Pay with Bytes: A Collaborative and Anonymous Storage Service

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Juan Ignacio Santa Cruz Cosp

Graduate Program in Computer Science and Engineering

The Ohio State University

2014

Master’s Examination Committee:

Dr. Kannan Srinivasan, Advisor

Dr. Prasun Sinha

Dr. Rajiv Ramnath
Abstract

In the past three decades the cost of storing digital information has been dropped exponentially [18]. However, the cost per byte of using cloud storage services is still expensive comparing to hard drive prices. In addition, when storing data in the cloud there is usually an implicit trust agreement with the service provider. After the NSA documents leaked in 2013 [20] many people have become aware that the Internet makes constant surveillance to end users a real threat. The purpose of our research is to address the concerns of privacy, anonymity and costs when using cloud storage services. In order to achieve these goals we used the following methods. First, a pseudo zero-trust collaborative architecture designed to trust no one. Second, address rewriting and multi-hop forwarding to hide the identities of paths source and destination based on Tarzan [19]. And finally, hard drive reutilization from every peer connected to the system. We are proposing a collaborative and anonymous storage service that will be useful for backup, share and publishing files. Every node will provide hard drive space from its own computer that will be used by other peers. In return, every node will obtain a certain amount of storage capacity
among its peers. Another important feature taking into account when designing the system was the ability to support low storage capacity nodes such as smartphones and tablets. PayWithBytes will not only be usable in corporate intranets but also for any Internet user requiring this service.
This document is dedicated to my family. Specially to my niece Emma, and my two nephews Tomas and Mateo...
Acknowledgments

It is with immense gratitude that I acknowledge the help of my thesis advisor, Dr. Kannan Srinivasan. Without his generous advice, guidance, encouragement and professional assistance this thesis could not have been completed. Furthermore, he not only helped by reviewing again and again the ideas of this thesis, but he also kept me motivated to work and do research since my very first weeks of classes 2 years ago. In my opinion, keeping the students highly motivated is the difference between a great professor and an outstanding one. The first semester was probably the hardest one for me, because I had to adjust to a very different academic system. However, his help and inspiring words made it easy for me to stay focused and to continue to try my best. Having the opportunity of working closely with him has been a huge honor for me.

My gratitude is also extended to Dr. Rajiv Rammath and Dr. Prasun Sinha for providing me with significant and constructive suggestions in the oral examination. Their suggestions have defined many new aspects in which I will like to extend this thesis in the future.

Finally, I want to thank my family for their constant support. Specially to Pablo for pushing me to pursue this degree, and to Rosita and Nico for everything. Mom and dad, it’s you when I look in the mirror.
Vita

December 2004 .......................... Cristo Rey Jr. High, Asunción, Paraguay

2010 ........................................ B.S. Computer Science, Catholic University - Asunción, Paraguay

2010 - 2012 ............................ Software Development Engineer, Roshka - Asunción, Paraguay


2012 present ............................ Graduate Student, The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Related Work</td>
<td>4</td>
</tr>
<tr>
<td>3. System Architecture</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Contract of a Transaction</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Receipt of a Transaction</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Store a File</td>
<td>11</td>
</tr>
<tr>
<td>3.4 Retrieve a File</td>
<td>15</td>
</tr>
<tr>
<td>3.5 Delete a File</td>
<td>16</td>
</tr>
<tr>
<td>3.6 Share Private Files</td>
<td>19</td>
</tr>
<tr>
<td>3.7 Share Anonymous Public Files</td>
<td>20</td>
</tr>
<tr>
<td>3.8 CLS Database Encryption Scheme</td>
<td>21</td>
</tr>
<tr>
<td>3.9 Anonymous Network</td>
<td>23</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4. Consistency of the System</td>
<td>24</td>
</tr>
<tr>
<td>4.1 Proof of Storage</td>
<td>24</td>
</tr>
<tr>
<td>4.2 Nodes Leaving the Network</td>
<td>26</td>
</tr>
<tr>
<td>4.3 Users Quota</td>
<td>26</td>
</tr>
<tr>
<td>4.4 Reputation System</td>
<td>27</td>
</tr>
<tr>
<td>5. Implementation and Availability</td>
<td>29</td>
</tr>
<tr>
<td>5.1 Results</td>
<td>31</td>
</tr>
<tr>
<td>5.2 Availability</td>
<td>33</td>
</tr>
<tr>
<td>6. Possible Attacks</td>
<td>35</td>
</tr>
<tr>
<td>6.1 Physical</td>
<td>35</td>
</tr>
<tr>
<td>6.2 Legal Actions</td>
<td>36</td>
</tr>
<tr>
<td>6.3 Denial of Service</td>
<td>36</td>
</tr>
<tr>
<td>6.4 Acquire Numerous Nodes</td>
<td>36</td>
</tr>
<tr>
<td>6.5 Anonymous Network attack</td>
<td>37</td>
</tr>
<tr>
<td>6.6 Fake Receipts</td>
<td>37</td>
</tr>
<tr>
<td>6.7 CLS and Nodes Compromised</td>
<td>38</td>
</tr>
<tr>
<td>7. Conclusion and Future Work</td>
<td>39</td>
</tr>
<tr>
<td>7.1 Recommendation for Future Work</td>
<td>39</td>
</tr>
<tr>
<td>7.2 Thesis Summary</td>
<td>40</td>
</tr>
<tr>
<td>Bibliography</td>
<td>42</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Store Diagram</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Retrieve Diagram</td>
<td>15</td>
</tr>
<tr>
<td>3.3 Delete Diagram</td>
<td>17</td>
</tr>
<tr>
<td>5.1 Client Software Stack</td>
<td>30</td>
</tr>
<tr>
<td>5.2 Store Request Performance Comparison</td>
<td>32</td>
</tr>
<tr>
<td>5.3 Retrieve Request Performance Comparison</td>
<td>33</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

After the NSA documents leaked [20] incident, big companies such as Google, Facebook and Dropbox have joined the Digital Due Process [10]. The due process states that there is an outdated law that gives many government agencies the right to read all of our communications without warrant.

That law is the Electronic Communications Privacy Act (ECPA), it was written in 1986. The due process is proposing a major reform to the ECPA that will hopefully take effect soon. Some companies are taking the concerns about NSA data access with specific actions. For instance, Microsoft [30] will start offering data storage services outside the US borders for customers in foreign countries. Although this offer may be interesting for many users, it does not seem like a real solution to the problem. As we can see privacy is huge concern nowadays for any Internet user, no matter what e-mail, e-commerce platform or social media service we used.

On the other hand, there are anonymity issues. Although there are not explicit laws in the U.S. that guarantees the right to remain anonymous there are some precedents that confirms that anonymity in certain cases is protected by law. For instance, the U.S. Supreme Court decision in the McIntyre vs. Ohio Elections Commission case in 1995 [16] stating the following: "...an author’s decision to remain anonymous, like
other decisions concerning omissions or additions to the content of a publication, is an aspect of the freedom of speech protected by the First Amendment.”

Anonymity is essential in many other scenarios. For instance, when using torrents for sharing and downloading files. The authors of OneSwarm [22] provides the following example. If an attacker realizes that a user is downloading a Linux security patch, then the users machine is probably unprotected against that exploit. The main issue is that there are techniques to identify which IP address is downloading a given torrent. Furthermore, there are methods that can even reveal the identity of the user. These methods do not necessarily have to be performed by large brand-name companies, they can be executed by almost any ordinary user [24].

In the past three decades the cost of storing digital information has been dropped exponentially [18]. However, the cost per byte of using cloud storage services is still expensive comparing to hard drive prices. This is due to the fact that storing bytes needs a constant investment of money to be preserved in time [18]. Most cloud storages services are powered by data centers, which make the costs rise. In addition, when storing data in the cloud there is usually an implicit trust agreement with the service provider.

The purpose of our research is to address the concerns of privacy, anonymity and costs when using cloud storage services. In order to achieve these goals we used the following methods. First, a pseudo zero-trust collaborative architecture designed to trust no one. This includes the centralized server that authenticates users and their transactions. Second, address rewriting and multi-hop forwarding to hide the identities of paths source and destination based on Tarzan [19]. And finally, hard drive reutilization from every peer connected to the system.
We are proposing a secure, collaborative and anonymous backup and sharing files system. It will not only be usable for corporate intranets but also for any Internet user requiring this service. We are developing as one project, which could potentially be deployed on the Internet and also in any given intranet. A key aspect of the architecture is to not sacrifice performance and user-friendliness. This is vital because otherwise users will not join the network. The more users we have the high anonymity we will be able to achieve.

The essential features we want to provide to the users of PayWithBytes are distributed backup, public and private file sharing and anonymous publication of files. The goal of the architecture design is to protect and hide the users identities, not only of those providing new content but also of the users retrieving them. Every architecture design decision was made taking into account the importance of privacy, anonymity and resource reutilization (low costs).

The rest of this thesis is organized as follows. Next, in Chapter 2, we briefly discuss related work. Chapter 3 discusses the details of the system architecture. After that, Chapter 4 gives some of the mechanisms to keep the system consistent. Chapter 5 describes the implementation. Next, Chapter 6 presents a list of the possible attacks against the system. And finally, Chapter 7 includes our conclusions that we have been able to achieve so far and the description of future work.
Chapter 2: Related Work

**Free Haven** project is a peer-to-peer, distributed, anonymous and persistent data storage developed at MIT [29]. To the best of our knowledge the system is not currently deployed on the Internet and the development has been discontinued.

One of the difficulties that the authors claimed with respect to the design of Free Haven is the need of broadcasting messages in order to retrieve content. As we mentioned before, Free Haven is a peer-to-peer distributed system, which makes the architecture different to our proposal. This is due to the fact that we have a Centralized Server with some basic responsibilities and resources. In addition, we do not require to broadcast messages for retrieving content.

**One Swarm (BitTorrent extension)** is a widely used BitTorrent compatible system that enables users to have an anonymous mode [22]. This mode allows users to share and download files anonymously. The system is specifically for sharing torrent files anonymously, so they actually solved a different problem than the one we are addressing. For instance, there is no need to protect the publisher of a given torrent file. Furthermore, as we have mentioned before if the anonymity mode is off then the users are vulnerable to some attacks [24]. In our proposal we aim to protect the user’s privacy at any moment, no matter what type of file he/she is downloading, sharing or publishing.
SymForm is a widely used distributed backup system that supports many different native applications clients [21]. The system allows the users to contribute by sharing a portion of their disk or by paying with money. They do not support sharing files or any type of anonymity. Furthermore, they encrypt your data with their own keys, which give the users two options. Trust that SymForm will not spy on user’s data or encrypt the data before backing up.

Freenet is a distributed anonymous information storage and retrieval system [15]. The peer-to-peer system is currently online and supports chats, forums, publication of files and some other features while protecting the anonymity of authors and readers. When a user wants to share a file they just insert the file into the network and the file is automatically distributed among different nodes. Chunks of data do not have a specific responsible. On one hand, data that is frequently retrieved is going to remain in the network. On the other hand, cold data will be drop whenever a node needs more space to store new data. This is one of the design decisions that make Freenet different from our proposal. Since there is no support for long-term permanent storage in Freenet, then providing a backup service on top of the network is not possible. As we have mentioned before, backup is one of key features we provide.

Although all of the systems mentioned before provide services such as backup, anonymous publication and sharing files separately, PayWithBytes addresses all of these requirements at once with an efficient, secure and anonymous design. Furthermore, the architecture provides anonymity for content providers and also for content consumers. PayWithBytes is not a full peer-to-peer system and neither it is a complete client-server architecture. This is the reason why we defined this design as an
hybrid between these two architectures and we called it a "collaborative" storage service.
Chapter 3: System Architecture

Before getting into the details of the architecture we present an overview of how the implementation will work. Every client will have to locally install PayWithBytes in a computer and shared a portion of its own hard drive for other peers to use. After that, the client will select a folder on its own hard drive and all the files in that folder are going to be replicated among different peers. Since the files are replicated, the user does not need to keep its computer connected to the Internet in order to successfully share or publish a file.

Private files are going to be encrypted with symmetric key encryption and shredded in multiple redundant files (or chunks). Every user will also have to generate a pair of public-private keys for encryption and decryption of messages. The metadata that will indicate how to re-assemble a given file will be hold in the local machine. A Centralized Light Server (CLS) will store an updated version of this metadata. It is essential to keep this version up to date due to the fact that a given computer could crash and without that metadata it would be impossible to recover files.

We described the CLS as ”light” in the sense that this is a data-center less system; only metadata will be stored in that server. As a part of the pseudo zero-trust architecture we do not trust the CLS and we do not provide any type of unencrypted
user data to the CLS. Even though one of the functionalities of the CLS is to certify transactions, anonymity is still preserved.

In the next 7 subsections, we are going to describe in detail the protocol that allows all of the PayWithBytes functionalities. In order to explain the details in a readable way, we introduce a simple example that will be used throughout these subsections. Node A needs to store a 2MB file in different peers. The file will be partitioned in 3 chunks (UUID\_X, UUID\_Y, UUID\_Z) of 1 MB each and they will be store in nodes B, C and D respectively. The 3 chunks together have 3MB because when splitting the original 2 MB file we are adding redundancy in each chunk. This will allow us to retrieve and reconstruct the entire file by only using 2 out of the 3 chunks.

The notation used for cryptographic primitives throughout this chapter is the following:

- \( H(M) \) : one-way hash of \( M \)
- \( E_k(M) \) : \( M \) encrypted with key \( K \)
- \( D_k(M) \) : \( M \) decrypted with key \( K \)
- \( k_A \) : public key belonging to \( A \)
- \( k'_A \) : private key corresponding to \( K_A \)
- \( a \) : as a subscript of \( E^a_M \) means a symmetric key generated by node \( A \)

The superscript for encryption and decryption indicates whether a symmetric scheme (\( t = s \)) or public key scheme (\( t = p \)) is used. The + operator indicates concatenation.
3.1 Contract of a Transaction

An unsuccessful attempt of storing a chunk can occur in many different scenarios. Step 2 and 3 of the store protocol (see section 3.3) can fail for different reasons. In scenarios with any type of failure, node A (the node trying to store data) will be responsible of the rollback operation. In order to rollback the operation we introduced the concept of a contract. The semantics of the contract is that node B has enough space in its harddrive to complete the request and that it will try to successfully complete the store request.

The contract will be send from node B to A before the Step 2 of the store protocol. It will contain a digital signature of B for non-repudiation purposes, plus the following information.

Contract{
  type : "Contract",
  file_uuid : E_{kB}(UUID_X),
  chunk_hash : H(chunk_content),
  solved_challenge : E_{kB}(solved_challenge_X),
  transaction_id : E_{CLS}(transaction_id_X, 1MB),
  size : "1MB",
  digital_signature : E_{kB}(H(contract))
}

We want to encrypt UUID_X with the public key of B because we do not want the CLS to be able to map a transaction_id with a file_uuid. The rollback is essential because the transaction_id_X will only be valid for N (total number of chunks) requests.
After a store operation failure, node A will send the contract to the CLS who will be able to review it and increment the validation of transaction_id_X if necessary. Any dispute that could rise between nodes A and B should be solved at this point.

Considering that node B does not trust node A, after receiving the contract it should check that all of the values are accurate. Failing to do this can allow B to misbehave and the rollback operation to be inaccurate. It is worth to mention that not following the protocol appropriately will affect the node’s reputation.

### 3.2 Receipt of a Transaction

We introduced now the concept of a receipt of a given store transaction. If node A successfully sent a chunk to node B, after node B store in its own hardrive it will have to send a receipt of that transaction to node A.

In order to provide authentication, non-repudiation and integrity node B will include a digital signature. The receipt will include the following information.

\[
\text{Receipt}\{ \\
\text{file_uuid} : \text{H}(\text{UUID}_X), \\
\text{file_hash} : \text{H}(\text{chunk_content}), \\
\text{valid_until} : \text{timestamp}, \\
\text{request_validation} : \text{E}_{\text{CLS}}(M), \\
\text{digital_signature} : \text{E}_{\text{B}}(\text{H}(\text{receipt})) \\
\}\n\]

When node A wants to complain about B deleting a chunk before the timestamp expiration, it can send this receipt as a proof that node B was supposed to hold that chunk.

The request validation field is the authorization by a third party (CLS), which approves that the request to store was valid. We do not want the CLS to have the opportunity to link a UUID_X with a given transaction_id_X. That is why we make the
CLS sign the M, which is equals to $H(\text{chunk} + \text{UUID}_X)$. If the CLS is able to link a UUID$_X$ with a given transaction$_{id}_x$, then it can know which UUID belong to which user. As we have stated before, we are proposing a pseudo-zero trust architecture, we do not want to trust the CLS with that information.

### 3.3 Store a File

In this section we describe the 3 steps necessary in order to store data remotely in other nodes. With the purpose of remotely store the 2MB file, we need to execute step 1 through step 3 for chunk UUID$_X$ as it is shown in Figure 3.1. After that we are going to repeat step 2 and 3 for UUID$_Y$ and UUID$_Z$. As we have mentioned
before, every chunk has a size of 1MB. Consequently, we need to request for 3MB of storage.

**Step 1: Request a transaction ID**

From node A to CLS, the entire request will be encrypted with the public key of CLS.

Request{
  type : "Need Transaction ID",
  size : "Need to store 3MB",
  node : "A"
}

From CLS to A, response is encrypted with Public Key of A.

Response{
  type : "New Transaction ID",
  value : "transaction_id_X",
  store_challenges : {sc1, sc2, sc3}
}

The CLS will store the information that indicates that transaction_id_X was sent to node A and also the metadata about splitting the 2MB file into 3 files of 1 MB each. The message decryption and the later use of the transaction_id_X will prove in subsequent request that the user is A, and that is a registered user within the system.

The transaction_id_X will only be available for N validation requests (see step 2), in this case N = 3 since we have partitioned the 2MB file in 3 chunks of 1MB each. Before the execution of step 2, node A will request a contract (see section 3.1) from node B. Once node A receives the contract from node B, it can safely proceed to step 2. The three store challenges are going to be solved (one per store request) and signed with the private key of A. This is used in order to be sure that the transaction_id_X is not re-used by Node B.

**Step 2: Request to store (Through the Anonymous Network)**
Node A will create a challenge C associated with the partitions of the file UUID that is trying to store in different peers. This request will be sent from Node A to B and it will be encrypted with the Public Key of B. The encryption is essential, otherwise an intermediate node could simply store the challenge_solution_C, which will give it the authority to retrieve and/or delete the file at any given point (see section 3.4 and 3.5)

Request{
  type: "Store",
  uuid_partition: "UUID_X",
  tr_id: E_{k_{CLS}}(transaction_id_X; size:1MB; E_{k_A}(solved(sc1)) ),
  challenge: (solution_C, E_{k_A}(challenge_C)),
  info: E_{k_A}("1 MB of information to store")
}

The challenge and the content of the file are going to be encrypted with a symmetric key algorithm. We prefer to encrypt these data with a new randomly generated secret key because the encryption should be more efficient than most public-private algorithms. Furthermore, it will make it easier to share files by just sharing this secret key with other peers.

After B receives the request from A, it needs to check with the CLS if the transaction_id_X is actually valid. The following request will be send from B to the CLS through the anonymous network.

Request{
  type: "is transaction_id_X valid?",
  tr_id: E_{k_{CLS}}(transaction_id_X; size:1MB; E_{k_A}(solved(sc1)) ),
  H(chunk_content); H(UUID_X);
}

13
After CLS checks whether the transaction_id_X is valid or not, it will reply to B with the following response. Note that CLS knows that the transaction_id_X belongs to node A. However, it does not know where that data is going to be stored since the request arrived through the anonymous network. Preserving the anonymity feature we claimed to support.

Response{
  type : "Transaction ID validity",
  req_validation :
    E_{\text{cls}}([H(chunk_content); H(UUID_X)]),
  answer : "It is valid"
}

The request validation field is the one that we are going to use in the Receipt (see step 3). We calculate 2 different hash values, because one file can have the same content (so the hash will be the same), but two files can not have the same chunk id (UUID_X). Since the request validation is digitally signed with the private key of the CLS, then node B can be sure that the response is actually from the CLS.

If the request is valid, then B can proceed to store. After the first attempt of storing the chunk UUID_X, the transaction_id_X will only be valid for the next N = 3 - 1 = 2 store operations.

Step 3: Send Receipt (Through the Anonymous Network)

If there are no errors in the previous steps, then B will send the receipt of the transaction to node A.

Receipt{
  file_uuid : H(UUID_X),
  file_hash : H(chunk_content),
  valid_until : timestamp,
  req_validation : E_{\text{cls}}(M),
  digital_signature : E_{\text{pub}}(H(receipt))
}
Where $M$ is equals to $[H(\text{chunk\_content}); H(\text{UUID}_X)]$. Finally, $A$ will update the encrypted metadata that is stored in CLS. This metadata is encrypted with $A$’s public key and it indicates where was the data stored, the symmetric key and the receipts obtained from the peers. It is important to mention that in this protocol for storing a file, node $B$ does not know that node $A$ is the one requesting this action. This is due to the fact that the communication happens within a single session of the anonymous interaction.

### 3.4 Retrieve a File

As illustrate in Figure 3.2, in order to retrieve a file node $A$ will have to repeat step 1 and 2 for every chunk and after that it will be able to reconstruct the file.

**Step 1 (through the Anonymous Network).**

From $A$ to $B$ : Request to retrieve $\text{UUID}_X$

Request {  

Figure 3.2: Retrieve Diagram

```
From B to A: Solve challenge \( C \) to prove that you own chunk UUID \( X \). Challenge \( C \) is the same challenge that node A created before storing the file, and it was sent it to B encrypted with \( K_a \).

Request{
  type : "Solve challenge",
  challenge : \( E^a(\text{challenge}_C) \)
}

**Step 2 (through the Anonymous Network).** A will solve challenge \( C \), and send it back to node B.

Response{
  type : "Challenge solved",
  challenge_solved : \( E^b(\text{solved}_\text{challenge}_C) \)
}

From B to A: If the challenge was correctly solved by A, then B can send the chunk UUID \( X \) to A.

Response{
  file_uuid : "UUID_X",
  info : \( E^a(1\text{MB previously stored in B}) \)
}

### 3.5 Delete a File

There are two different ways that a file could be deleted. On one hand, it can be deleted by a given node due to timestamp expiration. On the other hand, it can be deleted by request. In both cases it should be enough that any of the nodes involved in the deletion process send a message to the CLS, with the receipt and the node that is freeing space in its own hard drive. The CLS will proceed to update the quota of the Node.
We do not take too many safety measures in the deletion process against possible attacks. For instance, node B can fake that node A deleted a given file when node A did not. The reason why we do not take precautions is because there are no incentives for node B to bluff about this action; it will not create any advantage for its reputation. On the contrary, breaking the protocols will decrease the reputation of a node. As it is shown in the Figure 3.3, the following three steps are necessary to delete a given chunk by request.

**Step 1 (through the Anonymous Network).**

From A to B: Delete UUID_X

Request{
  type : "Delete chunk",
  file_uuid : "UUID_X"
}
From B to A: Solve SKₐ(challenge C)

Request{
  type : "Solve challenge",
  challenge : Eₐ(challenge_C)
}

Similarly as in the retrieve process, challenge C is the same challenge that node A created before storing the file, and it was sent to B encrypted with Kₐ.

**Step 2 (through the Anonymous Network).**
From A to B: A will solve challenge C, and send it to node B.

Response{
  type : "Challenge solved",
  challenge_solved :
    Eₐ(solved_challenge_C),
  status : "Successfully deleted"
}

From B to A: Compare with stored answer, if the solution is correct then we can proceed to delete the chunk. After that, we should send the deletion result status to A.

Response{
  type : "Deletion status",
  status : "Successfully deleted"
}

**Step 3 (through the Anonymous Network).**
From A to the CLS: Encrypt the new metadata and update it in the CLS (1MB deleted from node B).

After step 3, node A should also notify to the CLS that the file was deleted in order to update the storage capacity of B. Sending the receipt of the transaction and
indicating the node that it is freeing space in its own hard drive should be enough. This step should also be done through the anonymous network.

In this case, there is no special motivation for node A to bluff about the deletion of a file; consequently we do not take any special precautions in this case. If the CLS needs to prove that a file was actually deleted, then it can request for the receipt and test it for itself.

3.6 Share Private Files

The protocol for sharing private files is very straightforward. Assuming that node A wants to share a private file with node B, the procedure will be the following.

**Step 1.** We assume that Node A has the public key of B, and node B has the public key of A.

**Step 2.** When storing the file, node A created a symmetric secret key $K_a^s$. Now, node A needs to share this $K_a^s$ with node B.

**Step 3.** We take advantage of the public-private key infrastructure that we already have, and shared the symmetric secret key $K_a^s$ between the nodes. The following messages goes from node A to B, encrypted with the public key of node B.

```javascript
ShareKey{
    type : "sharing_key",
    value : $K_a^s$,
    files_stored_in_nodes : {B,C,D}
};
```
Step 4. At this point, node B is be able to retrieve the chunks from nodes B,C,D. Because according to the retrieving protocol it only needs to solve the challenge, which is encrypted with $K_a$.

3.7 Share Anonymous Public Files

In order to store and retrieve anonymously public files we will need to repeat the process of storing a file (see section 3.3) with two main differences. First, in this scenario there is no need to encrypt the information. And second, the challenge $C$ will not be encrypted, because the chunks will not belong to a specific node.

An essential part of the system is how to publish these files. We propose two different methods to do this. In both scenarios we might need to rotate the anonymous files through a different set of nodes every certain period of time so they do not remain in the same nodes until their deletion. This will help to prevent some of the attacks described in this document (see section 6). Assuming that node A wants to share anonymously the file, the options are the following.

Option 1: (through the Anonymous networks) Send a message from node A to the CLS showing which file we want to publish and where it is stored.

```
Request{
  type : "publish_file",
  fileUuid : "UUID",
  filesStoredInNodes :
    {B:UUID_X, C:UUID_Y, D:UUID_Z}
};
```

The CLS will have a list of anonymously public files links in order to make it accessible to everyone.
Option 2: (through some other secure channel) Another option for A if it does not want to publish in the CLS could be to share a the file with some users through some other means. The link or the package could include some embedded code like JavaScript, which will download the files locally for any user.

In order to retrieve an anonymous public file the user will request it through the Anonymous Network. It is important for security purposes that a user does not reveal its identity when downloading this type of file.

For deletion of public anonymous files, we are going to follow the approach of the Free Haven project [29]. The authors claim that the faculty to revoke files can put the original publisher in danger. Therefore, the public anonymous files are going to be deleted only after the timestamp expires. For the same reason, we consider a rotation of chunks between the nodes, so the peers holding a piece of anonymous data can avoid being attack.

3.8 CLS Database Encryption Scheme

Some database system designs encrypt data with a deterministic approach, such as the servers public key or with the same symmetric key all of the users data. This approach could potentially leak which cipher-text correspond to the same plaintext. With the architecture we are proposing this encryption scheme is not acceptable due to the fact that the CLS will be vulnerable to some attacks.

First, if the private key of the CLS gets compromised (same thing with the symmetric key), all of our users data will be also compromised. We do not want to have this type of single point of failure in which the entire system security could be in danger. Furthermore, encrypting data with that technique will typically mean that
we trust the CLS (the administrators will be able to decrypt users information), when we should not.

In addition, a malicious administrator can also infer same knowledge by studying data from different users. This is because deterministic encryption schemes allow the administrators to do equality checks. Therefore, they will be able to perform equality joins, group by, count, distinct, among others aggregate queries [26]. The administrator could try to predict where is the data stored for some users using some of the information that the system could provide. For instance, the nodes with higher reputation are more likely to store more data for different users. If nodes B, C and D have the highest reputation in the system; by scanning the database it could be possible to infer that Node A is storing in nodes \{B,C,D\} because many other nodes are probably doing the same and the encryption of the plaintext in a determinist encryption scheme \{B,C,D\} will always produce the same cipher-text.

When storing data in the cloud the principal concerns are security and confidentiality. Organization and users may trust specific cloud providers to operate storage services. However, in some scenarios cloud providers may not even trust their own employees [12]. One example shown in [12] is a confidentiality violation from database administrators of several banks from Switzerland who sold clients information to tax authorities from Germany and France [23].

These are the reasons why we decided to run all of the application and encryption logic on the clients, and not on the server side. Everything that is stored in the peers (except public un-encrypted files) or in the CLS is encrypted with the public key of a node or a symmetric key randomly generated by the node.
3.9 Anonymous Network

Deploying the service on the Internet could bring thousands of users into the system, which should enhance the security of the communication channels. On the other hand, in an intranet the environment is a lot more manageable; and we assume that we can trust at some level in all of the peers and that they will not try to take advantage of the system by making some of the attacks listed in this document. PayWithBytes can and should be used in these two different scenarios.

Because of this dual functionality of PayWithBytes it seems natural to use an architecture similar to the Tarzan project that was implemented at the IP level and provides NAT (network address translation) to forward packets outside of an intranet. Tarzan provides anonymity using a layered encryption and multihop routing. The node trying to communicate will choose a series of peers (belonging to our network) randomly, an send the data encrypted in a cascade of mixes, like a Chaumian mix [14].
Chapter 4: Consistency of the System

There are some aspects in which every peer and the CLS will have to be involved in order to maintain the system consistent.

4.1 Proof of Storage

The typical scenario where this protocol will be useful is the following. A client has been hacked to report that it is storing a few terabytes for fellow users; however, when the time comes to recover the data, the client is not really holding the data. This is the reason why we developed the proof of storage protocol.

In brief, the example is the following. Node A has sent a fragment UUID_X to B. After a while, A wants to check if B actually holds that given fragment. Every client will periodically send a proof of storage to the nodes that are holding data for them.

If the node is able to solve the challenge of the proof of storage then we can assume that it is actually holding the given fragment of the file. On the other hand, if the node is not able to solve the challenge then we can assume that the file has been corrupted or that the node is lying about holding the given fragment. In any case, we will report this to the LS so we can decrease the reputation of this node.
If the node B is down for some reason, after N retries on some X number of days we should also assume that the file fragment is lost. The idea of the challenge we introduced was motivated by the BitCoin proof of work [25].

**Step 1 (through the Anonymous Network)**

- From A to B: The request indicates that A wants to send a Proof of Storage to B. The challenge to solve is the calculation of the MD5(pre_string_to_add + content of UUID_X + post_string_to_add)

  Request{
  type : "proof_of_storage",
  file_uuid : "UUID_X",
  pre_string_to_add : "UUID random no. 1",
  post_string_to_add : "UUID rando no. 2"
  }

**Step 2 (through the Anonymous Network)**

- From B to A:

  Response{
  type : "solved_proof_of_storage",
  value : "solution"
  }

If the result of the Proof of Storage is wrong, then after some period of time node A will have to send the Receipt to CLS in order to make the reputation of B decrease.

On the other hand, if the result is correct then Receipt will have to be updated with a new timestamp. This means, that the partition of the file will be kept alive for a longer time after every successful proof of storage.

### 4.2 Nodes Leaving the Network

Since the CLS does not have any specific information about where is the information of a given node stored, then it is impossible for the CLS to make a clean up after a given node did not connect to the system for a large period of time.
The responsibility of the clean up belongs to the peers and that is why every file partition has a timestamp. If the timestamp expires then the node can safely delete that file. It is important to mention that every proof of storage will update this timestamp, extending the lifetime of the file.

4.3 Users Quota

There are two types of quota in every node. On one hand we have the remote storage quota; which is the amount of data that the node has already stored in some of the peers. In order to supervise this quota, we count every Store Request (see section 3.3) and we reduce the amount of storage by the size of the request. On the other hand, we have the local storage of a node; which is the amount of data that the peers are able to store in the given node. After a Store Request from node A to node B, B will send a Receipt to A. This receipt will be send to CLS, after some random period of time. This is due to the fact that we want to avoid timing attacks; which can make it easier for an eavesdroper to realize who is storing data at certain nodes.

The capacity of a user will decrease after every successful remote store of a file. At some point, the capacity could become zero; in that case the CLS will not allow executing any Store Request anymore. As we can see in the Store Request protocol, CLS is able to deny the request to a given node at any time. This will be useful when the node reaches its maximum capacity. The other case occurs when node A deletes a file that it is stored in node B. In this case, Node A should notify that the file was deleted from node B, and the CLS will be able to increase the storage capacity of node B.
4.4 Reputation System

The reputation of the nodes is going to be very simple and it will follow some rules. The reputation of every user will be maintained at the CLS. However, every node will be responsible for enhancing the accuracy of the reputation system.

Increment reputation

The uptime and bandwidth of the nodes are going to be measured by the CLS and the peers. For the uptime we want to measure this according to some user specific information. Node A may want to stay online during office hours in San Francisco, California. However, Node B may want to stay connected 24/7.

The utilized percentage of the total storage quota assigned for other peers it is also valuable information about the reputation of a node.

Decrease Reputation

Every failed Proof of Storage will be send to the CLS to keep track of the behavior of every node. Successful Proof of Storage requests are not going be send to the CLS. The nodes are expected to behave correctly.

Deleting files before the timestamp expiration. This can be proved thanks to the Receipt (see section 3.5)

Not being online during the time period of the day provided.
Chapter 5: Implementation and Availability

PayWithBytes is still under development, although a basic prototype of the system is ready. For the client side development we used Java, Jetty Http Client/Server [3], and Junit [4] as a testing framework. The database has a non-SQL design (key-value pair rows for every chunk) on top of SQLite [8] in order to make the distributed queries achieve the best performance possible when retrieving content.

We have used third-party libraries for the encryption algorithms and also for creating redundancy when splitting a file in chunks [27]. The encryption algorithms are AES-256 for symmetric key encryption [17] and RSA for public key encryption [28]. To create message digest we used SHA-1 [9].

It is worth mentioning that most of the components of the implementation communicate with each other through interfaces. That means that most of the components shown in the figure 5.1 can be easily replaced with different technologies. This is important because in the future we might need to modify some of them. For more details about the implementation see section 5.2.

The component called Commands on figure 5.1 contains the code for the six commands that can be used through the command-line.

1. Install

2. StoreCommand -f test_file.txt -n 6 -k 3 -t Public
3. RetrieveCommand UUID
4. DeleteCommand UUID
5. Recover
6. RunServer

The first command will install the application in the local machine, it will create the database with the necessary tables, and it will generate the Public-Private key pair for the user. Next, the StoreCommand will store your file remotely, the parameters are -f for the file path, -n and -k for the FEC algorithm and -t that defines the type of the file (Public or Private). For the RetrieveCommand and DeleteCommand there is only one parameter, the id of the file. The Recover command will restore all of the metadata information from the CLS into the local database. Finally, the RunServer command runs the HTTP Server that listens for other peers requests.
The implementation of the CLS is very basic, since it just provides a couple of features. We used Java Spring [7] and a PostgreSQL [6] database, we decided to use this open source database because even though this is a data-center less system the database will need to achieve some levels of scalability. Although the idea was to introduce a proof-of-concept implementation, we have tried to design the base of a solid and scalable project.

5.1 Results

In order to developed and test PayWithBytes we have used 8 different computers for the clients and 1 for the server (CLS). We obtained these machines from the Ohio Supercomputer Center [5]. Each machine has 12 processors and multiple cores.

The experiment that we found interesting to show was the comparison of two different implementations. On one hand, the sequential store and retrieval of content, and on the other hand a multithreaded implementation of the store and retrieval protocols.

In the multithreaded version of the store request, we created a new worker thread for every chunk (N threads). Each thread takes care of encrypting the data (if necessary) and also to store that chunk in a given peer. In addition, for the multithreaded version of the retrieve request, we create K threads in order to retrieve the content from the peers. Every time one of these K threads fails (maybe because the host is not connected to the network) we launch a new thread from the (N - K) group of remaining chunks. For all of these tests, we used N = 8 and K = 6, all of the files were of type Public, so none of them were encrypted.
Figures 5.2 and 5.3 illustrate the performance improvement of the multithreaded version compared to the sequential version of storing and retrieving. The interesting thing is that as the files get bigger the improvement percentage goes down. This is due to the fact that this is an I/O bound application. We are running all of these threads on the same local machine, on the same TCP stack, which means that the communication is limited by the throughput of the network. As we can see in Figure 5.3 the performance improvements are different compared to Figure 5.2, this is due to the fact that for retrieving we only create K threads, not N like in the store implementation. Consequently, it makes sense that they look different; however, the conclusion we made from both figures are the same.
5.2 Availability

PayWithBytes is under development as an open source project available in a public Git repository provided by BitBucket [1]. Even though the basic prototype is currently working, we are still working in these three different projects to improve them. The client, the CLS and finally a wrapper of a Reed Solomon implementation developed by [15] that adjusts their implementation to our needs and provides a high level API.

http://code.client.paywithbytes.com

http://code.server.paywithbytes.com

http://code.jfecwrapper.paywithbytes.com

For an update status of the project and feedback please visit the PayWithBytes web site.
http://www.paywithbytes.com
Chapter 6: Possible Attacks

The following subsections describes a series of technical and non-technical attacks on the system. The first 3 of them are based on the FreeHaven project [29], which describes similar attacks. Even though the architectures are different in many aspects, the prevention methods for these 3 problems are similar.

6.1 Physical

The most basic way to make a resource unavailable is to destroy nodes holding data. There is no harm in destroying a single node since we are going to split the files in N chunks and add redundancy such that any K of these chunks are going to be enough to rebuild the file (where $K < N$).

In order to destroy at least K of these nodes the attacker will need to know which ones are exactly those K nodes holding a given file. For private files this is not a problem, since only the owner of the file has this information. The CLS have this information but is encrypted with the nodes public key. For shared files only the users sharing the file know where the data is stored.

The only scenario in which this could be a problem is with anonymous public files, because everyone can get the information of where is the data stored. This can lead
to a targeted attack against those nodes. That is why we proposed to make these chunks rotate between a different set of nodes.

6.2 Legal Actions

This attack refers to prosecute the owner of an account registered in the system. This opens up a broad series of questions about trust and storing copyrighted or illegal data. In our understanding, a user should be able to deny knowledge about holding illegal data in its computer. One of the purposes of our research is making the Internet belong to the users again, and not to organizations spying on our data. On the other hand, we do not want this to become a massive protocol to share illegal information. We wanted to mention this, even though this legal action attacks are out of the scope of our research.

6.3 Denial of Service

A denial of service attack is unlikely to happen if we base the anonymity of the network in a system similar to the Tarzan Project. On the grounds that every node registered in the system will also have the responsibility of participating in the forwarding of encrypted messages. If a given node within the system tries to perform a denial of service attack we could just stop providing services to that particular user.

6.4 Acquire Numerous Nodes

One of the goals of the system we are proposing is to provide privacy to the users against important and powerful organizations that want to spy on user’s data. The problem is that these organizations can register many different users in the system, and each one of them with high storage capacity. Consequently, they could own
not just chunks but also complete files. Furthermore, if they eavesdrop on their connections they will be able to do some of the attacks listed in this document. There is nothing we could potentially do in this case to be safe, except to attract a vast amount of users to PayWithBytes. This will make this attack particularly complicated and unreasonably expensive.

6.5 Anonymous Network attack

There are many different attacks that can occur within an anonymous remailer protocol. The authors of the Tarzan project provide a list of all known attacks against mix-nets. Some of these attacks can reduce the anonymity of the nodes by linking senders and receivers of messages. An eavesdropper may trace messages by observing the network traffic, compromising nodes, compromising private keys, using timing attacks, among others. For more detail refer to the Tarzan project [19].

6.6 Fake Receipts

We cannot assume that we have a secure-installation of the software in every node for two reasons. First, the project will be open source. In addition, even if decide to make the code proprietary there is always a method to decompile an application (even if it is obfuscated) and try to modify the source code in order to take advantage of the system. Therefore, an intruder with a deep understanding of the system architecture and high hacking skills can decompile the application and modify the source code and then it could generate fake Receipts (see section 3.2).

However, there are not any advantages of creating fake receipts. This is due to the fact that every Store Operation is going to be approved and certified by the CLS.
Consequently, a fake Receipt could have the entire information fraudulent, except the
digital signature of the transaction_id by the CLS. For this reason, in order to create
fake receipts the intruder will have to make valid store operations, which will mean
that its own storage quota will decrease. In conclusion, the incentive of making this
hack is zero.

6.7 CLS and Nodes Compromised

If the CLS gets compromised the system and all of the data stored in the peers
will remain secure thanks to the pseudo-zero trust architecture. However, if the CLS
and many nodes in the system are compromised at the same time there are different
attacks that can be achieved. For example, if node B and CLS are compromised they
can eavesdrop to any incoming store request. After that, they can easily link a given
user with the file that is going to be stored. This is because the CLS knows that a
transaction_id belongs to a given user. Furthermore, node B knows the UUID of the
file, so together they can figure it out who is storing data in that particular node.
This is particularly important when publishing anonymous public files, in the case of
private files they are going to be encrypted so this should not be an important issue.

An important thing to mention is that if there are many nodes in the network,
intruders will need to compromise several nodes if they want to target the attack to
a specific node. Additionally, the data that was stored previous to all of these nodes
being compromised will still remain secure.
Chapter 7: Conclusion and Future Work

7.1 Recommendation for Future Work

One of the goals of our work is to make a better use of resources, whether it is on an intranet or on the Internet. We believe that sharing free hard drive space in every computer on a network can make the backup and share of files a low-cost process. It could potentially help the environment, since the architecture of the system should be more energy efficient that having huge data centers like DropBox or Google Drive. Following this thinking we might also want to share other kind of resources in the future, like CPU time in the SETI@Home Project [11].

We will provide a more detail description about how much information a given node can store in its peers according to the amount of capacity the node provides for others nodes to store. The formula will depend in the amount of redundancy that the user wants to add to its data. The more redundancy the fewer amounts of data a user will be able to store in its peers. However, redundancy will make user’s data safer because more peers can fail and they will still be able to recover their data.

It could be an interesting feature to add geo-location information about the nodes in the system. Redistributing data in different geographic zones can make the system safer against natural disaster. However, this redistribution of data could affect
performance when storing and retrieving the files; which is why we do not plan to implement but it could be interesting to have this as an option for the final users.

One of the next steps in this project is to develop a user interface, so we can make it easier to install and use PayWithBytes software. The two options we are considering is developing a set of native user applications for every operating system, it could be useful to make an HTML5 application so we can reuse the code in different platforms. The other option is to implement the UI using FUSE [2], which is a file system in user space. This will allow the users to simply mount a PayWithBytes filesystem within a given folder and start using it. The disadvantage is that this is only compatible on unix-like systems, but it is an interesting solution.

In addition, after releasing an stable version of the first prototype we also want to support some other key features that many of the storage system provides, such as incremental backups and versioning [13].

7.2 Thesis Summary

PayWithBytes is a collaborative and anonymous storage service that provides the services of backup, sharing and publication of files. The system design is based on three important features. First, a pseudo zero-trust collaborative architecture designed to trust no one. Second, address rewriting and multi-hop forwarding to hide the identities of paths source and destination based on Tarzan [19]. And finally, hard drive reutilization from every peer connected to the system.

Although there are currently systems that provide these services separately, PayWithBytes addresses all of these requirements at once with an efficient design. Supporting low
storage devices such as tablets and mobile phones, and the fact that we are introducing an economic storage service may sound irrelevant. However, we believe that creating a modern architecture that satisfies these common user requirements is important for one simple reason. A system with several users will provide the anonymity level that we desire. On the contrary, a system with few users will be weaker against the attacks described in this paper.

In addition, we predict that zero-trust architectures will become more relevant in the next years and that this feature will hopefully influence users to try a system like PayWithBytes. The conspiracy theories about governments and important organizations doing surveillance on the population are not only for fiction movies and books anymore. After the NSA leaks in 2013, it has been shown that this is actually happening and a lot of people are already aware of this.
Bibliography


[6] Postgresql: the world’s most advanced open source database.


