Unity: Secure and Durable Personal Cloud Storage

Beom Heyn Kim
Computer Science
University of Toronto
Toronto, Canada
bhkim@toronto.edu

Wei Huang
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
wh.huang@mail.utoronto.ca

David Lie
Electrical and Computer Engineering
University of Toronto
Toronto, Canada
lie@eecg.toronto.edu

ABSTRACT

Unity provides secure and durable storage for personal data that does not depend on the security or availability of a central service. Instead, Unity exploits the trend towards users having more personal computing devices and the increasing amounts of storage available on those devices. This motivates the design of Unity, which does not store data on the cloud provider at all, but instead leverages the availability of the cloud provider to mount a coordination service that enables a user's devices to provide durable storage for the user's data themselves.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls; D.4.5 [**Reliability**]: Backup procedures

General Terms

Design, Security

Keywords

Cloud, Storage, Replication, Unity

1. INTRODUCTION

People today have more personal data in digital formats than ever before – every year, the average person accumulates gigabytes of digital photos, bills, receipts, e-mails and documents into their personal "digital repository". In addition, due to the increasing number of devices people own, they want the ability to access a single, consistent repository of data from any of their devices, such as their personal computer (PC), smartphone or tablet. As a result, both incumbent and startup companies, such as Dropbox, Google Drive, Microsoft SkyDrive, and SugarSync have moved to fill this need by offering both free and paid online storage in the cloud. By storing their digital repository in the cloud, a user can ensure that the repository is always available and that any device with Internet connectivity will have access to their digital repository anytime they need it. Another benefit to cloud storage is durability. Cloud providers store user data on highly reliable systems, which protects the data against both failures in the cloud provider and failures of the user's devices.

However, one critical property that all personal cloud storage solutions currently lack is security from a malicious or compromised cloud provider. Cloud providers or their employees may be malicious or negligent, or the cloud provider may be compromised by a third party. Finally, depending on the jurisdiction the cloud provider is located in, the cloud provider may be compelled by law enforcement to turn over user data against the user's wishes. Security is at odds with the current cloud provider model, where user data is stored by the cloud provider on the cloud provider's machines. In one high profile case, it was found that while Dropbox uses encryption to protect data in transit, Dropbox must keep data unencrypted on their servers so that they can deduplicate user data to save costs. This "feature" was discovered by Christopher Soghoian to result a side channel that can allow attackers to guess the contents of a user's data repository [12]. Due to the questionable security of cloud storage services, organizations such as IBM have outlawed the use of Dropbox by their employees.

A number of research projects have shown how to detect or mitigate malicious file servers [4, 6, 8]. In addition, some cloud storage providers, such as SpiderOak, have also started to address security concerns by offering encryption with a key known only to the user. However, if malicious behavior is detected, these systems offer no way for a user to recover their data or change to another service provider. While current cloud storage services may eagerly replicate data on clients, meaning that some or all of a user's repository may also be replicated on the user's devices, the primary purpose of this replication is to reduce file access latency, not to enable recovery. As a result, if the cloud provider is found to be malicious, the user has no guarantee that such eagerly replicated data is consistent or fresh, and thus cannot rely on it as a means to recover their data. Another alternative is to replcate the data on more than one cloud provider so that the user may still recover data in the event that a cloud provider is found to be malicious [3]. However, this increases the cost to the user as they must use and pay for more than one service.

To address the need for a secure personal repository that retains the desirable properties of availability and durability, we propose Unity, which aims to provide a cloud-based per-

CCSW'12, October 19, 2012, Raleigh, North Carolina, USA.

sonal repository service that detects malicious behavior by the cloud provider and enables users to recover their data in such an event. Unity is motivated by the observation that the trend towards more numerous computing devices and more storage on those devices means that users will have no problem storing redundant copies of their data across their devices, but instead will need the cloud provider primarily for its reliability and availability. Unity redivides responsibilities between the user and cloud provider so that the user is responsible for providing storage, while the cloud provider maintains a highly available and reliable coordination service for the devices.

Two salient features distinguish Unity from previous work. First, the Unity protocol ensures that in the event of a cloud provider failure, the Unity clients will always be able to recover back to a consistent snapshot of the user's data repository. We treat a malicious cloud provider the same way as we treat a failed cloud provider – in both cases, we assume that the user can no longer safely access the provider. Second, to achieve recovery, the clients coordinate with each other to ensure that every object in the repository is replicated in order to tolerate both cloud provider and client failures. However, Unity is not purely a peer-to-peer protocol. The Unity clients utilize the cloud provider's high-availability to improve performance and object lookup, but do so in a way such that they do not need to trust the integrity of the cloud provider.

We begin by reviewing related work from the literature and then describe our Unity design. We then follow up with discussion and a brief description of our Unity prototype.

2. RELATED WORK

The problem of implementing secure storage on top of insecure components has been explored by several systems in the literature. One category of such systems aim to implement secure storage with potentially malicious file server. SUNDR [8] protects the integrity of data stored on an untrusted server and uses versioning to ensure that the order of updates to blocks of data stored on the server is preserved. However, SUNDR only provides a guarantee of "fork consistency" to the clients, which means that a malicious server can lie to clients about the contents of files so long as it lies consistently to each client. In this attack, the server essentially "forks" the view of the file system and maintains different views for different clients. This is possible in SUNDR because the clients do not communicate with each other. Plutus [6] enhances the ideas behind SUNDR to also provide confidentiality and controlled sharing among different clients using cryptographic primitives. However, the cost of encryption as opposed to just signing means that Plutus has significantly higher overheads than SUNDR. Finally, while both Plutus and SUNDR require special servers, Sirius [4] implements an encrypting and signing file system on top of a generic, insecure file server. Unity improves on these systems by providing durability for user data if the server becomes unavailable, either due to a crash or if the server is discovered to be malicious so it can no longer be used.

A key difference between Unity and central server systems is that Unity clients work together to replicate user data so that if the server fails or acts maliciously, the user's data can be recovered by the clients themselves. In this way, Unity bears many similarities to peer-to-peer file systems. A number of secure peer-to-peer file systems, such as Farsite [1] and OceanStore [11] use byzantine fault tolerance to protect against malicious peers corrupting data stored on the file system. However, using byzantine algorithms and having an entirely distributed design means that such systems will only provide reasonable performance when all clients are connected by low-latency network such as a LAN, in the case of Farsite, or incur a 3x performance penalty in the case of OceanStore. Unity combines a centralized directory server with eager replication to provide reasonable performance even across a WAN.

Systems such as HAIL replicate data cross several cloud providers for resilience against attack [3]. Unity uses the same idea of replication to protect against failure and malicious attack, but is able to be simpler and more efficient by using the user's own devices as replicas, which allows Unity to place greater trust in systems replicating data.

Finally, orthogonal but related to Unity are systems that implement proofs of retrievability (POR) and provable data possession (PDP) for cloud storage [2, 5]. These systems allow a cloud provider to prove to users that it is faithfully storing the users' data without having to send the entire data to users. Unity provides essentially the same guarantee that the user's data is stored in a durable way.

3. THE UNITY SYSTEM

We begin by describing our assumptions about the environment that Unity will operate in. Then we describe a basic system that is functional but is neither durable nor secure. We then enhance the basic system to provide both durability and security.

3.1 Assumptions

Unity makes several assumptions about the environment that the Unity clients operate in. First, Unity is designed to store personal data across a set of clients running on devices that are owned and administered by the same user, so we assume that all the clients trust each other and execute the Unity protocol correctly. Without this assumption, malicious clients could lie about replicating information, thus undermining durability, or leak user data, undermining confidentiality. This assumption also allows Unity to use symmetric keys for cryptographic operations such as encryption and signing. Second, since the clients all have a single user Unity assumes that concurrent writes are rare, thus allowing the use of pessimistic locking to ensure consistency. Unity relies on this strict consistency during operation to guarantee the existence of consistent snapshots for it to recover back to. Finally, Unity assumes that clients will be available most of the time, giving them ample time to replicate user data for durability, but does not require any one client to be online all of the time.

3.2 A Basic System

Unity uses a fixed-size data block as the primary unit of storage and replication and builds file systems on top of these blocks. Data blocks in Unity are grouped into sets called *data entities* or DEs for brevity. Unity keeps track of versions for each block. Each time a block is modified, the version number for that block is incremented and Unity permits several versions of a block to simultaneously exist across clients. The mapping of DEs to higher-level abstractions such as files is arbitrary and Unity allows application-specific mapping of a single file, a group of files or even an

entire file system to a single DE. This mapping has implications on the consistency guarantees provided by Unity as described below.

Since files may be accessed by different devices, Unity uses a conservative form of consistency control to prevent conflicts. Clients who wish to modify a DE must request and acquire a lease on the DE, meaning that only one client may modify the DE at a time. This ensures that modifications to a DE are always linearizable. Unity guarantees that DEs are recoverable back to a consistent snapshot, but does not guarantee consistency across DEs. For example, if there are three DEs a, b, and c, which are modified in order a, b and c, Unity may recover back to a state where writes to a and c are recovered but the write to b is lost. However, Unity guarantees that such inconsistencies cannot happen for blocks within a DE after recovery.

Unity uses a *directory server* provided by the cloud service provider, which is assumed to be highly available under normal circumstances, but may become unavailable or tamper with data stored on the server if it becomes malicious. To ensure that data can be recovered if the server is malicious, clients running on each of the user's devices coordinate with each other via the server to replicate each block up to a predetermined replication level n. The primary purpose of the server is to provide a central, online directory of the locations of each block so that clients may quickly locate and request a block they need. When a lease-holder modifies a block, it asynchronously notifies the directory server of the creation of a new version for that block. When a client needs to read a particular version of a block from a DE, the client queries the server, which returns a list of clients who currently hold copies of that block-version. The requesting client then requests the block directly from one of those clients. Thus, the directory server itself does not store any of the block contents, just metadata about the location of each block. To reduce read latency, clients may eagerly replicate blocks according to a client-specific policy based on factors such as the likelihood that the client will be used to access a particular DE, the network bandwidth available to the client, battery life and the availability of storage on the client [10]. When a client replicates a block, it notifies the server so that the server can update its directory of block locations to indicate that the block is now available on the new client.

The directory server also participates in assigning and revoking leases to the clients and records the identity of the current lease-holder for each DE, thus making it easy for a client to determine which peer to request the lease from. When a client wishes to modify one or more blocks within a DE, it requests the lease from the directory server, which then contacts the current lease-holder with a message indicating that the lease should be revoked and passed to the requester. The current lease-holder then sends a message to the requesting client indicating that it is releasing the lease. Within this message, the current lease-holder also includes a list of current block-versions for the DE. This direct communication between the current lease-holder and the new-lease holder not only reduce the latency for the lease-holder switch, but also is critical for security against a malicious server as we will explain below. At the same time, the current lease-holder also updates the server with any outstanding block-versions it has. With the message from the current lease-holder, the new lease-holder now has a fresh view of the latest blocks in the DE and may fetch and start modifying blocks within the DE.

4. PROVIDING DURABILITY

Currently, clients only request a block from another peer if they need to read or modify the block. While some replication will naturally happen because of this, the basic system above provides no guarantees for the durability of a particular block. If no other client reads a block, it will never be replicated and if the only client that has the block fails, the block is gone forever. While clients may eagerly replicate some blocks to reduce read latency, the above protocol only requires them to do this based on a locally determined policy, which cannot guarantee that all blocks will achieve the n replication factor that we desire from Unity. Since Unity does not depend on the cloud provider for durability, we must place additional constraints on the local replication policy of each client to ensure that blocks are appropriately replicated to achieve the desired durability.

To ensure that Unity can recover to a consistent snapshot for each DE in the face of up to $n-1$ client failures, we require that when eagerly replicating blocks, clients must replicate blocks that do not already have n replicas across the clients first. Clients can find out the replication level of each block-version by querying the directory server, which keeps track of which clients hold of each block-version. When there are several blocks that need replication, the clients make randomized local decisions. The use of randomized, local decision making has been shown to eventually converge to a near-optimal level of replication across all clients [9].

Requiring that clients replicate blocks but never delete them would mean that clients would end up exhausting all their storage on old block-versions that are no longer needed for recovery. Thus, the server runs a garbage collection algorithm that determines when it is safe to delete old block-versions while still guaranteeing that the clients can collectively recover back to a consistent snapshot of each DE. To do this, the server organizes notifications from the clients into *block-version entries*, where each entry stores the clients holding the block-version associated with the entry. An initial entry with version zero exists for each block in a DE and new entries are created each time a version of a block is created due to a modification. For each DE, the server keeps a log of all block-versions for the DE, sorted by the creation time of the block-version. The ordering of this log is well-defined because the file cannot have any concurrent writes due to the lease-holder policy described above. The server marks a contiguous region of the log for each DE as the *recovery window* for the DE. The recovery window for a DE is defined by three constraints: i) Each block in the DE must have exactly one version in the recovery window, ii) If a block has more than one version, then all versions except the latest version in the window are garbage-collected and iii) The window must be a contiguous region of the log and all non-garbage collected entries in the window must have at least n replicas. Clients are then free to delete garbage-collected block-versions from their local storage as they are no longer needed for durability. The recovery window mechanism ensures that when up to $n-1$ client failures occur, the remaining clients will collectively be able to recovery back to a consistent snapshot of the DE right after the latest block-version in the recovery window.

To illustrate the operation of the recovery window, con-

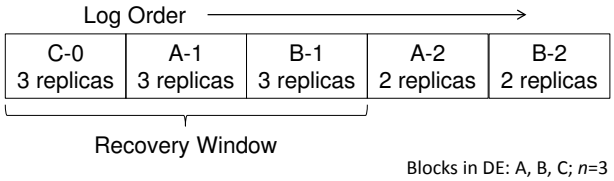

Figure 1: Recovery Window example with three blocks A, B, and C in the DE and $n = 3$. Block-version B-2 cannot be added to the recovery window even if it is replicated 3 times until block-version A-2 is replicated 3 times.

sider the example in Figure 1. Here, block-versions A-1, B-1 and C-0 have reached the required replication level of 3, and newer block-versions A-2 and B-2 are short one replica. If block-version A-2 is replicated one more time, rule (iii) above allows it to be added to the recovery window. Rule (ii) then dictates that block-version A-1 can be removed from the recovery window since a later version of the same block is in the window. If the $n-1$ clients were to fail at this point, the DE would be guaranteed to be recoverable back to a consistent snapshot right after A-2 was created (more block-versions might be recoverable depending on which clients failed). Now, suppose that instead of A-2 getting replicated first, B-2 reaches n replicas first. In this case, the recovery window does not change because including B-2 in the recovery window would also include A-2 (the window must be contiguous), which does not have n replicas, and this would violate rule (iii). To see why this is disallowed, consider if both B-2 and A-2 were added to the recovery window even though A-2 only has 2 replicas. According to rule (ii), both A-1 and B-1 can now be removed leaving C-0, A-2 and B-2, and the system should be recoverable up to a snapshot after the creation of B-2. However, if the two clients replicated the block-version A-2 now fail, the clients will be unable to recover back to a consistent snapshot because the replicas of both A-1 and B-1 have been garbage collected and all replicas of A-2 have been lost, violating the guarantee that the system can recover with up to $n-1$ failures. Leaving B-2 outside of the recovery window prevents A-1 and B-1 from being garbage collected until it is safe to do so.

To detect failed clients, the directory server maintains a heart-beat with each client. If a client fails to respond to a heart-beat within a timeout, the server marks the client's state as failed, but does not delete the list of blocks that the client is holding in case the client returns. Clients may query the server to get a status of each of the other clients. This allows clients to perform recovery when appropriate, as well as replicate blocks that the failed client was replicating.

5. PROVIDING SECURITY

At this point, we see that the server plays several important roles in Unity that make it a potential threat to the security of the user's data repository. We note that confidentiality for the user's data is trivially satisfied since the cloud provider's server only sees directory updates about the creation of block-versions and their locations, but never sees the data itself. While side-channel leakage due to the server being able to observe file access patterns is possible, we leave that out of scope in our current design. The four main responsibilities of the server, which are i) recording the existence and location of block-versions, ii) coordinating lease-holder switches, iii) garbage collection of old block-versions and iv) detection of failed clients, give the server four avenues with which it may affect the integrity and durability of the user's data repository. We now enhance the basic protocol to provide security and durability in the face of a malicious directory server.

5.1 Attacks on Block-Version Information

In maintaining the existence and the location of block-versions, there are essentially two types of attacks a malicious server could mount. First, a malicious server could attempt to forge or tamper with a block-version entry. We extend Unity to prevent this by having each client sign the notifications they send to the server about new block-versions or replication of a block-version. Thus, the server cannot forge a new block-version or create an entry that says a client has replicated a block-version when it hasn't.

Second, a malicious server could omit or replay new block-versions or omit or replay notifications that a block-version has been replicated. To prevent the former, the lease-holder includes a sequence number with the notification of each newly created block-version by the lease-holder, thus preventing a server from selectively omitting or replaying block-versions. Again, the sequence of block updates in a DE is well-defined because Unity ensures that updates to a DE are always linearizable. The server could stop accepting new block-versions completely, thus truncating the log, but this attack would be discovered as soon as there is a lease-holder switch as the old lease-holder forwards the list of block-versions it has created directly to the new lease-holder. This direct state-passing during a lease-holder switch is what prevents the "fork-consistency attack" described in SUNDR [8] as the lease-holders will all eventually gain a consistent view of the DE even in the presence of a malicious directory server. The use of sequence numbers places an additional constraint on a legitimate directory server, which must ensure that every client has seen a particular block-version before deleting it during garbage collection. Otherwise, a client cannot tell a legitimate directory server that is performing garbage collection from a malicious one that is illegally deleting block-versions. This also means that rather than querying the server about a specific block, clients will instead query the server for all block-versions since a particular sequence number.

Rather than omit a particular block-version, a malicious server could try to omit updates to a block-version when a client replicates the block. Unfortunately, such omissions cannot be detected with sequence numbers because replication is asynchronous across the clients. This attack has two consequences. First, the replication level for the block will exceed n as clients will think the block is under-replicated when it is not. Second, the clients that the server did not omit will receive more requests for block reads then they should under normal circumstances. In the worst case, the server omits all replication updates except for the very first update from the lease-holder, which it cannot omit because of the sequence number. In this case, the lease holder would be burdened with servicing all requests for the block. While both of these have implications on performance and resource utilization, we view these as acceptable as they do not impact the security or durability of user data. Replaying a replication message has no effect since clients only care if another client has the block or not.

5.2 Attacks on Lease-Holder Switching

In coordinating a lease-holder switch, a malicious server may misbehave in two ways. First, it may not send the revoke lease message to the current lease-holder. This has no harm since the requesting lease holder will not assume it has the lease until it receives a signed notification from the current lease-holder. It will eventually time out and try to acquire the lease again. Second, the server could send a revoke lease message to the lease-holder when no client actually requested the lease. In this case, the current lease-holder will send a message to the new lease-holder, but since the new lease-holder never requested the lease, the malicious server will be discovered.

5.3 Attacks on Garbage Collection

A malicious server may not execute the garbage collection algorithm properly. However, because the server cannot forge or omit block-versions, each client can correctly re-execute the garbage collection algorithm locally and verify that the server is implementing it properly. As mentioned above, a malicious server may cause some blocks not to get garbage collected by omitting replication notifications, but this has no impact on security or durability. However, the malicious server cannot cause blocks to be garbage collect blocks before their time, which if it could, would undermine the durability of the data.

5.4 Attacks on Failure Detection and Handling

The directory server is responsible for detecting failed clients and notifying the other clients. A malicious server could mark a client as failed when it has not actually failed. However, since the clients may communicate with each other directly, when the server does this, each client will verify the claim by attempting the supposedly failed client directly. We note that the server incorrectly marking a client as failed is not necessarily an indicator that the server is malicious as this could also occur due to a network partition or intermittent unavailability of the clients.

If a client actually does fail, this has little consequence unless the client is the lease-holder. A failure of a client that does not hold the lease only results in the other clients having to replicate the blocks the failed client was replicating to ensure that the system can still tolerate $n-1$ failed clients. However, failure of a lease-holder means that the remaining clients must elect a new lease-holder. They cannot rely on the server for this operation since a malicious server may take advantage of the circumstances and cause two or more lease-holders to be created, resulting in a forked state. Instead, the clients may either use an entirely distributed solution, such as the Paxos election protocol [7] and accept the constraints on timing and liveness that accompany such protocols, or the clients may rely on an out-of-band solution, such as having the user designate one of the clients to be the new lease-holder. All clients must be made aware of the identity of the new lease-holder so that if the failed lease-holder returns it will not be treated as a valid lease-holder. To implement this, whenever a new lease-holder is created, either due to a regular lease-holder switch from one client to another, or due to a failed lease-holder, the new lease-holder will insert a signed message into the log of the DE indicating that it is now the lease-holder. In this way, all other clients will be notified of the most recent lease-holder, including a lease-holder that returns back to the group after a failure. Similarly, when initializing a group of clients and there is no initial lease-holder, the clients must select a single lease-holder without depending on the server as well.

5.5 Recovery from a Malicious Server

If the cloud server fails or is discovered to be malicious the clients must be able to recover the data in the user's repository and initialize a new server to replace the failed one. Because the clients are already replicating data blocks to guard against client failure, they can reconstruct a consistent snapshot of the user's data repository from the replicated blocks. To initialize a new server, the clients must supply location information about which blocks are replicated at which clients, which they naturally have. Thus, to move the service to a new server, the clients inform the new server about which block-versions each of them is caching, as well as the current sequence number of each DE.

6. DISCUSSION

One major distinction between Unity and current cloud storage providers is that in Unity, the cloud service provider is not actually providing any storage or durability at all. Instead, the user themselves supplies all the storage and is responsible for supplying enough devices to ensure durability. The main value that the cloud provider brings in the Unity system is a high-availability directory server that coordinates the clients running on the devices of the user. A valid question might be whether this model where the cloud provider supplies fewer resources in exchange for security is a good value proposition for the user.

To answer this question, we observe that two trends in the user device market. First, user devices have been increasing in capacity without any significant increase in cost to the user. At the time of writing, the average mobile smartphone or tablet has anywhere from 16GB to 64GB of flash storage. This in itself is much larger than the "free" storage that most cloud storage providers are providing, which is in the range of 2-7GB. Thus, at no extra cost, users with today's devices are able to supply the same amount of free storage that is available from cloud storage providers today. With, Unity users may even exceed these free amounts without requiring any extra services from the cloud provider. The other trend is that users are acquiring more and more devices. A typical user may have a desktop PC and a laptop, as well as a smartphone and a tablet. In the future, we expect this trend to continue as users acquire network-capable set-top boxes for content delivery and games, home NAS and media servers, and computing and network capability is added to appliances and automobiles. Thus, users will have not only enough storage to store their repository, but also enough devices to achieve the replication needed for durability.

If a user does not have enough clients or enough storage across clients to replicate and store their data repository, they may use the services of an untrusted client provided by another cloud provider. The untrusted client behaves exactly like a regular client except that it only replicates blocks for durability and never requests leases for DEs because it never writes to the repository. When replicating data, the clients owned by the user sign and encrypt the data block contents before sending it to the untrusted client and add a signed hash to the metadata stored on the directory server for freshness. So long as the untrusted clients and directory server do not collude, untrusted clients can

effectively provide the necessary storage and redundancy to ensure durability for the user's data. It is important that a user does not use the same cloud provider for both the directory server and the untrusted client because together, they can undermine the durability of the user's data.

7. PROTOTYPE IMPLEMENTATION

We are currently working on a Unity prototype to evaluate the performance overhead of the durability and security guarantees that Unity provides. Both the directory server and Unity clients are implemented in about 9.9K lines of C code. Currently, we have two applications for Unity. In one application, we use Unity to implement a block device which can support any commodity local file system. In the other application, we implement a Unity-specific file system where each file is mapped to a DE.

Our preliminary results are promising. We conduct an experiment where we boot a 1.1GB virtual machine (VM) image on a Xen hypervisor where the virtual disk of the VM is backed by a block device using Unity. We have two machines in our experimental setup and initially all the blocks of the virtual disk are located on the first machine. We then use a simple workload where we boot and shutdown the VM on the second machine, causing Unity query the directory server and fetch all the necessary blocks from the first machine. Right now, the second machine does not do any eager replication so all block fetches are performed when the VM boot requests them. Using the same workload, we compare the performance of Unity to NFS where the image file is on an NFS share, to Dropbox where the image file is replicated from the first machine to the second machine through LAN sync and to the local disk where the image file is local to the machine running the hypervisor. For Dropbox, the image is replicated peer-to-peer through the LAN by Dropbox. We did 5 runs for each case. Unity takes an average of 32.68 $\pm$ 0.284 s while the local disk, NFS and Dropbox take 20.52 $\pm$ 0.224 s, 34.75 $\pm$ 0.726 s, and 36.58 $\pm$ 3.802 s respectively. The results suggest that Unity is competitive with both current cloud storage solutions and file servers.

8. CONCLUSION

Unity proposes a different model for maintaining a cloud-based personal data repository that is available, durable and secure. In the Unity paradigm, users have many devices that do not lack storage capacity, but need a service that is highly available and reliable to compensate for devices that are prone to failure, intermittent connectivity, physical damage or getting lost. By using the cloud provider to implement a directory service that coordinates the replication of spreading redundant copies across the user's devices, the preliminary results of our Unity prototype suggest that this can be achieved with performance that is comparable to the best cloud storage services today. The current Unity model assumes that failures are rare, but this may not be the case with a large number of cheap devices. In our future work, we plan to enhance Unity to tolerate failures more efficiently.

Acknowledgements

Funding is provided by the NSERC ISSNet Strategic Network, an NSERC Discovery Grant and an MRI Early Researcher Award.

9. REFERENCES

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *The 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, Dec. 2002.

[2] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *The 14th ACM Conference on Computer and Communications Security (CCS)*, pages 598–609, Nov. 2007.

[3] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *The 16th ACM Conference on Computer and Communications Security (CCS)*, pages 187–198, Nov. 2009.

[4] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *The 10th Symposium on Network and Distributed System Security (NDSS)*, Feb. 2003.

[5] A. Juels and B. S. K. Jr. PORs: Proofs of retrievability for large files. In *The 14th ACM Conference on Computer and Communications Security (CCS)*, pages 584–597, Oct. 2007.

[6] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *The 2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 29–42, Apr. 2003.

[7] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[8] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *The 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, Dec. 2004.

[9] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *The 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–41, Oct. 2011.

[10] D. Peek and J. Flinn. EnsemBlue: Integrating Distributed Storage and Consumer Electronics. In *The 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 219–232, November 2006.

[11] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *The 2nd USENIX Conference on File and Storage Technologies (FAST)*, pages –1–1, 2003.

[12] C. Soghoian. How Dropbox sacrifices user privacy for cost savings, 2011. http://paranoia.dubfire.net/2011/04/how-dropbox-sacrifices-user-privacy-for.html (last accessed: 07/13/2012).