DISTRIBUTED RESOURCE SHARING
USING THE BLOCKCHAIN TECHNOLOGY ETHEREUM

A Project

Presented to the faculty of the Department of Computer Science
California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in
Computer Science

by

Danny Murray

FALL
2016
DISTRIBUTED RESOURCE SHARING
USING THE BLOCKCHAIN TECHNOLOGY ETHEREUM

A Project

by

Danny Murray

Approved by:

__________________________, Committee Chair
Dr. Jinsong Ouyang

__________________________, Second Reader
Dr. Jun Dai

__________________________
Date
Student:  **Danny Murray**

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

__________________________________________, Graduate Coordinator

Dr. Ying Jin

Date

Department of Computer Science
Modern public cloud storage architectures, such as those offered by Amazon or Google, are hosted across multiple datacenters and when demand increases enough, datacenter storage capacity needs to be upgraded to meet it. Contrast this with the vast amount of storage that goes unused on consumer drives running on personal computers. While consumers' drive storage sits idle, doing nothing for them, those same consumers create the demand for cloud storage that fuels more datacenter growth. Furthermore, though these cloud storage services have their data decentralized, the providers of these services are themselves a central point of control. Though they may be reliable for data storage and retrieval, a user must also trust them to keep their data secure and not to sell or abuse it in ways not consented to. Further still, if a powerful government agency wants to gather data, the central control of these services makes them easy targets, and a user cannot reasonably expect the providers to deny the demands of their government.

This project aims to show, using a simple prototype, the possibility, and perhaps desirability, of a distributed alternative to centrally controlled cloud storage services that utilizes the unused space on consumer drives. Using a client application to interact with smart contracts built to run on the distributed computing platform Ethereum, a user can manage their data and interactions with other users. It allows users to sell their own disk space to other users for a fair
price. Periodic payments help incentivize these users to maintain the data they host, while data transfer payments help incentivize them to maintain a high availability. Those users buying disk space can buy from any number of users, backing their data up to all of them. Utilizing more users for redundant data backup will increase data availability and decrease the chances of data loss. This exchange creates a kind of peer-to-peer cloud for making use of idle storage.

Though the model laid out by this prototype is relatively simple and not likely to yield any improvements in performance, pricing, or availability compared to traditional cloud architectures, it still serves as a demonstration of the feasibility of the idea of a peer-to-peer cloud for storage sharing that is managed by Ethereum or a similar distributed application platform. In addition to solving the problems discussed previously, a cloud storage system like this, with a well thought out design for data management and user incentives, may be able to attain metrics that are comparable to the cloud services we see today.

_______________________, Committee Chair
Dr. Jinsong Ouyang

_______________________
Date
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>List of Tables</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
</tbody>
</table>

## Chapter

1. **INTRODUCTION**
   - 1.1 Problem .......................... 1
   - 1.2 Purpose And Scope ............ 2

2. **BACKGROUND** ..................... 4
   - 2.1 The Sharing Economy ............ 4
   - 2.2 Bitcoin And The Blockchain .... 5
   - 2.3 Ethereum .......................... 8

3. **DRS REQUIREMENTS** .............. 13
   - 3.1 Overview .......................... 13
   - 3.2 Account Management ............ 14
   - 3.3 Virtual Drive Creation And Management .......................... 15
   - 3.4 Directory Creation ................ 16
   - 3.5 Uploading Files .................. 16
   - 3.6 Downloading Files ............... 16

4. **DRS DESIGN** ....................... 18
   - 4.1 System Architecture .............. 18
   - 4.2 User Interface .................... 22
   - 4.3 Ethereum Smart Contract Design ........................................ 25
<table>
<thead>
<tr>
<th>Tables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 DRS Transaction Costs</td>
<td>53</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>7</td>
</tr>
<tr>
<td>4.1</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>24</td>
</tr>
<tr>
<td>4.5</td>
<td>25</td>
</tr>
<tr>
<td>4.6</td>
<td>28</td>
</tr>
<tr>
<td>4.7</td>
<td>31</td>
</tr>
<tr>
<td>4.8</td>
<td>33</td>
</tr>
<tr>
<td>4.9</td>
<td>34</td>
</tr>
<tr>
<td>4.10</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>42</td>
</tr>
<tr>
<td>5.5</td>
<td>43</td>
</tr>
<tr>
<td>5.6</td>
<td>44</td>
</tr>
<tr>
<td>5.7</td>
<td>44</td>
</tr>
<tr>
<td>5.8</td>
<td>45</td>
</tr>
</tbody>
</table>
5.9  Virtual Drive Content Management Controls......................................................... 46
5.10 Directory Creation Dialog.................................................................................. 46
5.11 Directory Creation Contract Code......................................................................... 47
5.12 Download Location Selection Dialog...................................................................... 48
5.13 File Upload Initiation Client Code.......................................................................... 49
5.14 File Upload Contract Code.................................................................................... 50
5.15 File Download REST Call Client Code................................................................. 51
5.16 File Request Completion Contract Code............................................................... 52
Chapter 1

INTRODUCTION

Distributed Resource Sharing (DRS) is a prototype of a peer-to-peer cloud storage system that allows users to buy storage space directly from other users and to sell their own unused space to other users. By using the distributed application platform Ethereum to connect users and manage their interactions, this system allows a user to create value from previously unused storage as well as giving them access to reasonably priced cloud storage that isn't subject to the pitfalls of centralized cloud solutions.

1.1 Problem

There exists, on most computers connected to the internet, a large amount of unused resources such as unused hard disk space and idle CPUs. No value is created by these resources for consumers while it remains idle. In contrast, many companies offering cloud services, such as Amazon and Google, offer their own resources at a price. These services are provided by many large datacenters that store vast amounts of data. That only get larger as demand increases, thereby creating more resources under their control and leaving the previously mentioned unused resources untouched.

These cloud services are popular because they offer convenient storage backup that can be accessed anywhere as well as the ability to increase computing resources as needed. However, they come with some drawbacks. These services require trust on the part of the user that the service provider will not sell or share their data with other companies or governments. Having data stored and maintained by one central authority requires a lot of faith that a user's data will be given the respect it deserves. Additionally, the prices for such services can be quite expensive.
because as previously mentioned, the resources they are offering are built exclusively for the purpose of selling them in this way.

With these issues in mind, alternatives are being developed that may eventually replace the cloud systems we know today [1]. A new blockchain technology, Ethereum, is steadily gaining popularity as a platform for distributed applications. With it, many new ways of computing are being conceived. It allows for decentralization in a way never before possible, which makes it an ideal platform for developing an alternative to the majority of cloud architecture we have today.

1.2 Purpose And Scope

This project, Distributed Resource Sharing, aims to show that, using the Ethereum blockchain, a different type of cloud storage is possible. A cloud system that utilizes the idle storage resources of consumer hard drives and offers a more decentralized and distributed way of storing data that is outside of any central control and could potentially be cheaper as well. By using Ethereum as a middle man between users, the data can be accessed from anywhere and no central authority needs to be trusted. Ethereum keeps track and what data is stored and where, giving the information needed to access it from anywhere.

The scope of this project intends to be quite narrow, as providing a fully functional cloud system is an enormous task. The core requirement is to create a prototype of simple cloud storage system where a user requesting cloud data storage has the request fulfilled by other users that provide the storage, all being mediated by interactions with the Ethereum blockchain. No attempt at sharing any resource other that hard disk space is made (such as sharing CPU power, which would be a difficult project on its own). Additionally, as this is a prototype and not intended for
real use, no attempt is made at providing encryption of stored data, though this would clearly be necessary to protect the user's data. Also, no attempt is made at optimization as the purpose is only to show that this kind of system can work. Finally, as this is a prototype, the user interface is limited only to allow interaction that shows the basic functionality.

This report will explain the project in detail, starting with the background information needed to understand it. Following that, the requirements, design, and implementation of the project are covered. Last is an evaluation of the results and followed by the conclusion and a discussion of future work for the project.
Chapter 2

BACKGROUND

This chapter aims to give the reader some background information on the inspirations, concepts, and technologies behind this project. Distributed Resource Sharing utilizes the concept of peer-to-peer networking to build an alternative model of cloud storage. It was inspired by and expands on the concept of the Sharing Economy, extending it into the digital realm. It is made possible with Ethereum (an evolution of the blockchain technology first used by Bitcoin), which allows it to connect peers without relying on centralized systems. Understanding all of these components, particularly Ethereum, is essential to understanding the project.

2.1 The Sharing Economy

The Sharing Economy was a strong inspiration for this project. It is the concept of people sharing their resources with each other instead of all purchasing their own, which is often more than they individually need (for example, think of kitchen or cleaning appliances which rarely see any use). This kind of resource sharing could be drastically cheaper and leave far fewer resources wasted. While it would be nice if these systems could rely on people to share their things with strangers for free, that's not how the majority of them work. Instead they'll make a market for a resource by connect users interesting in buying or selling that resource, which makes the “sharing” aspect of this seem like a bit of a misnomer. However, the sharing may be thought of as a reference, “to the use and access of shared physical or human resources or assets, rather than the fact that there is no monetary exchange” [2]. Regardless, this project uses it in the same way, with hard drive space being bought and sold between users, not given away for free.

Many popular examples exist such as Uber, Lyft, Airbnb, RelayRides, and SnapGoods,
all providing a way for people to share their physical goods [3]. However, there haven't been any
devoted to digital resources such as hard drive space or CPU power until some very recent
examples which are discussed in the conclusion of the paper. One of the main motivations of this
project is to extend the Sharing Economy to digital resources.

2.2 Bitcoin And The Blockchain

Bitcoin has attained enormous popularity in recent years and most people are at least
familiar with the name and essential concept, though there is much less familiarity with the
underlying blockchain technology that powers it [4]. Bitcoin is a trustless digital currency system
that uses a peer-to-peer network running a proof-of-work algorithm to maintain consensus on the
current state of the blockchain, which is a public digital ledger that records all Bitcoin
transactions (a transaction being the transfer of Bitcoin from one set of accounts to another). It is
significant in that all these features create a secure, reliable digital currency that doesn't rely on a
trusted third party and is outside of any central control. Bitcoin is the precursor to Ethereum,
which is the backbone of this project, and understanding it and the blockchain concept that
powers it will help with understanding this project. First the currency transaction system of
Bitcoin will be described, as it has relevance to the more complex transactions in Ethereum,
followed by a description of the blockchain, which is core to Bitcoin and Ethereum.
In the original paper on Bitcoin, released in 2008 by Satoshi Nakamoto, an electronic coin is defined as a chain of digital signatures [5]. Ownership is transferred when the owner creates a new transaction by signing the hash of both the new owner's public key and the previous transaction, after which this new transaction is added to the chain as shown in the simplified chain of ownership in Figure 2.1. Not shown there is that transactions can include multiple inputs, all of which must be digitally signed, as well as multiple outputs. This way, coins don't have to be handled individually (which would be akin to only using the smallest unit of a currency such as pennies), but instead can be combined and split as needed in single transactions.

With this coin system of transaction chains it is possible for a recipient to validate the chain of ownership, but they will be unable to prove that the coin hasn't been spent already. Currency systems using this same method had been around well before Bitcoin and all had run into the same problem [6]. The breakthrough for Bitcoin was the development of a new concept that allowed it to not need to rely on a trusted third party like most of the cryptocurrencies that
preceded it. This new concept was the blockchain, a public digital ledger that is managed by a peer-to-peer network to maintain consensus on the current state of the system. With this consensus it makes it impossible to double spend a coin because everyone on the network, via the block algorithm they run, has agreed on an exact sequence of transactions that determines the current state of coin ownership.

![Figure 2.2: A Simple Blockchain Diagram](image)

It is called the blockchain because at regular intervals a “block” is added to the existing chain of blocks, with a block being essentially a sequence of the most recent transactions. The “block time” for Bitcoin averages 10 minutes, meaning the state is changed that often by the addition of a new block. Running the transactions in the blockchain from beginning to end from an initial state (all accounts having zero Bitcoin) will result in the current state of the system. Without getting into too much detail, the decision of which transactions and in which order are included in the next block is made by a “proof-of-work” algorithm run by the so called “miners” running the peer-to-peer network. These miners receive transactions from Bitcoin users and use them to “mine” the next block. They are rewarded for it with Bitcoin, which is the incentive that maintains a large peer-to-peer network. To mine the next block, miners take the hash of the previous block's hash, all the transactions it has received in sequence, and an integer called a
nonce. If the value of the hash is below a certain threshold (which determines the difficulty and thus the average block time) then the mine was successful, otherwise the nonce is increased and the hash is done again. The first miner to achieve this is rewarded and will broadcast the result to the other miners who will validate it, run the transactions in it to get the current state, and then begin work on the next block. Though this algorithm is essentially just busy work at its core, it makes the system secure because the proof-of-work aspect means that an attacker can't just spawn an army of virtual machines to take control. They would need to control more than half of the computational power of the network to control the state of the blockchain. Additionally, it serves to establish a digital currency with no third party oversight by using this busy work to bring about a consensus on the system state.

2.3 Ethereum

Ethereum is a distributed computing platform that utilizes the blockchain technology first used by Bitcoin, but extends the usefulness of it well beyond cryptocurrencies. It was first proposed in late 2013 in a whitepaper by cryptocurrency researcher Vitalik Buterin, where he states the intention of Ethereum being to provide, “a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create "contracts" that can be used to encode arbitrary state transition functions” [7]. These features give it the power to run as the backbone for new types of truly distributed applications, such as the one developed for this project and many others.

Ethereum's blockchain, though much more advanced, is still run on generally the same principles as Bitcoin's. It still uses a proof-of-work algorithm run by a distributed network of peers to find consensus on the current state of the system, with peers being rewarded by
cryptocurrency (Ethereum's is called “ether”). That network receives transactions from users and the algorithm periodically determines which sequence of those transactions are included in the next block in the blockchain. New blocks determine how the state is altered and sequentially running all transactions in the blocks of the blockchain from initial starting conditions will give the current state of the system. Though the block time for Ethereum currently averages around 14 seconds while Bitcoin is about 10 minutes, both operate on the same core ideas of the blockchain.

Where Ethereum differs drastically is in the complexity of both the state stored by the blockchain and the ways transactions can modify that state. The state in Ethereum is made up of objects called “accounts” which are located by their 20 byte address [7]. There are two different kinds of accounts, externally owned accounts (EOA) and contract accounts (CA). EOAs are controlled by private keys and like a simple Bitcoin account, they have a field to store the account's current ether balance. CAs have a balance as well, but also additional fields for contract code and storage. CAs are controlled by their contract code which determines how they interact with other accounts and how they access and modify their storage, which is a key/value store for data that is persistent between transactions.

EOAs, but not CAs, can create transactions to be included in the blockchain. A transaction includes the address of the recipient account, a signature to identify the sending account, and an amount of ether to transfer from sender to recipient. That's all that's needed to transfer value, like Bitcoin transactions, to either type of account. However, there is an optional data field as well, and it can be used to call any public function in the contract code of a recipient CA. This gives transactions much more powerful way to induce state changes because of the Turing-completeness of the contract code. Though the contracts cannot initiate their own transactions, they are able to send “messages” to other contracts. Messages are almost the same as transactions, but are only initiated by CAs through their contract code, not by EOAs. Like
transactions, they include the message sender, the message recipient, the ether amount to transfer, and an optional data field to call a function in the recipient's contract code.

Calculating the state changes made by these transactions is done by the Ethereum Virtual Machine (EVM), which run the low level contract bytecode. One unique thing about how the EVM operates to achieve this is the concept of “gas,” which can be thought of as fuel purchased to execute a transaction. While Bitcoin does allow scripting for more complicated transactions, they are not Turing-complete like Ethereum code is. The problem Bitcoin had that Ethereum found a solution to is the issue of resolving transactions that involve loops. If a transaction had an infinite loop the state would be impossible to calculate and even if calculable, the computational time could still be enormous, which would cause huge problems for the blockchain. Ethereum's fairly simple solution is to make every operation a transaction executes have a cost associated with it. When creating a transaction, the amount of gas units the transaction will start with are purchased with ether in the sender's account at an arbitrary price, typically whatever the market price is currently (the price chosen can effect how quickly the miners running Ethereum's blockchain will include the transaction, if at all, with higher prices creating a higher incentive). Every bytecode operation on the EVM costs an integer amount of gas, with operations to modify or add to contract storage being by far the most expensive to because all those changes are stored on the blockchain forever. As a transaction executes, the gas it started with is depleted until the transaction completes or the gas runs out. If it completes, the ether cost of the remaining gas is refunded back to the transaction sender. However, if it runs out of gas no ether is returned and the transaction fails. This necessary system works to incentivize application code that is efficient both in terms of calculation and storage use, but is none-the-less an added burden to writing high quality Ethereum applications as discussed in the chapter on the project evaluation.

Contracts are normally written in high level smart contract languages that compile down
to the bytecode run by the EVM and make contract writing simpler. Solidity, which was used for this project, is the smart contract language of choice for Ethereum, with the most support, object-oriented features, and syntax similar to JavaScript [8]. Writing a contract in it is very similar to writing a class in other languages, with functions, member variables, and interactions with other contracts. A very unique feature of these contracts is their permanence and the immutability of their code. If a developer doesn't include code to allow their developer address to have special privileges (by only allowing access to a function if the message sender address equals the “owner” address that is stored by the contract when created), then they will have no more control over the contract than anyone else, meaning it can't be deleted and will run on the blockchain permanently. Even if a developer gives themselves special control in the contract code, they still have no ability to change the code itself once the contract is deployed (though of course there are certain workarounds that allow upgrading). This may seem like a burden rather than a feature, but there are a couple things to consider. They are called contracts for a good reason and just as a written contract shouldn't be modified after it was signed, Ethereum contract code is like a digital contract specifying, with the logical certainty of code, the exact behavior of the contract account, which can be seen as a kind of autonomous entity on the blockchain. This is ideal for some current and possible future applications that would benefit from a completely trustless model, where not even the developer can manipulate it once it is deployed, especially if the application involved money. One interesting Ethereum application to take advantage of this was “The DAO” (decentralized autonomous organization) which was the first DAO for Ethereum. It was defined by smart contracts that users could invest in, make proposals to, cast investment-weighted votes on those proposals, receive investment-weighted rewards when successful proposals pay off, and more [9]. It initially was wildly successful, earning $150 million from user investments, but was hacked in mid 2016, losing about $50 million of those investments due to a flaw in their code.
[10]. So just like with a written contract, no matter how well intentioned, if it is written poorly it can be exploited.

For this project, Distributed Resource Sharing, Ethereum was the perfect choice to be the backbone of this peer-to-peer cloud storage sharing system. With it, smart contracts written for it can store and manage user account data, storage metadata, and requests between users, with the transactions to interact with the contracts being created by project's client application. Ethereum makes these contracts accessible anywhere, always available, and only under the control of the user that owns them. It does this without any central control, thanks to its advanced use of the blockchain technology first used by Bitcoin.
Chapter 3

DRS REQUIREMENTS

3.1 Overview

The Distributed Resource Sharing (DRS) application's purpose is to provide a user with a way to sell their unused hard drive space and to backup their own data by paying for storage space on the unused space of one or more other users. That data should be able to be accessed from any computer running the DRS client once the user has logged in with their Ethereum account info. The DRS data stored on the Ethereum blockchain will be limited to basic user information, virtual drive listings, and metadata on each virtual drive's contents. The content metadata will include the drive's directory structure, file and directory names, file sizes, and file hashes (for validation of downloaded data). The contents of the files are not stored on the blockchain (which would be prohibitively expensive), but on the drives of other users who can be contacted directly as needed to add, modify, or retrieve drive contents. Throughout this paper, the terms “requester” and “provider” will be used in place of “user” when there is a need to clarify the roles being played by users. “Requester” is used when describing a user that is requesting to purchase new storage space or requesting access or modification to existing storage space. “Provider(s)” is used when describing one or more of the users fulfilling those requests by providing their own local storage space. This terminology is used in a similar way to how client and server are used in centralized architectures, but it is more apt for a peer-to-peer network.

To store data, a requester must first create a virtual drive with one or more providers backing up the drive. The data on any drive, when logged in through the DRS client, can be viewed and modified by adding or deleting new files at any time. No guarantee is made that the data will be available, though increasing the number of providers backing up the requester's data
can help increase the likelihood of obtaining the data.

To incentivize users to sell their disk space, pricing contracts are created during the creation of a new virtual drive between the requester and each provider. This immutable contract specifies the cost per byte per time period that the provider charges for hosting data. Additionally, it contains the price per byte transferred which determines the price for each byte uploaded or downloaded. This simple pricing helps to ensure that providers are incentivized to maintain a requester's data in the long term and to be online and available as much as possible to collect data transfer fees.

The user interface for DRS will be simple, offering enough to utilize the core functionalities the DRS application provides. It must do the following:

- Provide an interface to sign in/out using an Ethereum key.
- Provide an interface for the creation of virtual drives and display information about them.
- Provide an interface for creation of named directories within a drive and provide a virtual drive contents table that allows navigation of a drive's directory structure. Additionally the present working directory should be displayed when navigating a drive.
- Provide a means to upload any local file to be stored on a virtual drive and display that file's metadata listed in the virtual drive contents table.
- Provide the means to download any file from a virtual drive and store it locally.

### 3.2 Account Management

Users must be able to sign in/out from the DRS client using an Ethereum account key (which is an Elliptic Curve Cryptography key). If they are a first time user, a DRS user account
will be created in Ethereum, which will store their account information as well as information on any drives they create in the future. Once logged in, users will see displayed their Ethereum account address and the amount ether (Ethereum's currency) owned by that address. Additionally, any virtual drives owned by the account will be displayed.

To use DRS, the account must have a reasonable amount of ether to afford the contract creation and other transactions sent to Ethereum that is necessary for proper functioning of DRS. These transactions vary largely in cost, from pennies to dollars, with contract creation being the most expensive. Generally, the more expensive transactions used by DRS are more infrequent.

3.3 Virtual Drive Creation And Management

Once logged in, a user can create a virtual drive. The user must enter a name, a size, and the number of providers to host the data. The providers are selected from the pool of DRS users. A data storage pricing contract is made between the user and each provider, specifying the costs of hosting and transferring data to and from this provider. Once all providers have accepted the contract terms, the drive is created and may be viewed and modified by the user.

For the scope of this project the users requested to act as providers are selected arbitrarily and arbitrary pricing values are used for the data storage pricing contracts. However, in a real world application it would be ideal to have providers selected by highest availability, transfer speeds, and user ratings. For the data storage pricing values, it would be best to have current market rates be used for a default pricing which can then be modified by the user. For now, arbitrary values will work just as well to show what is possible with DRS.

A virtual drive table will display basic information (name, size, and space available) on all drives owned by the current user. When a drive is selected, a separate virtual drive contents table will display that drive's contents and from there the user can explore the drive contents,
create directories, upload files, and download files.

### 3.4 Directory Creation

When a drive has been selected a user can create a directory by supplying a name and the directory will be placed in the current directory being viewed in the drive contents table. Directories don't take up any space in the virtual drive and allow for better organization of the user's data.

### 3.5 Uploading Files

To upload files, a drive must be selected and the file's desired location in the virtual drive should be navigated to. Then the user will select upload and pick a file from the local computer to be uploaded. Data about the file, such as its location, name, size, and hash, are stored on Ethereum in the virtual drive's contract storage. A file request is created for each provider hosting the drive's data and the data storage pricing contract associated with each provider is given an amount of ether that is determined by the file's size in bytes multiplied by the cost per byte transferred (specified in the data storage pricing contract). The requester's DRS client contacts each provider directly (with a web service call using the IP address stored in their DRS user accounts). When the provider receives the upload request via a web service call, it will find the file request on Ethereum and accept it, which will release half of the funds stored by the request to the provider. The file is then stored locally by the provider, the requester is notified of completion, and the file request is marked as completed by the requester, which releases the remaining half of the funds to the provider. Now the file should be available for download when logged in as this user from any computer using the DRS client.

### 3.6 Downloading Files

To download a file, a user will double click any file displayed in the virtual drive contents
table. They will be asked to select a directory on the local drive in which to store the file once it is retrieved. Similarly to the file upload process, a file request is created for each provider and funds are allocated for this request based on file size and the data storage pricing contracts. However, instead of contacting every provider, the requester only iterates through them until one of them fulfills the request. The file request to the provider that successfully returned the file to the requester is marked as complete and all funds are released to that provider. All other funds are returned to the requester by retracting the unneeded file requests. The file data will now be accessible locally where it was stored on the requester's drive.
Chapter 4

DRS DESIGN

4.1 System Architecture

Figure 4.1: A Basic DRS System Architecture Diagram

The basic architecture of the Distributed Resource Sharing application is shown in Figure 4.1, focusing on the highest level components. All interactions between a requester (a user buying data storage through DRS) and providers (users selling their data storage through DRS) that host the requester's data are recorded and mediated through smart contracts written for DRS that run on the Ethereum network. For example, when a requester makes a request to upload or download a file, a transaction is sent to store a record of that request for each relevant provider in the contract storage on the Ethereum blockchain. Additionally, some ether is sent with those
transactions to pay for transfer fees. Uploading a file requires the file's metadata (specifically the name, size, and SHA3 hash) to be stored in the virtual drive contract. The requester can then query the contract storage to get the IP addresses of the providers and contact them directly through a web service call, notifying them of the request. The providers will then find the relevant file request, validate it, mark it as accepted to collect half the fee, and then complete the request. If a download was requested, the requester can then validate the data returned by checking its hash against the hash stored in Ethereum. Now the request is complete and the requester marks it as such, which pays out the remaining half of the fee to the provider.

Given that the users of DRS may occupy the role of requester and/or provider depending on whether they are buying storage, selling storage, or both, it seemed prudent to make one application serve both purposes. This way, whether a requester, a provider, or both, a user only has to worry about one application to manage their data and the data they store for other users. This simplifies using the system not only for users, for development of the project as well.
Figure 4.2: DRS Component Interactions
Figure 4.2 shows the main components of the DRS system and how they interact. The DRS Manager application is run on a user's machine and can operate as both requester and provider. Two kinds of network communication are used (shown in Figure 4.2 by red arrows), one for interactions with the Ethereum network and another to interact with peers for data transfer. Those interactions are described below and the details of each internal component, for both the DRS Manager and the Ethereum smart contracts, are described in the following sections.

To interact with the DRS smart contracts on the Ethereum network, an Ethereum client is used to handle the low level interactions such as finding peers running Ethereum nodes, synchronizing blockchain data with those nodes, sending transactions to the network, and retrieving new blocks as they are mined. Internally, initialization of the Ethereum client and interactions with it are mediated through an interface to simplify the creation of transactions, viewing contract data from the blockchain, and getting Ethereum event logs. Another component of the Ethereum interface provides internal components with a way to register as listeners for Ethereum events that are logged by DRS smart contracts, which allows them to be responsive to new changes that occur on the blockchain. Finally, because transactions to create or interact with a DRS contract all follow the same pattern but using different inputs, wrapper classes are used for interfacing with each contract. This way the contracts can be treated as just another local object, with all Ethereum interactions being abstracted away.

The other network interaction is the web service calls that are made from a requester to providers when requesting a file upload or download. During these interactions, both the requester and the providers are also communicating with the contracts on Ethereum to gather data to validate and complete the request, as well as modifying data to record the file request status, which also involves paying the transfer fee to the provider's Ethereum account. For both upload and download web service calls, the parameters passed include the information needed to validate
that the requester is who they say they are, that the file request exists and the file's metadata can be found, and that the hash found in the file metadata matches the hash of the requested file (the one uploaded or retrieved for download).

4.2 User Interface

![User Interface Mockup For The DRS Manager Application](image)

Figure 4.3: User Interface Mockup For The DRS Manager Application

The essential components of the user interface for the DRS Manager application are
shown in Figure 4.3. The user can log in or out from the session menu at the top left. Upon login, the user's Ethereum account ID is displayed as well as any Ether associated with that account. The virtual drives owned by the user, if any, are displayed in the virtual drives table in the middle of the window, which gives the name, size, and space available of each drive. When a virtual drive is selected, the directory contents of the root directory of that drive are displayed in the virtual drive contents table, displayed at the bottom of the window. The drive name and the current directory path being viewed are displayed above the table, as well as buttons for drive navigation, directory creation, and file upload. Each directory content row in the table shows the content's type (directory or file), name, size, and date modified. When a directory's table entry is double clicked, the contents of that directory are opened and displayed. The back arrow above the table can be used to go to the parent directory. When a file's table entry is double clicked a dialog is opened for the user to select a local directory to download the file to, after which the file is retrieved from the providers hosting the data and saved to the chosen location. To create a new directory the user clicks the directory icon above the virtual drive contents table, enters the name of the directory, and selects create, after which it will appear in the table. To upload a file to the drive location being viewed in the virtual drive contents table, the upload icon is clicked and a dialog will request a local file be selected. The file is then uploaded and will appear in the virtual drive contents table. Additionally, the display of the relevant drive's space available is decreased.

When the plus icon above the virtual drive table is selected, the virtual drive creation dialog is displayed and from there the user can select the drive's size and the number of providers to host the data. After clicking create, the user will then be able to view and modify the new table from the virtual drive table and the virtual drive contents table.
The class design to implement the user interface is a simple view hierarchy, shown in Figure 4.4. The main components are the two tables views, one for viewing and modifying the user's virtual drives and the other for viewing, navigating, and modifying the contents of the virtual drive selected in the virtual drive table. These table views and the tables themselves implement several listener interfaces to respond to events from other UI components or from events from an Ethereum contract. Additionally, actions and data input from the user trigger communication with the management interfaces to allow the user to sign in, sign out, create drives, navigate drives, create directories, and to upload and download files.
4.3 Ethereum Smart Contract Design

The smart contracts written for DRS are what tie the system together and make it work. They are the heart of the DRS model. Though they're not storing any of the bulk file data, the model they represent and the transactions defining how that model is changed are how this system of cloud storage can work in a peer-to-peer manner, without central control.

![Diagram of DRS Smart Contract Design](image)

Figure 4.5: DRS Smart Contract Design
The layout of the DRS contracts is shown in Figure 4.5. The root of the system is the DistributedResourceSharing contract, which is created only once and which all users reference. It stores and manages a list of all users, mapping Ethereum account addresses to a DRSUser contract. New users register their account address with the contract and are given a new DRSUser contract that they alone control. This store of users is later used to find candidate providers when creating new virtual drives.

The DRSUser contract stores and manages a user's preferences and storage metadata, for both their requester and provider roles. It contains the IP address at which this user can be reached to retrieve data it is hosting. It also contains a list of a DRSDataStorageContract contracts that the user is a part of, so the user can maintain what data they are currently hosting for other users. Last, it store a list of virtual drives that the user owns as the requester and allows the addition or removal of these drives.

The DRSVirtualDrive contract stores a virtual drive's basic information, such as its name, size, and bytes available, as well as the root directory of the drive which contains the directory structure and file metadata. Additionally, it contains a DRSDataStorageContract that is made between the requester of the virtual drive and each provider backing up the drive's data. From this contract the directory structure and file metadata can be viewed, upload and download requests are issued, and files and directories can be added and deleted.

Within DRSVirtualDrive, two structs, File and Directory, are used to store the virtual drive's directory structure and file metadata. The DRSVirtualDrive contract has a reference to a single Directory struct, the root directory. Within each Directory struct, a list of files and other directories is stored. To find, add, or delete a Directory or File, the appropriate function in DRSVirtualDrive is called with a directory path, telling it how to navigate from the virtual drive's
root directory to the location of the target Directory or File.

The last contract is DRSDataStorageContract, which is essentially an agreement between two users, one acting as the requester and the other the provider, as to the pricing of data storage and transfer for a single virtual drive. As stated previously, a single DRSVirtualDrive has one DRSDataStorageContract for each provider that backs up the data, each one potentially having different pricing and constraints. The prices are agreed upon when the DRSVirtualDrive is created and cannot be changed. The prices stored include the price per byte transferred (for data transferred by uploads and downloads) and the price per byte per period (e.g. a billionth of an ether every month for each byte, equating to a gigabyte of storage costing one ether per month). In addition to pricing agreements, DRSDataStorageContract defines a FileRequest struct and provides functions to create and transact with those structs. When the upload or download functions of DRSVirtualDrive is called, an specific amount of ether is sent to the contract to pay for transfer fees. After creating or finding the file metadata, the upload and download functions iterate through each DRSDataStorageContract it references, first passing them their portion of the fee and then calling the FileRequest creation function with the file's metadata. After creation, the FileRequest structs can be handled individually by requester and providers through the DRSDataStorageContract. It provides functions to accept and complete a FileRequest, both of which release half of the fee to the provider, the former being done first and only by the provider, the latter being done last and only by the requester. Additionally, the requester is given access to the function to retract a FileRequest and return all remaining funds back to the requester at any time before completion, which is used when a download does not validate, a download is not needed, or when a response is not received from a provider for an upload or download request.
4.4 DRS Manager Core Class Design

4.4.1 Ethereum Interface Design

Interactions with the Ethereum network and managed through calls to the interfaces provided by the Ethereum client. While this makes things easier, it can be made cleaner and simpler by placing additional layers of abstraction over the client interface, making interactions between DRS Manager components and DRS smart contracts more manageable. To send transactions through the Ethereum client, numerous details are needed, such as the amount of gas to use, the gas price, the transaction data in byte array form, the key of the sender, and the address of the receiving contract or user. Instead of handling that low level data anywhere interactions with contracts are needed, some of these details are handled first by the BlockchainManager and
simplified further by ContractWrapper and its subclasses. This interface to the Ethereum network is shown in the simple diagram in Figure 4.6, where it can be seen that all data to and from the Ethereum client goes first through ContractWrapper subclasses and then to the BlockchainManager (with the exception of the EthereumEventManager class which only reads Ethereum log data with the help of the Ethereum client, as described in the section on Ethereum Event Management).

The BlockchainManager interface provides several convenience methods for sending transactions. One such method requires only the receiving address and the byte array of the transaction data, with the sender's key being retrieved from the SessionManager (described in the next section) and other values set to defaults. The BlockchainManager also provides methods to wait for the inclusion of a transaction in the blockchain by listening for new block events and notifying the waiting thread when a block is found with the relevant transaction receipt. This is useful for when a new created contract or data returned from a transaction is needed to continue processing. Finally, it provides the option to handle resending transactions if they are not included after some number of blocks for whatever reason.

All DRS Manager interactions with the DRS smart contracts are done through ContractWrapper subclasses. This abstracts away the details enough that dealing with the DRS contracts is as simple as dealing with any other class. ContractWrapper does several things to provide this simplicity. First, instances of it are created with a reference to the address of the contract being wrapped so that when calling the send transaction methods of BlockchainManager, the address can be used as the receive address of the transaction. ContractWrapper is itself essentially stateless, but an instance's contract address is what it uses to access and modify the state of the contract on the blockchain. Second, it contains data about the functions of the contract it is a wrapper for. This allows easy transformation of function inputs and outputs to and from
byte array data that is used by the Ethereum client. Finally, it provides methods to call contract
functions and constant functions (read only operations that don't require a new transaction). While
these methods are much better than dealing with transaction byte data directly, they are not
specific to any contract. These functions accept any number and type of arguments and return a
list of an unspecified type. This is remedied by the contract specific ContractWrapper subclasses.

Each subclass of ContractWrapper provides methods for interacting with all the public
methods of a specific contract, such as the DRSUser contract. The methods have the exact names
as the contract's functions and the equivalent input and output types. This is much less error prone
and easier to deal with, as the method parameters are set and the return type is known. This final
level to the Ethereum interface makes dealing with contracts as simple as any other object.
4.4.2 Session Management

Figure 4.7: DRS Session Management Design

The DRS Manager application operates with one user account at a time. The private key of any valid Ethereum account can be used to sign in. An instance of the SessionManager class stores the account key and uses the account address to query the DistributedResourceSharing contract for the DRSUser contract associated with the address. If none is found, SessionManager sends a transaction to the DistributedResourceSharing contract to add a new DRSUser contract linked to the account address and a wrapper for the DRSUser contract is stored.

Several classes reference the SessionManager instance, as shown in Figure 4.7, because their operations are intended to be done by or for a single user, the one currently signed in.
BlockchainManager, described in the previous section, uses the key of the signed in user every
time it sends transactions to the Ethereum client because these transactions must be signed by the
user sending the transaction. VirtualDriveManager utilizes SessionManager during virtual drive
creation to create the drive in the appropriate DRSUser contract. FileManager and
FileRequestWebService both utilize it to sign and validate requests. Finally, the display of ether
owned by the user references SessionManager when it updates periodically.

Whenever a session is changed with a log in or log out, the SessionManager sends an
event that notifies listeners of the old account key and the new account key. Several user interface
classes utilize this to refresh their views with the data from the current user. For example, the
virtual drives table is clear and repopulated with the drives owned by the current user. When a
user signs out, most user interface functions are disabled since they can't be done without a user.
4.4.3 Virtual Drive Management

The creation of virtual drives is relatively simple, as seen in Figure 4.8. When the add virtual drive button of the ClientDrivesTableView is clicked and the form that is then displayed has been filled out and submitted, the VirtualDriveManager is passed the data to create a new drive. The data contains the drive name, size, and provider count. For simplicity, pricing data is set to default values and and the DRSUser contracts chosen as providers are selected randomly. SessionManager is used to get the DRSUser contract wrapper for the current user and calls the function to add a virtual drive, passing it the aforementioned data. Again for simplicity, the providers automatically accept the DRSDataStorageContract for this DRSVirtualDrive.

Figure 4.8: DRS Virtual Drive Management Design
4.4.4 File Request Management

File upload and download requests are handled by direct and indirect interactions between the requester's FileManager and the provider's FileRequestWebService, as shown in Figure 4.9. The direct interactions are mediated by a REST web service and indirect actions by transactions with DRS Ethereum contracts.

When a user selects a file to upload or download to or from a virtual drive, the FileManager class is used to fulfill this request. FileManager calculates the total transfer fee required and includes that in a transaction to the DRSVirtualDrive contract's upload or download function, which creates a FileRequest for each DRSDataStorageContract used by the drive. FileManager then gets each DRSDataStorageContract from the DRSVirtualDrive and uses them to get the IP address of each provider for this drive. Using that address, a web service call is made...
to each provider with all the information needed to verify the requester's identity and to find and validate the data needed for the request. In the case of downloading, the providers are contacted sequentially until the file has been successfully downloaded, whereas for uploading the file needs to be given to all providers. After each call to a provider, the DRSDataStorageContract between that provider and the requester is told to either complete or retract the FileRequest, depending on whether the request was successful or failed, respectively. Completing the request will release the fee to the provider while retracting it will refund it to the requester. In the case of downloading, the requests for all providers that were not needed are retracted.

The FileRequestWebService class handles upload and download requests from requesters to providers. It validates several things such as that the requester is who they say they are, that this provider and the requester share a DRSDataStorageContract, and that the file hash matches the file sent for upload or retrieved for download. After validation, the DRSDataStorageContract is used to accept the request, which releases half of the fee amount specified in the FileRequest struct. The file's location on the provider's local drive will be a path starting with the DRS root directory, followed by the requester data directory, a requester specific directory named for their Ethereum account address, a directory for the virtual drive's name, followed finally by the path to the file on the virtual drive. Once validation is complete and the file request has been accepted, the file is stored at that location for uploads or retrieved from that location and returned for downloads.
4.4.5 Ethereum Event Management

When an event occurs, such as the creation of a new virtual drive, DRS components should be able to receive that event and handle it appropriately. Though contracts can define and log events, these events are not broadcast outside of the Ethereum network. In fact, they are simply log statements stored in the block where they occurred. They include the hash of the event signature, the address of the contract that logged the event, and the data determined by the event signature that is included when logging the event. The EthereumEventManager listens for new block events and class gathers and filters event logs from the blocks with the help of the
Ethereum client. Events can be filtered by the hash of the event signature, the address of the contract that logged the event, and the data for any parameter designated as indexed in the event signature. DRS components can implement the EthereumEventListener interface and add themselves to EthereumEventManager as a listener for a specific event filter. Then as new blocks are received, these components can be responsive to relevant contract events as they occur.
Chapter 5

DRS IMPLEMENTATION

This chapter highlights key aspects of the implementation of the DRS system, showing and explaining screenshots of the final product along with noteworthy pieces of code. In particular, the process of uploading a file to a virtual drive is covered in detail, highlighting the interactions between requester, provider, and the DRS smart contracts. First, a brief discussion of the technologies chosen for this project is given.

5.1 Technology Used

To develop this project, an Ethereum client, a language, and libraries were needed to create the client application, DRS Manager, which is used for both requester and provider roles, as well as a language with which to write smart contracts, of which there are a few. In this section the choices for which languages and libraries are used are described briefly.

For interacting with the Ethereum network, an Ethereum client is needed, of which there are eight, written in different languages such as C++, Python, and Go [11]. For developing the DRS Manager application, Java was chosen for it’s familiarity, strong library support and importantly, that there is an Ethereum client for Java (EthereumJ) that is maintained and developed. The EthereumJ client provides several things, including finding nodes running Ethereum, syncing blockchain data with those nodes, mining locally on a private blockchain for testing, sending transactions, notifying when there is a new block or other blockchain events, and providing helper classes for compiling contract code and for contract interactions.

The Java Spring library was chosen to handle dependency injection and to make
development of the REST web service for requester-provider interactions easier. Gradle is used for dependency management, which made it easy to stay up-to-date with the latest EthereumJ builds. Git and Github were used for source control. JavaFX was used for the user interface. Finally, the project was developed in Eclipse, save for the smart contract development. Initially the Mix IDE was used for contract development, but due to its limited capabilities and extremely frequent crashes, development was instead done with a simple text editor.

There are a few options for a smart contract development language, but only one, Solidity, is receiving the full support of the Ethereum Foundation. Given how new Ethereum is, even Solidity is lacking in features and can be buggy, so it seemed unwise to choose anything else. Smart contract languages like Solidity are high level languages that are compiled down to bytecode for execution on the Ethereum Virtual Machine. Solidity is an object oriented, JavaScript-like language. All DRS contracts were written in it and are compiled through EthereumJ using the 'solc' compiler.
5.2 Managing Login Sessions

Figure 5.1: The DRS Manager User Interface When Signed Out
When the user first starts the client application, DRS Manager, or they log out, the user interface displayed is shown Figure 5.1. All components are disabled until they log in by selecting the “Session” menu at the top left.

![Session Menu Options](image1)

**Figure 5.2: Session Menu Options**

![Login Dialog](image2)

**Figure 5.3: Login Dialog**

When the users selects “Login” from the menu shown in Figure 5.2, the login dialog appears, as shown in Figure 5.3, and requests an Ethereum account private key. As an Ethereum account is just an Elliptic Curve Cryptography (ECC) asymmetric key, a new user can simply use ECC to generate a public/private key pair and use that for all future interactions with the DRS client. The key entered is stored by the SessionManager component and is used for signing all transactions sent to the Ethereum network to enable validation that the transaction came from that account. Immediately after getting the user's key, the address of that key (which is just the hash of the public key) is used to find if there is a DRSUser contract associated with this address in the
DistributedResourceSharing contract. If there is not, a new DRSUser contract is created and mapped to the address.

```
contract DistributedResourceSharing {
    DRSUser[] internal m_users;
    mapping (address => uint) internal m_userIndexMap;

    function DistributedResourceSharing() { }

    function addUser() public returns (DRSUser user) {
        uint index = m_userIndexMap[msg.sender];
        if(index == 0){
            m_users.push(new DRSUser(msg.sender));
            index = m_users.length;
            m_userIndexMap[msg.sender] = index;
        }

        return m_users[index - 1];
    }
}
```

**Figure 5.4: New User Creation Contract Code**

The Solidity code shown in Figure 5.4 shows how the map of addresses to users is maintained and modified by interactions with the DistributedResourceSharing contract. The mapping type used in Solidity is not iterable, which is very inconvenient, so a simple workaround was devised by using an array of DRSUsers and a map from a user address to the array index of their DRSUser contract. In the addUser function, DRSUser for the sender address is searched for, created if not found, and then returned. The new DRSUser is created with the address of the sender and added to the mapping. Note that “msg.sender” is an implicit global variable and is the address of the user or contract that made the call. The function addUser is called directly by the user with the key they signed in with so the DRSUser contract is owned exclusively by them.
5.3 Viewing User Data

Figure 5.5: The User Interface When Logged In
The user interface components are all enable once the user has logged in, as shown in Figure 5.5. Near the top is a display of the user's ID, which is just the address of the key they signed in with, along with the amount of ether currently stored at that address (the amount of ether shown is astronomical because this was just an account on a local blockchain for testing). Below that is the virtual drives table, which displays the name, size, and space available of each drive owned by the user. When selected, a drive's contents are displayed in the table at the bottom of the window, with the content type (directory (D) or file (F)), name, size and date modified displayed for each element.

5.4 Creating And Deleting Virtual Drives

Virtual drives can be created or deleted with the buttons above the virtual drives table, highlighted in Figure 5.6. Click the red 'X' button will remove the selected drive from the list and delete the associated DRSVirtualDrive contract and all its associated data. Clicking the green plus
button displays the dialog for creating a new virtual drive, as shown in Figure 5.7. The user enters the name, size, and number of mirrors (the users storing the data). After clicking create, the drive is created and awaits acceptance. Once accepted, it will show up in the virtual drives table and is available for use. For simplicity, the providing users are selected at random from the list of all users (from the DistributedResourceSharing contract) and by default providing users will automatically accept the pricing contracts for new drives. Methods for selecting the best possible users at the best prices are discussed in the following chapters.

```solidity
modifier userOnly()
{
    if(msg.sender != m_userAddress) {
        throw;
    }
    // "_" indicates where the body of code goes
}

function addVirtualDrive(string name, uint size,
    DRSUser[] servers, uint pricePerBytePerPeriod,
    uint64 paymentPeriod, uint pricePerByteTransferred)
    public userOnly returns (bool driveAdded) {
```

**Figure 5.8: Example Of Modifiers In Virtual Drive Creation Contract Code**

If a contract's function is public, such as DRSUser's function addVirtualDrive in Figure 5.8 is, it can be called by any user or contract with the contract's address. This function takes the data it is given to create a new virtual drive in this DRSUser contract's list of drives. This is something that only the owner of the DRSUser contract should be able to do. To prevent other entities from calling it, we can add a modifier to the function. Modifiers can be added to a function definition to run code that is frequently used. Adding the “userOnly” modifier to addVirtualDrive runs a check to see that the address calling this function is the same as the address of the user that owns this DRSUser contract. If this check fails, an exception is thrown and the transaction fails, meaning that nothing happens. The ‘_’ seen in Figure 5.8 indicates that the code for the function being modified is injected there, so after userOnly passes its check, the
addVirtualDrive function is run, now with certainty that the call is from the account owner.

5.5 Structuring And Navigating Virtual Drive Contents

![Virtual Drive Content Management Controls](image)

Figure 5.9: Virtual Drive Content Management Controls

When viewing a virtual drive's contents or modifying them in any way, the virtual drives table at the bottom, along with controls above it, highlighted in Figure 5.9, are used. Navigating the content is done by double clicking directories to open them and clicking the back button to go to the parent directory.

![Directory Creation Dialog](image)

Figure 5.10: Directory Creation Dialog

To structure the drive with directories the new directory button to the right of the back button is selected and the new directory dialog is displayed, as shown in Figure 5.10. All that is required is the name of the new directory. After entering this, a new directory is created within the virtual drive in the directory currently being view in the virtual drive contents table.
Adding a new directory is accomplished by a call to the `addDirectory` function in the `DRSVirtualDrive` contract, as shown in Figure 5.11. It requires the path of the parent directory and the name of the new directory. The path is a list of hashes of directory names, needed to navigate from the root directory of the virtual drive to the parent directory of the new folder. After checks are made that the path exists and the directory name isn't taken, a new directory is added to the parent directory. Before completing, a `DirectoryAdded` event is logged, which can trigger events in the DRS client's Java code with the help of the `EthereumEventManager` component, which can monitor new blocks in the Ethereum blockchain for relevant events. The virtual drive contents table listens for this event and updates its display to include the new directory.

To delete files or directories, the red 'X' above the virtual drive contents table is clicked and the selected table entry will be deleted. For files the function to accomplish this is as simple as finding and deleting the file metadata in the `DRSVirtualDrive` contract and then increasing the available space on the drive by the size of the deleted file. The function for deleting directories

```solidity
def addDirectory(bytes32[] path, string name)
    public onlyOwner returns (bool added) {
        Directory parent = getDirectory(path);
        if(path.length > 0 && isRoot(parent)) {
            return false;
        }

        bytes32 nameHash = sha3(name);
        uint index = parent.directoryIndex[nameHash];
        if(index != 0) {
            return false;
        }

        parent.directories[parent.directoryCount] = Directory(
            name, 0, 0, block.timestamp, block.timestamp);
        parent.directoryCount++;
        parent.directoryIndex[nameHash] = parent.directoryCount;

        DirectoryAdded(m_owner, path, name);
        return true;
    }
```

**Figure 5.11: Directory Creation Contract Code**
uses recursion to eradicate all contained content. It first deletes all the directory's files using the same function used to delete individual files, then it makes a recursive call on each subdirectory to wipe out the entire directory structure beneath it.

5.6 Uploading And Downloading Files

The last and most important functionalities are the uploading and downloading of files. The green up arrow in Figure 5.9 initiates the upload process, while double clicking a file in the virtual drive contents table initiates the download of that file. Both functions bring up a dialog like that in Figure 5.12 for selecting a file to be uploaded or a location to download the file to.
Figure 5.13: File Upload Initiation Client Code

```java
for (DRSDataStorageContract contract : drive.getDataStorageContracts()) {
    feeTotal = feeTotal.add(contract.getFee(fileSize));
}

DRSVirtualDrive.UploadStatus uploadStatus =
    drive.uploadFile(
        feeTotal, dir.getPathHashes(), file.getName(),
        new BigInteger(fileHash), fileSize).get();
```

The first steps for both uploading and download are getting the file size and the file hash, with uploads acquiring this from the locally selected file (as well as all the bytes of the file) and downloads get this from the file metadata for the file selected from the virtual drive. Both then proceed to use the file hash to calculate total transfer fee for all DRSDataStorageContracts for the selected drive, as shown in the first section of Java code in Figure 5.13.

In second half of Figure 5.13, the call to DRSVirtualDrive's uploadFile function is made. The downloadFile function is similar, but requires only the information to find the file metadata, not to create it. Note that these are not the contracts themselves, but the Java wrapper classes discussed in the design section, which make interactions with them as simple as any other object. The calls to these functions pay the amount specified by the feeTotal variable from the user's account to the DRSVirtualDrive.
For file uploads, validation of available space and the creation of the file metadata are the first steps, as shown in the first half of Figure 5.14. After this, for downloads as well, the DRSVirtualDrive creates a file request in each DRSDataStorageContract, splitting the money it just received between each of those contracts according to their pricing terms, as seen in the second half of Figure 5.14. An interesting point to note is the use of exceptions here, which are to make sure all the currency is paid or none of it is. By using the “throw” keyword the transaction is invalidated, so it can be made certain that if one of the payments or request creations fails, all of them are undone.
The next step is to make a REST web service call to the users providing the data. The IP address of each storage provider for the DRSVirtualDrive is found in their DRSUser contract which DRSDataStorageContract has a reference to. For file downloads the URL is created and called as shown in Figure 5.15, which is similar to uploads except a file is sent with the request instead of returned from it. The fileHash parameter is used to find the correct file request struct in the DRSDataStorageContract located at the address given by the dataStorageContract parameter. The filePath parameter is used to find the file on the provider's system. All the other parameters are for validating that the request is to the right provider, that the requester is who they say they are, and file hash verification. After all the validation is done, the web service running on the provider's machine will accept the file request to release half of the transfer fee to them, after which they store the file and return a success status or retrieve the file and return the file bytes.
Figure 5.16: File Request Completion Contract Code

For uploads, each provider is contacted to give them the data being uploaded, while for downloads, providers are contacted in sequence until one is able to provide the data. Every successfully completed file request is marked as completed by using the completeFileRequest function of DRSDataStorageContract, as shown in Figure 5.16. Unsuccessful or unneeded requests use the retractFileRequest function to reclaim the money paid for the fee. The completeFileRequest function can only be called by the user requesting storage. It first finds the request, then validates that has been accepted by the provider and not yet completed, pays the remaining fee for the request to the provider, and finally zeroes out the remaining fee and marks the request as completed. This is the final step in the process. Uploaded files are now accessible anywhere and downloaded files are ready to use.
EVALUATION AND FUTURE WORK

6.1 DRS Transactions And Their Costs

In addition to the money paid to providers for hosting and transferring data, transacting with DRS contracts on Ethereum also costs money. As described in the background chapter, Ethereum transactions are fueled by gas, which is paid for in ether. The cost of that ether in US dollars for DRS transactions vary from fractions of a penny to just over $2 (at the current price for one ether, approximately $12). The more costly transactions are those that are done once for any given user, such as creating a new DRSUser, or done infrequently, such as creating a new DRSVirtualDrive. With the permanent nature of all data on the blockchain, data storage is far and away the most expensive operation and contract creation can take a significant amount storage depending on the size of the contract code. Below in Table 6.1 the cost in gas, ether, and US dollars of some DRS transactions can be seen. Note that while several of these transactions are not directly creating contracts, the code they are calling does indeed create and store contracts (e.g., the addVirtualDrive function of DRSUser).

<table>
<thead>
<tr>
<th>Transaction (contract – function)</th>
<th>Gas</th>
<th>Ether Cost</th>
<th>US Dollars</th>
</tr>
</thead>
<tbody>
<tr>
<td>DistributedResourceSharing – contract creation</td>
<td>7,061,993</td>
<td>0.177</td>
<td>$2.12</td>
</tr>
<tr>
<td>DistributedResourceSharing – addUser</td>
<td>5,118,564</td>
<td>0.128</td>
<td>$1.54</td>
</tr>
<tr>
<td>DRSUser – addVirtualDrive</td>
<td>5,683,567</td>
<td>0.142</td>
<td>$1.70</td>
</tr>
<tr>
<td>DRSVirtualDrive – addDirectory</td>
<td>138,181</td>
<td>0.0035</td>
<td>$0.04</td>
</tr>
<tr>
<td>DRSVirtualDrive – uploadFile</td>
<td>419,942</td>
<td>0.01</td>
<td>$0.13</td>
</tr>
<tr>
<td>DRSVirtualDrive – downloadFile</td>
<td>94,374</td>
<td>0.0024</td>
<td>$0.03</td>
</tr>
<tr>
<td>DRSDataStorageContract – acceptFileRequest / completeFileRequest / retractFileRequest</td>
<td>28,384</td>
<td>0.0007</td>
<td>$0.008</td>
</tr>
</tbody>
</table>

*Table 6.1 DRS Transaction Costs*
These costs are not enormous, but they are far from ideal when considering they are in addition to storage and transfer fees. If a system similar to the one used in this project were to be competitive, it would need to keep these costs very small. In face, with better design of the smart contracts, these costs, especially for the transactions that create new contracts, could be significantly reduced. This can readily be done if the code for operating on a contract is deployed only once and used as reference. Then transactions for new contracts only have to store space for the data they need, rather than deploying the whole library of code every time.

6.2 Performance, Availability, Security, And Pricing

The main goal of this project was to show that idle disk storage can be put to use by creating a peer-to-peer alternative to centrally controlled cloud systems, one where users can sell their storage and buy from others to create their own cloud storage. In that, it has succeeded by creating a simple prototype that shows this is possible. However, there is much work to be done to make such a system become a viable competitor to the cloud storage services that dominate the market today. No attempt was made on this project to offer better performance, availability, security, or pricing and undoubtedly if tested on those metrics, it would fall far short of any cloud storage service offered currently. This does not mean that such a system could not hope to compete. In fact, the opposite may be true, and with the right design and optimization it may outperform on some if not all of those metrics. There are numerous ways these metrics could be improved in future work, many of them being obvious and even easy, but which were not included in this project mainly because they weren't necessary for what this project aimed to demonstrate. A few of these are noteworthy and are described below.

The first improvement needed in regards to security is encryption of user data, the
importance of which should be obvious for a system like the one in this project. Clearly a user would not want their private data stored unencrypted on the drives of random people that they have no reason to trust and such a system would never be taken seriously. This would be relatively easy to implement by simply generating a symmetric key for data encryption, encrypting that key with the user's Ethereum account public key, then stored that encrypted key in the smart contract for that user's account. This would make it available anywhere, but only usable by them. This simple improvement would be an immeasurable increase in security given that once encrypted, not only are the users hosting the data unable to view it, but neither are any attackers or governments without first being able to break the encryption.

Another improvement would be the addition of a rating system for users providing storage to help increase performance, availability, and pricing. The client application could keep statistics on how often each data provider is available and what their average transfer speeds are. This data could be stored in a ratings contract on Ethereum and used whenever a user requests cloud storage space. The users that provide this storage space should be automatically selected from those with the best ratings, though perhaps the requesting user may wish to choose less than ideal candidates that offer their services at a lower price. With a rating system like this in place, there are stronger incentives for users selling data to do so cheaply while still providing the best performance and availability that they can offer.

The final improvement to be discussed is splitting data between multiple drives, which has a strong effect on performance and important implications for availability and security. The prototype discussed in this project only concerns itself with data redundancy by using multiple providing users as mirrors, or exact copies, of the requesting user's data. For example, if a user purchases 1 gigabyte of storage from 8 different users, the data stored in that gigabyte is the same for all 8 users. If this prototype were to add data striping to split the bytes of files between all the
providing users, 1 gigabyte could be stored by 8 users, each one providing only 128 megabytes of space. For any file, one eighth of it would be stored on each drive, so uploads and downloads would be 8 times as fast. However, this would also make it about 8 times less likely that the requesting user is able to upload or download data because every single provider needs to be available. This can be mitigated by having more providers mirroring the data for redundancy. To do this in the previous example, the 8 users would be doubled to 16, with two users mirroring each 128 megabytes, for a total of 2 gigabytes purchased for one level of redundancy. Another effect of splitting the data between providers is that it increases security. Even if the data wasn't encrypted, having it split like this means that no one has a user's data in any usable form, only fragments. Combined with encryption, this system could arguably have better security than other non-distributed cloud storage services could hope to achieve. Simply by having user data distributed on this peer-to-peer network, the incentive to hack or spy on that data in any significant way is hindered because while individual users that provide data may be relatively easy to compromise, the rewards for doing so are drastically smaller. Only a compromise of a large part of the peer-to-peer network could provide the amount of data worthy of a hack or a government's interest. Adding encryption and data splitting to that means that, even if a user were a high profile target worthy of a sophisticated attack, their data is still extremely hard to piece together and decrypt.
As we move more and more towards cloud systems to provide solutions to our computing problems, it is important that alternative architectures are explored to be certain that the solutions provided are the best ones to meet our needs. With most public cloud storage systems being offered by a single entity, be it Amazon or Google, the alternative of a peer-to-peer cloud seems ripe for exploration. Additionally, with the rise of the so called “Sharing Economy,” it is only a matter of time before digital resources become a part of that economy too and these resources are what can make a peer-to-peer cloud a reality. This project aimed to be a small stepping stone to that reality by showing that idle storage resources can be utilized by a purely peer-to-peer cloud. This was done with a simple prototype that uses a client application to interact with smart contracts written for the distributed application platform, Ethereum. This interaction allows users to buy or sell other users' unused disk space. The disk space that gets sold hosts user data and acts as the distributed equivalent of datacenter storage for large cloud storage services. When users purchase disk space, they purchase from several users and create their own cloud storage purely from peers, utilizing a resource that would otherwise go wasted and they do all this without the need for any central authority to facilitate it.

As the last chapter discussed, this project aimed to highlight a new paradigm for the cloud, not focusing on any standard metrics of success, which leaves much to be desired in terms of usability. Since the start of this project, some new platforms with similar ideas have been gaining attention and they aim to achieve those metrics of success that this project did not. Notably, Storj Labs Inc. has recently released a free beta of their product called Storj, which their whitepaper describes as a blockchain-based peer-to-peer network for end-to-end encrypted
distributed object storage [12]. This platform utilizes their own blockchain (as opposed to Ethereum's) and would provide real world solutions to the problems laid out in this paper and additionally, if it's claims are believable, it will be competitive with other cloud storage services in regards to performance, availability, security and even pricing. Another platform, called Golem, is about to start their first crowdfunding campaign and aims to use the Ethereum blockchain to create a peer-to-peer cloud for computing power similar to how this project does for disk space. In their recent whitepaper, they call it “the first truly decentralized supercomputer” that would be built from “contributions of individual and professional providers” [13].

Both of these new platforms are driven by a shared concern about the problems discussed in this paper, that of the dominance of centralized cloud solutions and the underutilization of computing resources. This project has shown that it can be done, but continued work on Storj, Golem, and others projects that follow suite will pave the way for making these peer-to-peer clouds a reality by improving performance, availability, security, and pricing. If they can start to become competitive on those metrics compared the current cloud giants, such as Amazon or Google, the advantages these new platforms have may start to sway most users. Perhaps then we may see a paradigm shift away from centralization as the norm, shifting instead towards solutions that utilize a distributed network of peers both for efficiency and decentralization.
BIBLIOGRAPHY


