D-Sync: A Distributed, Automatically Synchronized File Hosting System

Paul Elliott
University of Oregon
paule@cs.uoregon.edu

David Stevens
University of Oregon
dstevens@cs.uoregon.edu

***Abstract*—File hosting systems such as Dropbox and Google Drive are widely used Internet tools for data storage and backup, collaborative work, and file sharing. These systems, while increasingly popular, rely on centralized storage of data, which pose inherent privacy concerns for certain types of users and working groups. In this work, we present D-Sync, a distributed file hosting architecture that allows users to store and share data from workspace directories on local machines. All workspace directories are updated and synchronized whenever an offline user connects to the system, and whenever an online user makes a change to his workspace directory. In evaluating our design, we show that D-Sync scales reasonably with workload and outperforms Dropbox with respect to total synchronization time. Furthermore, we show that our system maintains data consistency despite arbitrary node failures.**

I. Introduction

Of the wide range of today's Internet tools, file hosting systems such as Dropbox and Google Drive are certainly among the most popular. These systems offer convenient (and often cheap) means to store and backup data, in addition to making that data available to any device with Internet access and the proper credentials. The ever-growing ubiquity of Internet access and the advent of cloud storage have helped these systems flourish, allowing them to serve billions of files for millions of users.

As the most popular file hosting system on today's Internet, Dropbox leads the way in this growing domain. Since January 2010, Dropbox has steadily expanded from about 4 million to 100 million users, and increased its usage rate from roughly 250 million to 1 billion files saved per day [3]. Stop for a moment and consider the sheer magnitude of these statistics: even roughly averaged, this means that Dropbox users collectively save over 1 million files *every two minutes*.

This frequency of usage points to a key aspect of Dropbox's popularity: its capacity for collaborative work. The system allows its users to share data with other select users in the so-called working group, and also keep a local copy of the shared data on any owned machine or device. Most importantly, Dropbox synchronizes quickly and automatically across local and cloud replicas by employing **delta encoding**, in which only the *changes* to files are saved and uploaded. The efficiency of this synchronization strategy makes Dropbox a convenient tool for both synchronous and asynchronous collaboration among working groups. The benefits of these features have not gone unnoticed, and continue to attract attention [1], [5].

When we consider data privacy in Dropbox and other cloud-based file hosting systems, however, we see that not everything is silver lining. Privacy issues with cloud services have recently fallen under scrutiny by the security community [15], [14], and Dropbox has not escaped this critical eye. Aside from a few exposed security issues [10], [11], the top file hosting system has been criticized for its policies on data discretion. In order to facilitate delta encoding, Dropbox maintains encryption keys to calculate hashes for saved files. When a change is detected in a shared folder, Dropbox hashes blocks of the data and uses those hashes to see if they are stored *anywhere* on *any* of its servers. Using this information, it only needs to upload the new chunks. This seemingly clever trick can enable a gigabyte file to almost 'magically' upload in seconds. This is a huge boon for the company, as it allows them to offer more space to clients at almost no cost if the client is sharing data shared somewhere else.

Unfortunately, this optimization comes at the cost of security. One simple attack is immediately obvious: Imagine an agent, Alice, that wants to know if a given copy-writed media file has been shared on Dropbox. She could simply create a Dropbox account, add the file to her shared folder and watch to see how quickly it uploads. If it uploaded far too quickly, she immediately knows that it has already been uploaded and hashed somewhere else. She could then obtain a subpoena to require Dropbox to disclose which users have this file. Since Dropbox does not let users control how their files are encrypted, and since they have already hashed the location of every block of this file, they can easily determine who has these files and would be required by law to report them. [4].

One solution to the privacy issues of centralized data would be to distribute data storage across users' local machines. The file hosting system itself would remain blind to this data, and merely marshall encrypted updates between user machines to handle synchronization. For such a distributed file hosting system to be successful, it would have to achieve the following design goals (in addition to privacy):

- **Accuracy:** The system must be able to properly synchronize all updates across all *online* users. In other words, whenever a user needs to pull data from the system, it must pull a completely up-to-date copy. Furthermore, a user should not push outdated data to the system.
- **Transparency:** The system itself should largely be transparent to the user. To facilitate transparency, the system must enable automated and asynchronous updates for

shared data. When a user is online, changes should be pushed immediately after they occur. In addition, a user should not have to block and wait for the system to update.
- **Scalability:** The system must be scalable with respect to increasing workloads and working group sizes.
- **Efficiency:** The system must be efficient with respect to the latency of message and data transfers. In particular, the amount of extra information (beyond file data) that is passed between networked entities should be small.

In this work, we present D-Sync, a distributed and automatically synchronizing file hosting architecture that addresses the privacy issues of centralized data storage. D-Sync addresses these concerns by distributing data in workspace directories across user machines, called **clients**. Clients make changes to the files within their local workspace, and changes are automatically pushed to the system. A **broker** program marshalls these changes and ensures that all online local workspaces are updated, but remains blind to the actual data itself. In this manner, working groups can collaborate efficiently over the Internet without the privacy concerns of centralized data.

We evaluate our system's performance with respect to the latency of synchronization across different networking scenarios. In particular, we measure the total time it takes to synchronize all clients with variable working group and update file size. We also benchmark our system against Dropbox's performance. Our comparative analysis shows that the D-Sync architecture scales reasonably with larger working groups and workloads, and furthermore outperforms Dropbox in several scenarios.

The contributions of this work include:

- *D-Sync: a distributed file hosting architecture.* automated synchronization of files and protects the privacy of its users. Our architecture includes schematics for its major components (clients and brokers) as well as a communication protocol and automated synchronization strategy. While similar proposals exist, to the best of our knowledge our work is the first to posit such an architecture as a privacy-preserving alternative to conventional file hosting systems.
- *An evaluation of distributed file hosting architectures.* In evaluating our prototype system's performance and accuracy, we provide a first look at how such architectures may perform on various network scenarios. Furthermore, we conduct a comparative analysis of Dropbox's performance, and present the results in conjunction with our prototype evaluation.

The remainder of our paper is organized as follows. In Section II we highlight previous work related to our project. Section III covers our system design, including its major components and design decisions. We evaluate our system in Section IV, specifically analyzing its performance and synchronization correctness. In Section V we discuss our work and future directions, and finally we conclude in Section VI. Section VII contains a brief description of our code and division of labor on this project.

II. Related Work

The topics of synchronization and replication strategies have been considerably researched within the general domain of distributed file systems. Much of this work has focused on replica maintenance for the purpose of data availability [12], [6], [8], [2], [7]. Pu et al's work discusses various replica control mechanisms for maintaining epsilon-copy serializability (ESR), a correctness criterion that allows asynchronous maintenance of mutual consistency for replicas [12]. However, their evaluation does not extend to specific domains of distributed file systems. Their research also lays valuable foundations for evaluating replication strategies in specific domains (i.e., [7]). While we also build on Pu et al's groundwork, our emphasis on automated synchronization and privacy constitute a distinct focus from other follow up research.

In another similar work, Chun et al evaluate replication strategies for storage systems distributed over the Internet [2]. Their research focuses on data durability, the assurance that data put into the system is not lost due to disk failures, and claims that this property be held more important than conventional availability. While their work uncovers a valuable property that should be widely applicable in distributed storage systems, our research focuses specifically on availability as a necessary prerequisite of automated synchronization.

To the best of our knowledge, our we are the first to present a tested design for a distributed file hosting system optimized for large-scale, real-time collaboration and lacking any central storage.

III. System Design

Here we cover the design of our system architecture. We first state the overall scope we used to work toward our design goals of accuracy, transparency, scalability, and efficiency. We then describe the main components of our system–namely, the client and broker programs, followed by the communication protocol that these entities use. In these sections we will highlight specific design decisions that we needed to address in order to achieve our stated goals, and explain how we handled them.

A. Scope

Before we could begin designing a distributed file hosting architecture, it was necessary to define the scope for our work. Our assumptions for this work are as follows:

- *Connection reliability:* The individual components of our system connect via TCP stream sockets in order to pass messages and data. This precludes reliable transmission of data, with failure detection and subsequent retransmission. While TCP handshakes will increase the latency of message passing, reliable transmission allows us to focus on the accuracy of our system.
- *Conflict resolution:* Automated synchronization necessarily implies instances of conflicts between closely committed updates. Previous work ([13], [9]) has been done

on conflict resolution strategies in distributed domains, generating several viable solutions. We rely on these promising schemes in place of proposing our own solution, which is beyond the scope of this work.
- *Security:* Our work falls in the domain of networked applications, and therefore is subject to a litany of network security attacks. The security community has addressed such threats in a body of literature that is too vast to properly reference. Instead we indicate where we leverage secure hashing algorithms (SHA) and public key infrastructure (PKI) to facilitate the integrity and privacy of our system, and leave the rest to the capable hands of security researchers.

Having declared the scope of our project, we now detail the D-Sync architecture. The main components of this system are clients and brokers, networked via TCP stream sockets. In the following sections we will describe the primary components and actions of each entity, as well as any design decisions made along the way.

B. Major Design Decisions

The first design decision that we needed to cover was the synchronization model. We had a number of options, but to best ensure that our system maintained data consistency we chose Read Once Write All (ROWA). In this model, updates are pushed to all clients in the working group at once. When a client needs to pull data, the system merely fetches it from the nearest client.

In addition to selecting a synchronization model, we also needed to specify a communication model for our system. We chose an asynchronous communication model, allowing users to collaborate without blocking on updates.

With the major design decisions covered, we next turn to the client and broker programs.

C. Client

The client program is essentially a thin local program for handling changes to a user's local workspace. On the one hand, it must watch the client's workspace and detect changes to files and folders. In order to detect changes, the client must maintain local storage tracking the current status of the workspace.

One of the first design decisions we had to handle on the client-side is the discovery phase: how does a client connect to the rest of the D-Sync architecture? We note that given that working groups who would potentially use this system have inherent privacy concerns, it is reasonable to assume that they would have interest in a secure system. Therefore, users must be invited to the working group and the system in general. An invited user could obtain connection information for a specific broker to which he/she can initially connect. Such offline information would also include any public keys needed to verify the broker, and any symmetric keys needed for encrypting data.

Once a client has access to the network, it can begin to collaborate with the working group. In the next sections, we cover the various actions that a client performs.

1) Initialization: In order to implement our ROWA synchronization model, we had to take a unique approach to initializing a client. When a client comes online, it immediately needs to check whether its offline changes are ahead of the system and push those changes. After this push, the system should make the client pull down any changes that it is behind on. We call this initial phase the **batch update**. This action enables our system to maintain data consistency for online clients, which we demonstrate in our evaluation Section IV-B.

2) Other Client Actions: The remaining actions of the client are fairly trivial. When the client detects an update to the local workspace, it sends the system a request (RQST) message containing a hash of the filename and the file's revision number. The client then waits asynchronously for an acknowledgement (ACK) from the broker. We cover the need for a request message when we discuss the broker in section III-D. Upon receiving an ACK, the client updates the revision number of the changed file, and pushes that file to the system.

The only other required actions of the client are to handle pushes coming from the broker. If the broker pushes down a change, the client should accept that change and update both its local workspace and the associated file's revision number. The client should not have to worry about calculating revision numbers itself–that logic is handled on the broker-side.

We next turn to the broker, the primary component of our system. As with the client, we describe the broker's primary function and actions.

D. Broker

The broker is the computation-heavy component of our system architecture. It is responsible for marshalling data between clients in a given working group. The broker must handle the update requests of all connected clients in such a way that synchronization remains transparent to the user. Furthermore, while all data flows through the broker, the broker itself remains blind to the raw data itself by virtue of hash functions or crytography on the client side. Therefore, while the broker must maintain local storage itself to determine the whether client requests are fresh, this local storage need not be more than a dictionary of encrypted filenames mapped to revision numbers.

1) Broker Actions: The first action that a broker must handle is receiving a batch update from a newly initialized client. The broker must check whether each pushed update is ahead of or behind the system. Changes ahead of the system are pushed to the remaining clients. Otherwise, the broker pulls current changes down to the initializing client.

A broker must also handle the client's normal requests, though this is done in a similar fashion. If the requested change is ahead of the system, the broker pushes that change to all clients, otherwise it pulls down a change to the requesting client.

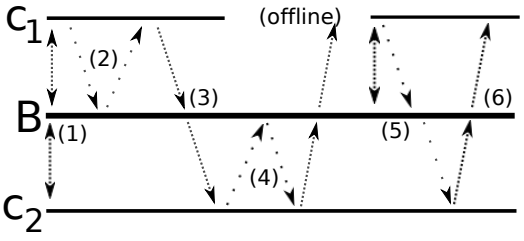


Fig. 1. **A typical timeline involving two clients C_1, C_2 and broker B.**

Having described both the client and the broker, we now cover a typical timeline of events occurring between two clients, C_1, C_2, and broker B as pictured in figure III-D1. In (1), both clients connect to the broker or are already connected at this point in time. C_1 sends a request to the broker in (2), which is acknowledged. C_1 sends out the actual change in (3), which the broker pushes to C_2. Around (4), C_1 goes offline and C_2 makes a request, is Acknowledged, and pushes a change. This change does not reach C_1, who is offline. When C_1 returns in (5), C_1 pushes its local workspace. The broker determines that C_1 is behind the system due to the change in (4), pulls that change from C_2, and in (6) updates C_1.

2) Distributed Broker: While the broker description above suggests a single broker or broker network, the broker functionality could easily be applied to a distributed manner. In such a architecture, each client would have its own broker program. Brokers would maintain connection information of all other connected brokers. In the event of a channel failure or broker crash, other brokers could try to reconnect to the disconnected broker using exponential backoff. In this manner, the broker network could prevent partitions of the network due to node failures.

In addition to network partitions, a distributed broker solution would also have to manage requests in a new way. When a given broker receives a request from its client, it would contact the other brokers to determine whether the change could be committed. Algorithms for consensus between distributed nodes would need to be leveraged in order to make this feasible. Once such an algorithm is put into place, however, a client could easily connect to this new distributed broker as is. The high level API implemented by the broker would remain the same, with the extra consensus logic remaining transparent to the client. We return the notion of a distributed broker in our discussion Section V.

IV. Evaluation

We evaluated our system with respect to performance and accuracy. For the former we measure the latency of key synchronization tasks across different network scenarios, and compare the results with a benchmark evaluation of Dropbox. For the latter we test specific use cases where synchronization could fail, and demonstrate that the system properly handles these cases.

A. Performance

In order for D-Sync to be a feasible architecture for distributed file hosting, it needs to perform comparably to existing systems. We therefore evaluate our prototype system with respect to the latency of updates. We not only show that D-Sync scales reasonably well, but furthermore outperforms Dropbox in every evaluation.

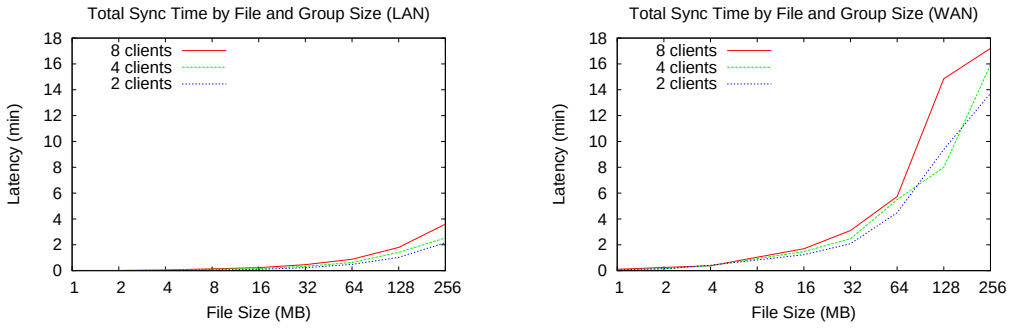
1) Metrics and Parameters: In order to evaluate the latency of our system, we measure the total time it takes for adding a single file of increasing size to propagate across all online clients. We focus on adding files since these updates will necessarily have the greatest latency, given that the system must propagate raw file data to all online clients. We measure the total latency as: (time stamp of when the last client receives the entire file) - (time stamp of when the sending client starts sending the file).

We define the parameters of our performance tests in the following manner. First, we evaluate two separate network scenarios: a LAN scenario with two computers networked over the UO Secure wireless network, and a WAN scenario with the same two computers on two separate home wireless networks. Within these two testing scenarios, we test additional parameters to help determine how our architecture scales. First, we evaluate working group sizes of 2, 4, and 8 clients. Should the system experience more than linear growth in latency as the working group size increases, then our system will not reasonably scale. Finally, we evaluate our system with file update sizes ranging from 2^n MB, where $0 \leq n \leq 10$. We do this in order to gain insight on how our system scales with respect to file size.

2) Setup: We set up our tests in the following manner. We designate one computer to run the broker program and the other to run the variable number of client programs. Though the client programs all run on the same computer, this is a reasonable simulation of multiple computers on the same network, as messages must travel to the broker and back. It also eliminates the variables associated with using machines of varying performance. In the evaluation of LAN sync, the broker is on the same network as the clients, whereas it runs on a seperate network when evaluation WAN sync.

To conduct a test we start the broker program, followed by the client programs. Once all clients are connected to the broker, we move a text file of the appropriate size into the workspace directory of a given client, and measure the total time for all other clients to receive the file. To ensure that our measurements are correct, we synchronize the local clocks of both computers using the network time protocol. We repeat the sending of text files in a serial fashion until all files have been sent.

To evaluate Dropbox, we use the following setup. First, we share a Dropbox folders between two users. As we only had 2 computers with write-privelages sufficent enough to install Dropbox, we were only able to evaluate the 2-client scenario. Once our shared folder was created, we sequentially added the same test files to one folder and used the same procedure



(a) LAN scenario.

(b) WAN scenario.

Fig. 2. **The total synchronization time for D-Sync across different network scenarios (shown in log scale).**

to time the latency between when the file was added on one client and when it finished writing on the other.

To prevent delta encoding advantages of Dropbox, and to simulate encoded data, all test files consisted of randomaly generated strings of characters that should be statistically impossible to exist on any of Dropbox's servers. Also, given the client-server nature of Dropbox, we have no Dropbox LAN scenarion. In the following section, we cover the results of these tests.

3) Scalable Synchronization: We first analyze the total time it takes to synchronize files across the LAN network scenario. Figure 2(a) shows this synchronization latency for variable working group and file sizes. From this graph, we see that total synchronization time follows a linear growth pattern. Furthermore, the latency increases only slightly with respect to increasing working group size. While we will need a benchmark before making a statement on whether this is scalable, for now we are satisfied that total synchronization time does not grow unreasonably.

Next we compare the results of the LAN network scenario to the WAN network scenario (2(b). The WAN data is considerably noisier, reflecting variable congestion and delay in the underlying network. However, it is still clear that total synchronization time still increases linearly with respect to file size. Most importantly, the data from these two networking scenarios show that increasing numbers of clients only contributes a modest amount of latency to our system's synchronization time. Therefore, we find the D-Sync architecture to be scalable with increasing working group size.

One can clearly see that latency for file synchronization increases significantly once we step out of a comfortable LAN scenario to the wide area network. But is this increase in latency reasonable with respect to existing solutions? To determine this, we compare these results with our results from our evaluation of Dropbox. The following graph shows that our system is remarkable more efficient than Dropbox, which takes almost an hour and a half to sync a 256 MB file, while our system takes a modest 5 to 15 minutes to sync over LAN and WAN, respectively. This clearly shows that the elimination of central storage dramatically speeds up syncronization time between any number of users.

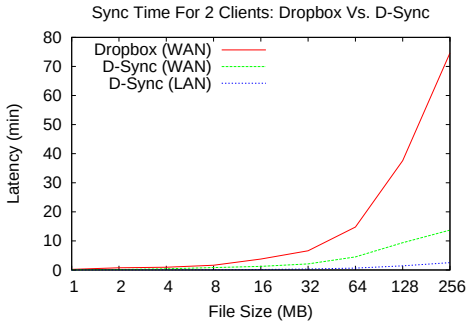


Fig. 3. **A comparison of Dropbox and D-Sync's total synchronization time for two clients across network scenarios.**

B. Accuracy

In addition to providing an efficient solution with low latency, it is important that our architecture provide data consistency across different scenarios. In other words, the system must properly synchronize all updates across all online users. In this section, we evaluate specific scenarios where where the replicas in a given working group could become inconsistent across clients, and demonstrate how our system handles these scenarios.

1) Client Departures: A client could depart from the system in several ways, the most common of which are going offline and crashing. There are three primary scenarios that could cause data inconsistency in the event of a client departure:

(1) the departed client could make offline changes to his local workspace;
(2) one or more online clients could make changes while the departed client is offline; or
(3) both the departed client and one or more online clients could commit conflicting changes.

In all of these cases, the system would have to properly update shared data when the departed client returns and does an initial batch update.

We preliminarily evaluate these scenarios in stress tests using eight clients and one broker in a LAN setting. We start each client and connect them to the broker, and begin

committing changes on various local workspaces. At random intervals, we take one or more clients offline and then bring those departed clients back after a fixed length of time (2 minutes). Throughout the test we continue to commit changes to the local workspaces of all clients, online or offline.

Our results from four such tests show that the initial batch update properly handles all cases of potential data inconsistency. Across four tests, we observe 31 instances of case (1), 26 instances of case (2), and 12 instances of case (3). In case (1), the departed client's offline changes are immediately pushed to the broker, who commits them to the rest of the system. Case (2) is similarly handled; the broker pulls up-to-date files from the nearest online client and pushes them to the returning client. For case (3), our prototype broker accepts the first offline change it receives, discarding subsequent updates with the same timestamp. While a better solution would involve merging or handling such conflicts in some manner, conflict resolution strategies are beyond the scope of this work.

Furthermore, we note that our system maintains accuracy even when other clients depart during a re-entering client's batch update. In these cases, the second departing client will also be updated upon return. In this manner, all online clients will maintain a current workspace for all online updates. These preliminary tests show that the D-Sync architecture keeps data consistent in the presence of arbitrary client departures.

V. Future Work

Due to limitations of available resources, there is more potential work to be done in evaluating D-Sync It would be interesting to evaluate our system over different network conditions. Our WAN evalaution was limited because we had only 2 machines that we were able to install our software on and therefore were only able to use 2 different LAN's at a time. Similarly, it would be interesting to evaluate the performance of Dropbox over different working group sizes to see how it scales with the number of users and compare that to our own results.

There are also many things that could be done to improve D-Sync. Although security of files was a significant motivator for our project, we focused on making a system that would be easy to make secure, but did not implement any encryption scheme, as this was beyond our scope. Another necessary inclusion would be conflict resolution for files with no unambiguous 'most recent' version. This could be done by either implementing some sort of version control system, requiring users to resolve conflicts in real time, or creating conflict files that users can resolve asynchronously. Again, while necessary for a user-friendly system, this was beyond our scope. Once conflict resolution and/or file encryption have been added to D-Sync, it would be interesting to evaluate what effect, if any, they have on the system's overall performance.

Lastly, while our system uses a distributed storage scheme, communication is controlled in a undesirably-centralized fashion. Luckily, as described in III-D2, a distributed version of our broker system could be implemented and, provided it implements the same API as our more-centralized broker, could be easily swapped in. This process would be completely transparent to the client, with the exception of any changes to performance. In this way, we have layed the groundwork and created a easily-extendible framework for any future improvements to the broker protocol. We have included the current version of our code, as of this publication, and the most recent version will continue to be maintained on our Github repository: *https://github.com/dzstevens/CIS-630*

VI. Conclusion

Cloud-based file hosting systems such as Dropbox are popular and heavily used, but raise privacy concerns associated with centralized data.

In this work we design and evaluate D-Sync, a distributed and automatically synchronizing file hosting system. Our architecture meets our design goals of accuracy, transparency, scalability, and efficiency. Finally, we evaluate our system's performance, and find it to scale well and outperform Dropbox in certain scenarios.

References

[1] Mercurial on dropbox.

[2] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of NSDI*, volume 6, pages 225–264, 2006.

[3] J. Constine. Dropbox is now the data fabric tying together devices for 100M registered users who save 1B files a day, 2012.

[4] J. Constine. How dropbox sacrifices user privacy for cost savings, 2012.

[5] A. Dachis. How to use dropbox as a killer collaborative work tool, 2011.

[6] O. Damani, A. Tarafdar, and V. Garg. Optimistic recovery in multi-threaded distributed systems. In *Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on*, pages 234–243. IEEE, 1999.

[7] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[8] S. Goel and R. Buyya. Data replication strategies in wide area distributed systems. Technical report, ISBN 1-599044181-2, Idea Group Inc., Hershey, PA, USA, 2006.

[9] W. Hurley and K. Habermehl. Collaborative model for software systems with synchronization submodel with merge feature, automatic conflict resolution and isolation of potential changes for reuse, Jan. 13 2004. US Patent 6,678,882.

[10] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security*, volume 8, 2011.

[11] D. Newton. Dropbox authentication: insecure by design, 2011.

[12] C. Pu and A. Leff. *Replica control in distributed systems: as asynchronous approach*, volume 20. ACM, 1991.

[13] D. Shakib, S. Norin, and M. Benson. System and method for distributed conflict resolution between data objects replicated across a computer network, July 28 1998. US Patent 5,787,262.

[14] C. Soghoian. Caught in the cloud: Privacy, encryption, and government back doors in the web 2.0 era. *J. on Telecomm. & High Tech. L.*, 8:359, 2010.

[15] M. Van Dijk and A. Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *Proceedings of the 5th USENIX conference on Hot topics in security*, pages 1–8. USENIX Association, 2010.

VII. Appendix

In this appendix we describe the division of labor for this project and reference our code base. Both Paul and David collaborated evenly on designing the system, making design decisions, and researching related work and background material. David led the way on code development, with Paul contributing to the client local storage ($db\_utils.py$) and the client in general. Paul led the way on writing the paper, with David contributing with editing, portions of the evaluation and discussion, and notes for the design. Both Paul and David shared the workload on evaluating the system and debugging.

Our code base is entirely within a directory called *code/*. Everything was coded in Python 2.7.2 with some Shell scripts for testing. We include a README file in this directory which explains the major components of the code base.