it will not be controllable by anyone and as a result, the objectiveness of the produced rulings will depend only on the submitted evidence and not in the interests of the entity that maintain the framework.

7.2 | Federated learning

The performance, usually in terms of accuracy, of machine learning models, depends on the data that are used during the production of the models. A process that is known as model training and requires access to all the employed data. Federated learning is a recently proposed model training method that does not need to access to the data and enables entities that want to collectively train a machine learning model to train it without giving access to their data to each other. The process is orchestrated by a centralized entity. In short, via federated learning, multiple data owners can collaborate using a predetermined protocol that works iteratively and requires only the exchange or the model parameters between each participant and a centralized server and does not require sending the actual data [21].

DIBDApp can be used for the exchange of the model parameters and eliminate the need for a centralized entity. The motivation behind employing DIBDApp is to incentivize more entities to participate since they will not have to trust a centralized entity.

8 | DISCUSSION

In this section, we discuss the inherent characteristics of DIBDApp (e.g. privacy) as well as design decisions (e.g. integration of reputation mechanisms) and potential optimizations (e.g. to reduce the gas costs) that can be part of the next version of DIBDApp.

8.1 | Privacy on DIBDApp

DIBDApp's bootstrap function maintains a list of bootstrap nodes on the blockchain following a full replacement policy, thus writing IP addresses of every member of the network on the public immutable ledger. This privacy tradeoff (similar to the fact that IPs are easily monitored on DHT-BitTorrent [41]) can be solved with a more sophisticated implementation of the smart contract involving asymmetric cryptography, but it would increase the number of transactions for a node to bootstrap. Another way to diminish this trade-off is by implementing a replacement policy for the bootstrap nodes based on volunteering, where the community of the network would be the distributed nodes allowing new peers to be bootstrapped. This still leads to an incentives issue where registering as a bootstrap node costs money (but is cheap as Figure 8 shows) and resources to answer the bootstrapping queries. The functionality of the DIBDApp DApp does not allow users to delete the metadata of a file even if the file is no longer available for sharing. The `FileTracker` smart contract does not have a delete metadata function. Even if it had, the previous states of the smart contract are available on the old blocks of the blockchain and contain the previous versions of the metadata. A signed flag protocol could be employed to propagate to other peers of the off-chain network the request for not seeding nor referring a file anymore, but nothing can oblige all pears to conform. Developers can make some efforts to allow authors to lower the visibility of their files, but the decentralized character of the network prevents to ensure full deletion. This lack of deletion reflects the ambiguity of GDPR [42] for decentralized architectures. The right of erasure of GDPR (Article 17) cannot be satisfied by the developer of a DIBDApp service. On the other side, the developer is not fully in charge of her service because she does not administrate its running version.

8.2 | Mixing DIBDApp with centralized entities

In DIBDApp, uploading metadata has a cost. A service that has frequent changes of states writes frequently on the blockchain and increases its monetary cost. DIBDApp can be considered to be mixed with a centralized server to design services that are long-term decentralized but short-term centralized to avoid excessive transactions. The state of the back-end could be cached on a centralized server for the time of one "round" and periodically pushed on-chain as a batch. In the case where the centralized server would become unavailable, the service would start to rely on the decentralized backup. On the long run, the service does not have a permanent single point of failure and cannot be censored.

8.3 | Transaction fees of DIBDApp

DIBDApp faces the problem of assigning the responsibility of paying the transaction fees when a user uses a feature that results in an upload on the `FileTracker` contract. When the developer decides to charge the users, she forces them to have means of payment (i.e. Ethereum wallets). The natural way of paying these transactions is that the user triggers it herself, that is, her Ethereum identifiers sign the transaction. However, if the purpose of the online service has nothing to do with Ethereum or cryptocurrencies, it may be tedious to require its users to have Ethereum identifiers. Therefore, the developer can build a payment management layer between transaction triggering and user interaction, where the users pay a specific entity with a specific mean of payment, the funds are used to fill an Ethereum account that is used to sign the transactions in the code. In this way, the amount paid by the user can be a customized function of how much they cost in reality.

8.4 | Ethereum and BitTorrent reliance

The performance of DIBDApp relies on the assumption than DHT-based BitTorrent and the Ethereum blockchain are secure and efficient. BitTorrent, even though considered robust and powerful, has been shown as vulnerable [43] and perfectible [44]. It is clear that many ways exist to improve BitTorrent (IPv6 [45] for NAT issues [46], seeding incentives [47] etc.). The Ethereum network can also suffer from congestion and has several points of potential improvements [48, 49]. Regarding scalability, DIBDApp does not add any additional requirement over those of BitTorrent and Ethereum. Thanks to DIBDApp's simple and efficient implementation, the performance of a DIBDApp service does not decrease with the number of users (given the scalability of Ethereum and BitTorrent).

8.5 | Reputation mechanisms and incentives for DIBDApp

The DIBDApp client offers the users the functionality of rating files via a transaction to the DIBDApp DApp. The rating function is simple and can be called by any online peer. Considering that the DIBDApp DApp does not keep track of the users who downloaded each file, this may lead to undesired behavior by malicious users. A more sophisticated function that is based on the rating history of each user and defines a reputation score to each of them can characterize the accuracy of their judgments and therefore regulate their impact in the file rate [50, 51], even if the users are anonymous [52]. File ratings are useful to the DIBDApp client since it can set up a threshold under which it stops to seed them. Similarly, a reputation system for the off-chain network can assist the DIBDApp clients to choose from whom to download or get the populated DHT. Apart from reputation mechanisms, incentives can also be developed to demotivate bad behaviour. Proof-of-burn mechanisms [53], for example, can automatically withdraw ETH from users accounts for punishment and deposit ETH for their serviceableness.

9 | CONCLUSION AND FUTURE WORK

We implemented the DIBDApp library as a Java API relying on Ethereum smart contracts and the BitTorrent protocol. Additionally, we used Solidity programming language to develop the DIBDApp DApp that is composed of two smart contracts and is deployed in the Rinkeby testnet. The contribution of the DIBDApp is threefold: (i) the process designed for users of a DHT-based P2P network to join it eliminates the need for centralized bootstrap nodes. (ii) The functions designed to upload and download files store metadata on a decentralized smart contract and abolish the need for a centralized index of those files. (iii) The combination of the indexing and the bootstrapping, associated with a DHT-based BitTorrent network, makes it possible to design online services of various forms without relying on any centralized authority.

The future development of DIBDApp will be in two directions, one towards the development of more sophisticated and optimized indexing to satisfy more demanding criteria of an online service back-end. The second direction is towards the decentralisation of existing popular applications, with Instagram to be the first target.

ORCID

Dimitris Chatzopoulos https://orcid.org/0000-0002-4765-5085
Pan Hui https://orcid.org/0000-0001-6026-1083

REFERENCES

1. Cunningham, B.M., et al.: Peer-to-peer file sharing communities. Inf. Econ. and Policy 16(2), 197–213 (2004)
2. McCourt, T., Burkart, P.: When creators, corporations and consumers collide: Napster and the development of on-line music distribution. Media Cult. Soc. 25(3), 333–350 (2003)
3. Chawathe, Y., et al.: Making gnutella-like p2p systems scalable. In: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 407–418. ACM, New York (2003)
4. Ripeanu, M.: Peer-to-peer architecture case study: Gnutella network. In: IEEE Peer-to-Peer Computing, pp. 99–100. IEEE, Piscataway (2001)
5. Matei, R., et al.: Mapping the Gnutella network. IEEE Internet Comput. 6(1), 50–57 (2002)
6. Cohen, B.: The bittorrent protocol specification (2008)
7. Fry, C.P., Reiter, M.K.: Really truly trackerless bittorrent. School of Computer Science, Carnegie Mellon University, Tech. Rep. 06–148 (2006)
8. Taddia, C., Mazzini, G.: A multicast-anycast based protocol for trackerless bittorrent. In: IEEE SoftCOM 2008, pp. 264–268. IEEE, Piscataway (2008)
9. Stoica, I., et al.: Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM CCR. 31(4), 149–160 (2001)
10. Lua, E.K., et al.: A survey and comparison of peer-to-peer overlay network schemes. IEEE Commun. Surv. Tutorials 7(2), 72–93 (2005)
11. Maymounkov, P., Mazieres, D.: A peer-to-peer information system based on the XOR metric. In: International Workshop on Peer-to-Peer Systems, pp. 53–65. Springer, Berlin Heidelberg (2002)
12. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009)
13. Chatzopoulos, D., et al.: Privacy preserving and cost optimal mobile crowdsensing using smart contracts on blockchain. In: 2018 IEEE 15th International Conference on Mobile Ad Hoc and Sensor Systems (MASS), pp. 442–450. IEEE, Piscataway (2018)
14. Zyskind, G., et al.: Decentralizing privacy: Using blockchain to protect personal data. In: 2015 IEEE Security and Privacy Workshops, pp. 180–184. IEEE, Piscataway (2015)
15. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper 151, pp. 1–32 (2014)
16. blk.io. Where java meets the blockchain, https://github.com/web3j/web3j, 2018. Accessed 25 September 2020
17. Benet, J.: Ipfs-content addressed, versioned, p2p file system. arXiv preprint arXiv:1407.3561, 2014
18. Wilkinson, S., et al.: Metadisk a blockchain-based decentralized file storage application. Storj Labs Inc., Technical Report, hal. pp. 1–11 (2014)
19. Chen, Y., et al.: An improved p2p file system scheme based on ipfs and blockchain. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 2652–2657. IEEE, Piscataway (2017)
20. Edson, C.T., Jr, et al.: Defining "fake news" a typology of scholarly definitions. Digit. Journal. 6(2), 137–153 (2018)
21. Yang, Q., et al.: Federated machine learning: Concept and applications. ACM Trans. Intell. Syst. Technol. (TIST) 10(2), 1–19 (2019)
22. Thede, N.: Decentralization, democracy and human rights: A human rights-based analysis of the impact of local democratic reforms on development. J. Human Develop. Capabilities 10(1), 103–123 (2009)
23. Pappas, C., et al.: IPLS: A framework for decentralized federated learning, 2021
24. Filecoin Community: Filecoin: A cryptocurrency operated file storage network, 2014
25. Nebulous Inc.: Sia: Simple decentralized storage. Accessed: 14 Oct 2014
26. Tron, V., et al.: Swarm: a decentralised peer-to-peer network for messaging andstorage. Technical report, Ethersphere (2019)
27. Wilkinson, S., et al.: Storj: A peer-to-peer cloud storage network. (2014)
28. Loewenstern, A., Norberg, A.: BitTorrent DHT Protocol (2018)
29. Houlihan, T.: BitTorrent Magnet URI Extension (2017)
30. Swan, M.: Blockchain: Blueprint for a new economy. O'Reilly Media, Inc. (2015)
31. Conoscenti, M., et al.: Blockchain for the internet of things: A systematic literature review. In: 2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA), pp. 1–6. IEEE, Piscataway (2016)
32. Tomashpolskiy, A.: Bt, http://atomashpolskiy.github.io/bt/, (2019). Accessed 25 September 2020
33. JetBrains.: ttorrent-lib, https://github.com/jetbrains/ttorrent-lib, (2018). Accessed 25 September 2020
34. Remix IDE. https://remix.ethereum.org Accessed 25 September 2020
35. Ethereum. Solididty. https://solidity.readthedocs.io Accessed 25 September 2020
36. Rinkeby Ethereum Testnet. https://www.rinkeby.io Accessed 25 September 2020
37. Etherscan: The Ethereum Block Explorer. https://etherscan.io/ Accessed 25 September 2020
38. The Ethereum Wallet. https://wallet.ethereum.org/ Accessed 25 September 2020
39. Rinkeby Authenticated Faucet. https://faucet.rinkeby.io/ Accessed 25 September 2020
40. Silverman, C., Singer-Vine, J.: Most americans who see fake news believe it, new survey says. BuzzFeed News 6 (2016)
41. Wolchok, S., Halderman, J.A.: Crawling bittorrent DHTs for fun and profit (2010)
42. European Parliament and Council of the European Union. General Data Protection Regulation (2016)
43. Wang, L., Kangasharju, J.: Real-world sybil attacks in bittorrent mainline dht. In: IEEE GLOBECOM, pp. 826–832. IEEE, Piscataway (2012)
44. Fan, B., et al.: The design trade-offs of bittorrent-like file sharing protocols. IEEE/ACM Trans. Netw. 17(2), 365–376 (2009)
45. Xinxing, Z., et al.: A measurement study on mainline DHT and magnet link. In: IEEE First International Conference on Data Science in Cyberspace (DSC), pp. 11–19. IEEE, Piscataway (2016)
46. Fan, B., et al.: Stochastic analysis and file availability enhancement for bt-like file sharing systems. In: IEEE International Workshop on Quality of Service, pp. 30–39. IEEE, Piscataway (2006)
47. Pouwelse, J., et al.: The bittorrent p2p file-sharing system: Measurements and analysis. In: International Workshop on Peer-to-Peer Systems, pp. 205–216. Springer, Berlin Heidelberg (2005)
48. Chen, S., et al.: A comparative testing on performance of blockchain and relational database: foundation for applying smart technology into current business systems. In: Distributed, Ambient and Pervasive Interactions: Understanding Humans: Proceedings (Part I) of the 6th International Conference (DAPI 2018). Lecture Notes in Computer Science LNCS, vol. 10921, pp. 21–34. Springer, Cham (2018)
49. Kim, S.K., et al.: Measuring ethereum network peers. In: Proceedings of the Internet Measurement Conference 2018, IMC '18, pp. 91–104. ACM, New York (2018)
50. Marti, S., Garcia-Molina, H.: Taxonomy of trust: Categorizing p2p reputation systems. Comput. Networks 50(4), 472–484 (2006)
51. Dingledine, R., et al.: Reputation in p2p anonymity systems. In: Workshop on Economics of Peer-to-Peer Systems, vol. 92. ACM, New York (2003)
52. Marti, S., Garcia-Molina, H.: Identity crisis: anonymity vs reputation in p2p systems. In: Proceedings of the Third International Conference on Peer-to-Peer Computing, 2003 (P2P 2003), pp. 134–141. IEEE, Piscataway (2003)
53. Stewart, I.: Proof of burn (2012)

How to cite this article: Schutz P, Gal S, Chatzopoulos D, Hui P. Decentralizing indexing and bootstrapping for online applications. IET Blockchain. 2021;1:3–15. https://doi.org/10.1049/blc2.12001