CYRUS: Towards Client-Defined Cloud Storage

Jae Yoon Chung
Dept. of CSE
POSTECH
dejavu94@postech.ac.kr

Carlee Joe-Wong
PACM
Princeton University
cjoe@princeton.edu

Sangtae Ha
Dept. of CS
University of Colorado
sangtae.ha@colorado.edu

James Won-Ki Hong
Dept. of CSE
POSTECH
jwkhong@postech.ac.kr

Mung Chiang
Dept. of EE
Princeton University
chiangm@princeton.edu

Abstract

Public cloud storage has recently surged in popularity. However, cloud storage providers (CSPs) today offer fairly rigid services, which cannot be customized to meet individual users' needs. We propose a distributed, client-defined architecture that integrates multiple autonomous CSPs into one unified cloud and allows individual clients to specify their desired performance levels and share files. We design, implement, and deploy CYRUS (Client-defined privacY-protected Reliable cloUd Service), a practical system that realizes this architecture. CYRUS ensures user privacy and reliability by scattering files into smaller pieces across multiple CSPs, so that no one CSP can read users' data. We develop an algorithm that sets reliability and privacy parameters according to user needs and selects CSPs from which to download user data so as to minimize latency. To accommodate multiple autonomous clients, we allow clients to upload simultaneous file updates and detect conflicts after the fact from the client. We finally evaluate the performance of a CYRUS prototype that connects to four popular commercial CSPs in both lab testbeds and user trials, and discuss CYRUS's implications for the cloud storage market.

1. Introduction

Cloud computing, driven in large part by business storage, is forecast to be a $240 billion industry in 2020 [15], with 3.8 billion personal storage accounts in 2018 [31]. Yet even as storing files on the cloud becomes ever more popular, privacy and reliability concerns remain. Cloud storage providers (CSPs) have experienced high-profile data leaks, e.g., credit card information from Home Depot [32] and celebrity photos from iCloud [39], as well as outages: Amazon's popular S3 service went down for almost an hour in 2013, affecting websites like Instagram and Flipboard [37]; and Dropbox went down for three hours in 2014 [18].

These performance deficiencies are especially serious for enterprise file sharing, a fast growing market [21] that requires high reliability and rapid file synchronization. File sharing presents unique challenges to cloud storage, as different clients (e.g., different users or multiple devices belonging to one user) must be able to securely retrieve the latest versions of the file from the cloud. Further complicating this need is the fact that some clients may not always be active, preventing reliable client-to-client communication. We can identify several requirements for effective file sharing:

- **Privacy:** Users wish to prevent other parties, including cloud providers, from accessing their data.
- **Reliability:** Users must be able to access data stored in the cloud at all times.
- **Latency:** To effectively share files, users must be able to quickly retrieve the latest file versions from the cloud and upload edited versions in real time.

The performance criteria above may sometimes conflict, e.g., greater privacy may require more client-cloud communication and worsen the latency of file updates. We therefore seek a system design that allows clients to freely navigate these tradeoffs according to their own needs and priorities. To this end, we propose a cloud storage system called CYRUS (Client-defined privacY-protected Reliable cloUd Storage) that satisfies the above requirements by leveraging user accounts at multiple clouds (e.g., private storage servers

EuroSys'15, April 21–24, 2015, Bordeaux, France.

ACM 978-1-4503-3238-5/15/04. . . $15.00.
http://dx.doi.org/10.1145/2741948.2741951

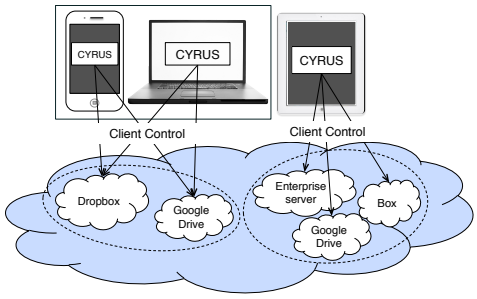


Figure 1: CYRUS allows multiple clients to control files stored across multiple CSPs.

or user accounts at CSPs like Dropbox and Box). CYRUS controls the distribution of data across different CSPs from user devices, allowing users to customize their achieved performance requirements and trade off between them as necessary. Figure 1 illustrates CYRUS's basic architecture, with multiple users sharing files stored at multiple CSPs.

1.1 Distributed Cloud Storage Architectures

Leveraging multiple CSPs represents an architectural change from the usual Infrastructure-as-a-Service (IaaS) model, which rents customized virtual machines (VMs) to users. While some CSPs offer software-defined service level agreements with customized performance guarantees [22], these approaches fundamentally leave control at the CSP. Clients control their VMs or specify performance through software on the cloud and are still limited by the CSP's functionality. Yet CYRUS does not consist solely of autonomous clients either, unlike peer-to-peer systems that allow each client node to leverage resources at other connected nodes for storage or computing. CYRUS does not treat CSPs as peers and does not assume that clients can directly communicate with each other; instead, the CSP "nodes" become autonomous storage resources separately controlled by the clients.

CYRUS's distributed, client-controlled architecture enforces privacy and reliability by scattering user data to multiple CSPs, so that attackers must obtain data from multiple CSPs to read users' files; by building redundancy into the file pieces, the file remains recoverable if some CSPs fail. File pieces can be uploaded and downloaded in parallel, reducing latency. Since the client controls the file distribution, clients can choose where to distribute the file pieces so as to define customized privacy, reliability, and latency.

CYRUS is not the first work to recognize the benefits of a client-defined approach to cloud storage. DepSky, a "cloud-of-clouds" system, also controls multiple CSPs from the client [7]. Yet while DepSky demonstrates a proof-of-concept system with Byzantine quorum system protocols, it does not fully address the practical challenges of realizing such client-controlled storage for large-scale deployment. In particular, users wishing to share files can have stringent latency requirements, which they may wish to trade off with privacy and reliability. For instance, higher privacy requirements would require downloading file pieces from slower clouds. CYRUS allows full privacy and reliability customization and optimizes latency in realizing a deployable and customizable storage system.

CYRUS's design addresses four challenges not fully covered in previous work:

- **Heterogeneous CSP file handling:** CSPs have different file management policies and APIs, e.g., tracking files with different types of IDs and varied support for file locking. To accommodate these differences, CYRUS uses only the most basic cloud APIs. This limitation makes it particularly difficult to support multiple clients trying to simultaneously update the same file; unlike conflict resolution protocols such as Git [2], CYRUS must be able to resolve conflicts without server-side support. Our approach is inspired by large-scale, performance-focused database systems [5, 11, 13]. We allow clients to upload conflicting files and then resolve conflicts as necessary. Other approaches require more client-CSP communication, increasing latency [7].
- **Sharing files between clients:** CYRUS's client-based architecture and the lack of reliable client-to-client communication make sharing files between clients difficult: clients need to share information on each file's storage customization. We avoid storing this data at a centralized location, which creates a single point of failure, by scattering file metadata among CSPs.
- **CSP infrastructure sharing:** Some CSPs may share infrastructure, e.g., Dropbox running its software on Amazon's servers [16]. Storing files at CSPs with the same physical infrastructure makes simultaneous CSP failures more likely. To maximize reliability, CYRUS infers these sharing relationships and uses them in selecting the CSPs where files are stored.
- **Optimizing file transfer delays:** Client connections to different CSPs have different speeds. Thus, CYRUS significantly reduces latency by optimally choosing the CSPs from which to download file pieces, subject to privacy and reliability constraints on the number of file pieces needed to reconstruct the file.

We discuss other works related to distributed, client-controlled cloud storage in Section 2.

1.2 Research Contributions

In designing, implementing, and deploying CYRUS, we make the following research contributions:

Client-based architecture integrating multiple CSPs (Section 3): CYRUS's software, located at client devices, scatters files across different clouds so that no one CSP can access them, but the file can be recovered if some CSPs fail. CYRUS is CSP-agnostic, requiring only basic APIs at each CSP, and supports multiple clients (i.e., devices) concurrently accessing the same files with no increase in latency.

These devices may be owned by different users, allowing them to share data.[1]

Customizable reliability, privacy, and performance (Section 4): Users can configure reliability and privacy levels by specifying the number of CSPs to which different files should be scattered and the number of pieces required to reconstruct the file. We infer CSPs sharing cloud platforms and select those that do not share infrastructure to maximize reliability. Users select the CSPs from which to download the file pieces in parallel so as to minimize latency.

CYRUS operations on multiple clouds and clients (Section 5): Users build a CYRUS cloud by calling a standard set of APIs. CYRUS transparently aggregates file operations on multiple CSPs, internally storing and tracking file metadata and CSP storage locations for efficient operations.

Prototype implementation and deployment (Sections 6 and 7): We implemented a prototype of CYRUS on Mac OS X and Windows.[2] We evaluate its performance on four commercial CSPs and show that CYRUS outperforms other CSP integration services in real-world scenarios. We also present results from a small pilot trial in the U.S. and Korea.

Discussion of market implications (Section 8): While many users today only use storage from one CSP to avoid managing multiple CSP accounts, CYRUS encourages users to use more CSPs for greater privacy and reliability. CYRUS thus evens out demand across CSPs and makes it easier for market entrants to gain users.

2. Related Work

Byzantine fault-tolerance systems have been studied for reliable distributed storage [10, 17, 25–27]. These works proposed protocols that can read and write with unreliable CSPs. Realizing these protocols, however, requires running servers at CSPs, which is not generally applicable to cloud services based on HTTP(S) REST APIs. Similarly, object-based distributed storage systems [9, 40] also require running code at the servers to handle multiple transactions and synchronization. In these systems, user data are split into smaller objects and distributed on multiple object storages. To reconstruct files, the systems maintain metadata, including the required objects and their storage locations.

Another approach to integrating multiple CSPs is to use proxy servers [3, 19, 30, 35, 38]. The proxy can scatter and gather user data to and from multiple CSPs, providing transparent access for users. Yet while the proxy server is a data sharing point among multiple clients, allowing greater deduplication efficiency, it is also a single point of failure.

Cloud integration from the client has been proposed in [6, 7, 23, 24, 29]. CYRUS, however, provides a customizable framework that sets reliability and privacy levels while optimizing user delays in accessing and storing files. Moreover, CYRUS allows multiple clients to upload conflicting files and resolve conflicts later, which improves the transaction (e.g., file upload) speed compared to alternative approaches. Table 1 compares CYRUS with similar storage systems. We design CYRUS to achieve all of Table 1's functions without a central server for coordinating client's requests.

3. CYRUS Design

We first highlight some important challenges in building CYRUS in Section 3.1 before presenting CYRUS's system architecture and API for user interaction in Section 3.2.

3.1 Design Considerations

CYRUS's operations and functionalities must reside either on the clients, CSPs, or a combination of both. Yet in realizing such functionalities, we face two main challenges: *heterogeneity in CSP file handling* and the *lack of reliable direct client-to-client communication*.

To illustrate CSP heterogeneity, Table 2 shows the APIs and achieved performance for a range of CSPs. Though most are similar, the API implementations differ in the functions they provide and their file object handling. For example, Dropbox uses files' names as their identifiers, while Google Drive uses a separate file ID. Thus, when a client uploads a file with existing filename, Dropbox overwrites the previous file, but Google Drive does not. CYRUS accommodates such differences by only using basic cloud API calls: authenticate, list, upload, download, and delete, which are available even on FTP servers. We therefore shift much of CYRUS's functionality to the clients, influencing our design choices:

Ensuring privacy and reliability: Standard replication methods for ensuring reliability do not require significant client or CSP resources but are not secure. Many encryption methods, on the other hand, are vulnerable to loss of the encryption key. CYRUS overcomes these limitations by splitting and encoding files at the client and uploading the file pieces to different CSPs. For further security, the pieces' filenames are hashes of information known only to the clients. Reconstructing the file requires access to pieces on multiple, but not all, CSPs, ensuring both privacy and reliability.

Concurrent file access: Since most CSPs do not support file locking, CYRUS cannot easily prevent simultaneous file uploads from different clients: the second client will not know that a file is being updated until the first client finishes uploading. Theoretically, since POST in HTTP is not idempotent, the server status should change when the first update finishes, and we could handle the conflict by overwriting the first with the second file update. However, CSPs do not always enforce this standard. Thus, a locking or overwriting approach requires creating lock files and checking them after a random backoff time, leading to long delays [7]. CYRUS instead creates new files for each update and then detects and resolves any conflicts from the client.

[1] In the rest of the paper, "users" and "clients" are used synonymously and can refer to multiple devices owned by one person.

[2] Demo video available at `http://youtu.be/DPK3NbEvdM8`.

	Erasure coding	Data deduplication	Concurrency	Versioning	Optimal CSP selection	Customizable reliability	Client-based architecture
Attasena [6]	Yes	No	Yes	No	No	No	No
DepSky [7]	Yes	No	Yes	Yes	No	No	Yes
InterCloud RAIDer [23]	Yes	Yes	No	Yes	No	No	Yes
PiCsMu [24]	No	No	No	No	No	No	No
CYRUS	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 1: Comparison of CYRUS's features with similar cloud integration systems.

CSP	Format	Protocol	Authentication	**RTT** (ms)	**Throughput** (Mbps)
Amazon S3*	XML	SOAP/REST	AWS Signature	235	1.349
Box	JSON	REST	OAuth 2.0	149	2.128
Dropbox	JSON	REST	OAuth 2.0	137	2.314
OneDrive	JSON	REST	OAuth 2.0	142	2.233
Google Drive	JSON	REST	OAuth 2.0	71	4.465
SugarSync	XML	REST	OAuth-like	146	2.171
CloudMine	JSON	REST	ID/Password	215	1.474
Rackspace	XML/JSON	REST	API Key	186	1.704
Copy	JSON	REST	OAuth	192	1.651
ShareFile	JSON	REST	OAuth 2.0	215	1.474
4Shared	XML	SOAP	OAuth 1.0	186	1.704
DigitalBucket*	XML	REST	ID/Password	217	1.461
Bitcasa*	JSON	REST	OAuth 2.0	139	2.281
Egnyte	JSON	REST	OAuth 2.0	153	2.072
MediaFire	XML/JSON	REST	OAuth-like	192	1.651
HP Cloud	XML/JSON	REST	OpenStack Keystone V3	210	1.509
CloudApp*	JSON	REST	HTTP Digest	205	1.546
Safe Creative*	XML/JSON	REST	Two-step authentication	295	1.075
FilesAnywhere	XML	SOAP	Custom	202	1.569
CenturyLink	XML/JSON	SOAP/REST	SAML 2.0	293	1.082

Table 2: APIs and measured performance of commercial cloud storage providers. Throughput is calculated from the measured RTT assuming a 0.1% packet loss rate and 65,535 byte TCP window size. All measurements were taken in Korea.

Client-based architecture: To reconstruct a file, a client needs access to its metadata, i.e., information about where and how different pieces of the file are stored. The easiest way to share this metadata is to maintain a central metadata server [8], but this solution makes CYRUS dependent on a single server, introducing a single point of failure and making user data vulnerable to attacks at a single server. At the other extreme, a peer-to-peer based solution does not guarantee data accessibility since clients are not always available. Our solution is to scatter the metadata across all of the CSPs, as we do with the files. Clients access the metadata by downloading its pieces from the CSPs; without retrieving information from multiple CSPs, attackers cannot access user data.[3] This approach ensures that CYRUS is *as consistent as the CSPs where it stores files.*

Selecting CSPs: CYRUS chooses the number of file pieces to upload or download so as to satisfy privacy and reliability requirements. When choosing *which* CSPs to use, CYRUS chooses CSPs on independent cloud platforms so as to maximize reliability. For instance, CSPs marked with an asterisk in Table 2 have Amazon destination IPs, so a single failure at Amazon's datacenters could simultaneously impact these CSPs, compromising reliability. The table also shows that different CSPs have very different client connection speeds; thus, carefully selecting the CSPs from which CYRUS downloads files can significantly reduce the latency of file transfers.

[3] Since we store metadata pieces at *all* CSPs, clients can always find and download metadata pieces.

Functionality	CYRUS API call
create a CYRUS cloud s	`s = create()`
add a cloud storage c	`add(s,c)`
remove a cloud storage c	`remove(s,c)`
get a file f of version v	`f' = get(s,f,v)`
put a file f	`put(s,f)`
delete a file f	`delete(s,f)`
list files under a directory d	`[(f,r),] = list(s,d)`
reconstruct s'	`s' = recover(s)`

Table 3: CYRUS's Application Programming Interface.

3.2 CYRUS Architecture

Users interact with CYRUS through Table 3's set of API calls. These include standard file operations, such as uploading, downloading, and deleting a file, as well as file recovery and CSP addition and removal. To realize this API while meeting the design challenges in Section 3.1, CYRUS must

perform three main functions: integrate multiple clouds, scale to multiple clients, and optimize performance.

Integrating multiple clouds: As explained above, CYRUS scatters file pieces to multiple CSPs so that no single CSP can reconstruct a user's data. We increase reliability by storing more file pieces than are necessary for recovering the file. This idea is encapsulated in a (t, n) *secret sharing scheme*, which divides user data into n shares, each stored on a different CSP [36]. Secret sharing divides and encodes the data in such a way that reconstructing any part of the original data requires at least t of the file shares. Taking $t < n$ thus ensures reliability, and taking $t > 1$ ensures that multiple CSPs are required to recover user data. Users can reconstruct the file using the shares from any t CSPs.

Scaling to multiple clients: To download files, clients must know the locations of the files' shares. Thus, we maintain a separate metadata file for each file stored on the cloud; as a client uploads a file, it records the share locations in this metadata. We store the metadata in a logical tree at CSPs, with clients maintaining local copies of the metadata tree for efficiency. All clients can sync their local copies of the metadata tree to track updated share and file locations. The tree structure also allows CYRUS to handle conflicting file updates. Clients do not lock files while modifying them but can upload conflicting file versions as different nodes on the metadata tree. We traverse the tree to find and resolve file conflicts.

Optimizing cost and performance: CYRUS reduces users' cost by limiting the amount of data that must be stored on CSPs. Before scattering files, we divide each file into smaller discrete chunks. Unique chunks are then divided into shares using secret sharing, which are scattered to the CSPs. Figure 2 illustrates this division of files to chunks and chunks into shares. Since different files can use the same chunks, deduplication reduces the total amount of data stored at CSPs, conserving storage capacity. We also limit the amount of data that can be stored at different CSPs, e.g., to the maximum amount of free storage capacity.

We choose the number of shares to upload in order to satisfy reliability and privacy constraints, i.e., n and t for secret sharing. Adjusting these parameters allows CYRUS to adapt to changes in cloud conditions and user preferences. After choosing n, we select the CSPs so that they do not share a cloud platform. When reconstructing files, we minimize the latency of downloading shares.[4]

4. Optimized Cloud Selection

We first consider inferring which CSPs run on the same cloud platforms in Section 4.1, which helps to ensure reliability by avoiding correlated CSP failures. Given this lack of correlation, we then consider how many shares to upload to CSPs (Section 4.2) and present an algorithm for choosing the CSPs from which to download shares in Section 4.3.

[4] This downlink CSP selection is client-specific and does not affect other clients' performance: regardless of the CSPs selected for downloading the shares, any modified shares are uploaded to the same set of CSPs for other clients to retrieve (uplink CSP selection is discussed in Section 5.3). Thus, when other clients retrieve the modified shares, they have the same set of choices for CSPs from which to download the modified shares.

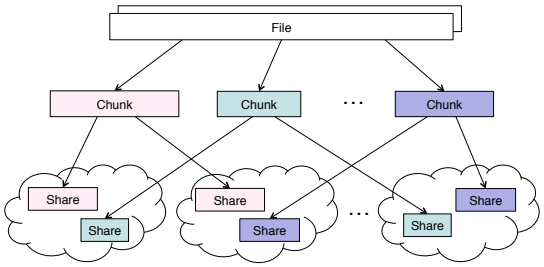


Figure 2: Mapping files to shares.

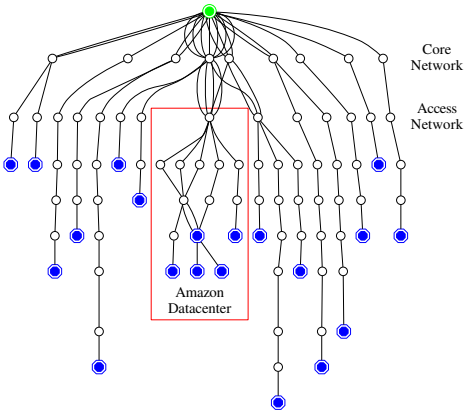


Figure 3: Clustering of Table 2's CSPs. The root and leaf nodes represent the client and CSPs respectively.

4.1 CSP Platform Independence

Storing shares of the same chunk at CSPs on a common cloud platform increases the chance that client data will not be recoverable, which is particularly serious during long-term CSP outages. To prevent this scenario, clients that highly value reliability can cluster their CSP accounts by cloud platform and store a chunk's shares on at most one CSP from each cluster. Since infrastructure locations rarely change, the clustering need only be done once and updated when the client adds a new CSP account.

Since some users will have many CSP accounts, manually clustering CSPs by cloud platform is often infeasible. We can therefore infer the CSP clusters by either network probing to detect the client-CSP routing paths or by geolocation based on destination IPs. CYRUS takes the first approach, as geolocation can only detect approximate geographical proximity, while routing traces can more precisely identify shared infrastructure. We use `traceroute` to find the path between a given user and each CSP and construct the minimal spanning tree of the resulting graph.[5] Figure 3 shows an example of the tree from a single client to the CSPs

[5] With some CSPs, clients connect to endpoint APIs that are separate from the file storage locations. We can read the subsequent internal CSP connection to the file storage location to infer its true IP address.

from Table 2. We hierarchically cluster the CSPs by horizontally cutting the tree at a given level. For example, we find five CSPs deployed on Amazon, which are marked with asterisks in Table 2.

4.2 Balancing Privacy and Reliability

Users specify CYRUS's privacy and reliability levels by setting the secret sharing parameters n and t for each chunk. The user first specifies the privacy level by choosing t, or the number of shares required to reconstruct a chunk: since we upload at most one share to each CSP, t specifies the number of CSPs needed for reconstruction. Taking $t = 2$ is sufficient to ensure that no one CSP can access any part of users' data, but more privacy-sensitive users may specify a larger t.

The user then specifies reliability in terms of an upper bound ϵ on the overall failure probability (i.e., the probability that we cannot download t shares of a chunk due to CSP failure). The failure probability of any given CSP, which we denote by p, is estimated using the number of consistent failed attempts to contact CSPs.[6] Users specify a threshold, e.g., one day, of time; if a CSP cannot be contacted for that length of time, then we count a CSP failure. The probability that we cannot download t shares equals the probability that more than $n - t$ CSPs fail, which we calculate to be $\sum_{s=0}^{t-1} C(n,s)\,(1-p)^s\,p^{n-s}$. We then bound this probability below ϵ by searching for the minimum n such that

$$\sum_{s=0}^{t-1} C(n,s)\,(1-p)^s\,p^{n-s} \leq \epsilon. \tag{1}$$

We find n by increasing its value from t to its maximum value (the total number of CSPs or clusters); taking the minimum such n limits the data stored on the cloud. The chunk share size is independent of n, so uploading n shares requires an amount of data proportional to n.

4.3 Downlink Cloud Selection

CYRUS chooses the n CSPs to which shares should be uploaded using consistent hashing on their SHA-1 hash values. To download a file, a client must choose t of the n CSPs from which to download shares. Since we choose $t < n$ for reliability, the number of possible selections can be very large: suppose that R chunks need to be downloaded at a given time, e.g., if a user downloads a file with R unique chunks. There are then $C(t,n)^R$ possible sets of CSPs, which grows rapidly with R. We thus choose the CSPs so as to minimize download completion times.

We suppose that file shares are stored at C CSPs. We index the chunks by $r = 1, 2, \ldots, R$ and the CSPs by $c = 1, 2, \ldots, C$, defining $d_{r,c}$ as an indicator variable for the share download locations: $d_{r,c} = 1$ if a share of chunk r is downloaded from CSP c and 0 otherwise. Denoting chunk r's share size as b_r, the total data downloaded from CSP c is $\sum_r b_r d_{r,c}$. We use β_c to denote the download bandwidth allocated to chunk c and find the total download time

$$\max_c \left(\sum_r \frac{b_r d_{r,c}}{\beta_c} \right). \tag{2}$$

While minimizing (2), the selected CSPs must satisfy constraints on available bandwidth and feasibility (e.g., selecting exactly t CSPs).

Bandwidth: The bandwidth allocated to the CSP connectors is restricted in two ways. First, each CSP has a maximum achievable download bandwidth, which may vary over time, e.g., as the CSP demand varies. We express these maxima as upper bounds: $\beta_c \leq \overline{\beta}_c$ for all CSPs c. Second, the client itself has a maximum download bandwidth, which must be shared by all parallel connections. We thus constrain

$$\sum_c \beta_c \leq \beta, \tag{3}$$

where β denotes the client's total downstream bandwidth.[7]

Feasibility: CYRUS must download t of shares of each chunk, and can only download shares from CSPs where they are stored. We thus introduce the indicator variables $u_{r,c}$, which take the value 1 if a share of chunk r is stored at CSP c and 0 otherwise. Our feasibility constraints are then

$$\sum_c d_{r,c} = t, \quad d_{r,c} \leq u_{r,c}. \tag{4}$$

CYRUS's download optimization problem is thus

$$\min_{y,d,\beta} y \tag{5}$$

$$\text{s.t.} \frac{\sum_r b_r d_{r,c}}{\beta_c} \leq y;\ c = 1, 2, \ldots, C \tag{6}$$

$$\sum_c \beta_c \leq \beta,\ \beta_c \leq \overline{\beta}_c,\ \sum_c d_{r,c} = t,\ d_{r,c} \leq u_{r,c} \tag{7}$$

Exactly solving (5–7) is difficult: first, the constraints on y are non-convex, and second, there are integrality constraints on $d_{r,c}$. We thus propose a heuristic algorithm that yields a near-optimal solution with low running time. Moreover, our algorithm is solved *online*: we iteratively select the CSPs from which each chunk's shares should be downloaded, allowing us to begin downloading chunk shares before finding the full solution. This allows us to outperform a heuristic that downloads shares from the CSPs with the highest available bandwidth. In that case, all chunk shares will be downloaded sequentially from the same t CSPs, so some

[6] We do not consider link failures, as CYRUS is designed to reduce the impact of failures at CSPs. Thus, CSP failure probabilities are taken as uniform, which is observed in practice within an order of magnitude [12, 34], and independent, since we choose CSPs with distinct physical infrastructure. If CSPs have different failure rates, we can simply set p to be the largest, so that we conservatively overestimate the probability of joint CSP failures.

[7] Each client maintains local bandwidth statistics to all CSPs for different network interfaces.

for $\eta = 1$ *to* R **do**
  Solve convexified, relaxed (5–7) with fixed CSP selections $d_{r,c}$ for $r < \eta$;
  Fix bandwidths β_c;
  Constrain $d_{\eta,c} \in \{0, 1\}$ for all c;
  Solve for the $d_{r,c}$ with fixed bandwidths;
  Fix CSP selections $d_{\eta,c}$;
end

Algorithm 1: CSP and bandwidth selection.

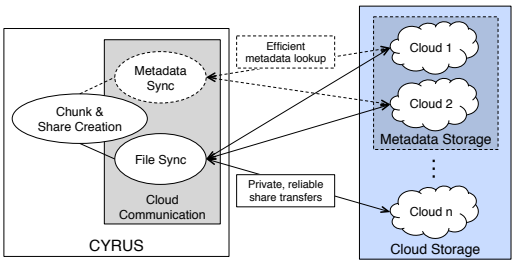


Figure 4: Decoupling metadata and file control.

shares will wait a long time before being downloaded. Our algorithm downloads shares in parallel from slower CSPs.

Our solution proceeds in two stages. First, we compute a convex approximation to (5–7) that does not consider the integer constraints on $d_{r,c}$. To do so, we first convexify (6) by defining $D_{r,c} \equiv d_{r,c}^{1/2}$. The resulting constraints $\sum_r b_r D_{r,c}^2/\beta_c \leq y$ are then convex, with the non-convexity moved to the new constraints $D_{r,c} = d_{r,c}^{1/2}$. These are approximated with linear over-estimators $\hat{D}_{r,c} \geq d_{r,c}^{1/2}$ that minimize the discrepancy between $\hat{D}_{r,c}^2$ and $d_{r,c}$.[8] We find that the closest linear estimator is $\hat{D}_{r,c} = 3^{1/4} d_{r,c}/2 + 3^{-1/4}/2$. On adding these constraints to (5–7) and replacing (6) with $\sum_r b_r \hat{D}_{r,c}^2/\beta_c \leq y$, we can solve the convexified version of (5–7) for the optimal $y, \beta_c, d_{r,c}$, and $\hat{D}_{r,c}$.

We then fix the optimal bandwidths β_c, turning (5–7) into a linear integer optimization problem that can be solved with the standard branch-and-bound algorithm. Branch-and-bound scales exponentially with the number of integer variables, so we impose integer constraints on one chunk's variables at a time; thus, only C variables are integral ($d_{r,c}$ for one chunk r). We constrain chunk 1's $d_{1,c}$ variables to be integral and re-solve (5–7), then re-solve the convex approximation, fix the resulting bandwidths, constrain $d_{2,c}$ to be integral, etc. Algorithm 1 formalizes this procedure.

5. CYRUS Operations

We now present CYRUS's operations realizing Section 3's architecture. To do so, we first note that CYRUS's performance depends on not only its CSP selection algorithm (Section 4) but also storage of file metadata (i.e., records of a file's component chunks and share locations). Since the metadata is both much smaller than the actual shares and accessed more often, we separate file and metadata control (Figure 4). We describe our file and metadata storage in Sections 5.1 and 5.2 respectively and then show how these are used to upload, download, and sync files at multiple clients.

[8] By requiring that these approximations be over-estimators, we ensure that if the constraints $\max_c \sum_r b_r \hat{D}_{r,c}^2/\beta_c \leq y$ hold, then (6) holds as well.

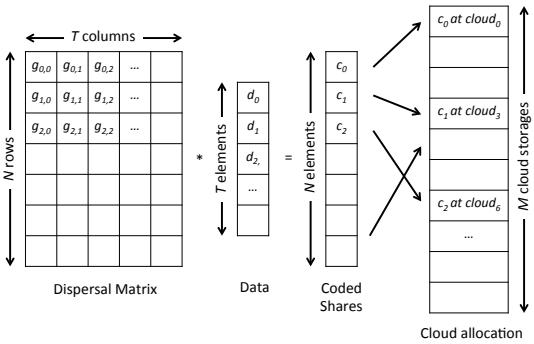


Figure 5: A non-systematic Reed-Solomon erasure code.

5.1 Storing Files as Shares

CYRUS divides files into chunks and then divides the chunks into shares, which are stored at CSPs.

Constructing file chunks: CYRUS divides a user's file into *chunks* based on content-dependent chunking, and stores only unique chunks to reduce the amount of data stored at CSPs. When a file is modified, content-dependent chunking only requires chunks to be modified if their contents are changed, unlike fixed-size chunking, which changes all chunks. We determine chunk boundaries using Rabin's Fingerprinting [33], which computes the hash value of the content in a sliding window w_i, where i is the offset on the file ranging from $0 \leq i < file\_size - window\_size$. When the hash value modulo a pre-defined integer M equals a pre-defined value K with $0 \leq K < M$, we set the chunk boundary and move on to find the next chunk boundary.

Dividing chunks into shares: After CYRUS constructs chunks from a file, it uses (t, n) secret sharing [36] to divide each chunk into n *shares*. We implement this secret sharing as a non-systematic Reed-Solomon (R-S) erasure code [28]. As shown in Figure 5, with R-S coding the coded shares $(c_0, c_1, ...)$ do not contain the original elements $(d_0, d_1, ...)$, so no data can be recovered from the file chunk with an insufficient number of shares. Indeed, R-S coding goes further than secret sharing: it can recover a chunk's data even if there are errors in the t shares used to reconstruct the chunk, with n/t times storage overhead.[9] To improve security, we take the R-S code's dispersal matrix as the Vandermonde matrix generated by a t-dimensional vector computed from a consistent hash of the user's key string. Since R-S decoding requires the dispersal matrix (Figure 5), reconstructing a chunk from a given set of shares $(c_0, c_1, \ldots)$ requires the key string.

Naming: We name each share as $H'(index, H(chunk.content))$, where H is the SHA-1 and H' any hash function. This naming scheme ensures that no CSP can discover the

[9] R-S codes need n times the storage space of the original data.

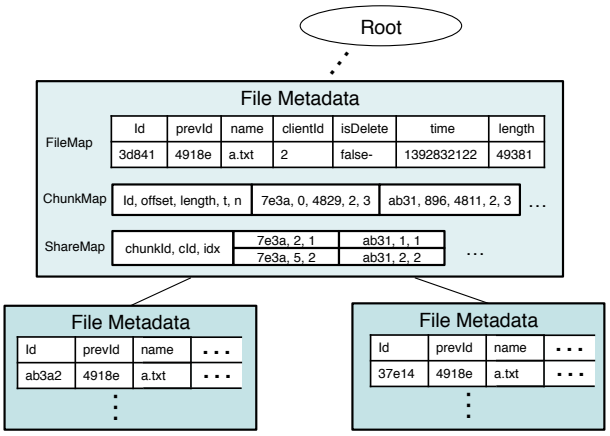


Figure 6: Metadata data structures.

index of a given share, but that it can be recovered by the client. Moreover, each share is guaranteed to have a unique file name, since a share's content is uniquely determined by the chunk contents, the share creation index, and t. Thus, we only overwrite the existing file share if its content is the same, reducing the risk of data corruption.

5.2 Metadata Data Structures

CYRUS maintains metadata for each file, which stores its modification history and share composition. The metadata are stored as a logical tree with dummy root node (Figure 6); subsequent levels denote sequential file versions. New files are represented by new nodes at the first level. Each node consists of three tables:

FileMap: This table stores the file `Id`, or SHA-1 hash of its content, and `prevId`, its parent node's ID (`prevID`$= 0$ for new files). We also store the `clientID`, indicating the client creating this version of the file, as well as the file name, whether it has been deleted, last modified time, and size.

ChunkMap: This table provides the information to reconstruct the file from its chunks. It includes the `Id`, or SHA-1 hash, of each chunk in the file; `offset`, or positions of the chunks in the file; the chunk sizes; and the t and n values used to divide the chunks into shares.

ShareMap: This table stores the shares' CSP locations. The `chunkId` is the chunk content's SHA-1 hash value, `idx` is the share index, and `cId` gives the CSP where it is stored.

In addition to file-specific metadata, CYRUS maintains a global chunk table listing the chunks whose shares are stored at each CSP. We store the metadata files using (t, m) secret sharing at a fixed set of m CSPs. Clients maintain local copies of the metadata tree for efficiency and periodically sync with the metadata stored at the CSPs.

5.3 Uploading and Downloading Files

Uploading files: Algorithm 2, illustrated in Figure 7, shows CYRUS's procedure for uploading files. The client first updates its metadata tree to ensure that it uses the correct parent node when constructing the file's metadata (steps 1 and

```
1  Function Upload(file)
2      head = getHead(file)
3      newHead = SHA1(file)
4      UpdateHead(file, head, newHead)
5      chunks = Chunking(file)
6      for chunk in chunks do
7          UpdateChunkMap(newHead, chunk)
8          Scatter(chunk)
9      end
       // Wait until uploading all chunks
10     UploadMeta(getMeta(file))
11 Function Scatter(chunk)
12     clouds = ConsistentHash(chunk.id)
13     if chunk is not stored then
14         shares = RSEncode(chunk, t, n)
15         for share in shares do
16             conn = clouds.next()
17             conn.Requester(PUT, share)// Async event
                   requester
18         end
19     end
```

Algorithm 2: Uploading files.

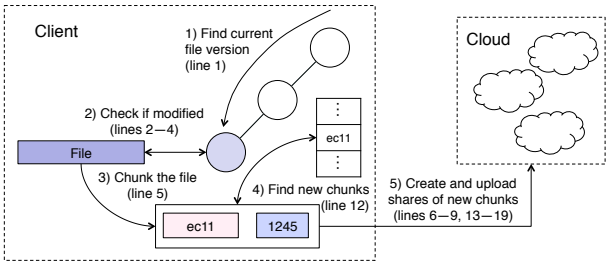


Figure 7: Uploading a file (lines refer to Algorithm 2).

2 in Figure 7). We then divide the file into chunks (step 3) and avoid uploading redundant chunks by checking whether shares of each chunk are already stored in the cloud (step 4). New chunks are divided into shares, as in Section 5.1, and the shares scattered to CSPs (step 5). CYRUS uses consistent hashing [20] to select the n CSPs at which to store shares of each chunk, allowing us to balance the amount of data stored at different CSPs and minimize the necessary share reallocation when CSPs are added or deleted (Section 5.5). The CSPs are selected by mapping the hash value of the chunk content to a position on the consistent hash ring, which is partitioned among CSPs. We then select the first n CSPs encountered while traversing the ring and send upload requests to the corresponding cloud connectors. The returns of the requests are handled asynchronously, as explained below. We upload the file metadata to CSPs (line 10 in Algorithm 2) only after receiving returns from all upload requests, so that no other client will attempt to download the file before all shares have been uploaded.

Downloading files: CYRUS follows Algorithm 3 to download a file. First, the client checks for the latest file version by sync-ing its metadata tree with the cloud (line 2). CYRUS then identifies the file's chunks and gathers their shares, selecting the download CSPs using Algorithm 1 (lines 3–5) and handling returns from the download requests

```
1  Function Download(file)
2      meta = downloadMeta(file)
3      for chunk in meta.chunkMap do
4          Gather(chunk, meta.shareMap[chunkId])
5      end
       // Wait until downloading all chunks
6      if checkConflict(meta) then
7          meta.conflict = True
8      end
9      updateSnapshot(file, snapshot)
10 Function Gather(chunk, shareMap)
11     clouds = OptSelect(chunk, shareMap)
12     for share in chunk.shares do
13         conn = clouds.next()
14         conn.Requester(GET, share)// Async event
               requester
15     end
```

Algorithm 3: Downloading files.

as asynchronous events (lines 12–15). Finally, CYRUS checks for file conflicts and prompts users to resolve them (lines 6–8); we elaborate on this step in Section 5.4.

Asynchronous event handling: Due to CSPs' different link speeds and shares' different sizes, CYRUS requires asynchronous event handling for uploads and downloads. Upon receiving response messages from CSPs (in most cases an HTTP response message with status code), CYRUS's cloud connectors send the events to a registered event receiver at the CYRUS core. The receiver can receive four types of share transmission events: GET META, PUT META, GET, and PUT; and maintains three boolean variables: ShareComplete, ChunkComplete, and FileComplete. ShareComplete is set to 1 if the share is successfully uploaded or downloaded, and ChunkComplete is set to 1 if n shares are successfully uploaded or t shares successfully downloaded. FileComplete is set to 1 if all chunks of a file are successfully uploaded or downloaded.

5.4 Synchronizing Files

Synchronization service: Clients sync files by detecting changes at their local storage and CSPs. Changes at the local storage can be detected by regularly checking last-modified times and file hash values. Changes at CSPs can be seen by looking up the list of metadata files stored in the cloud, since a new metadata file is created with each file upload (Algorithm 2). As explained in Section 5.2, the metadata are stored at a fixed subset of CSPs, making this lookup efficient.

Despite regular metadata sync-ing, nonzero network delays make it possible for two CYRUS clients to attempt to modify the same file at the same time. Clients cannot prevent such conflicts by locking files while modifying them; some CSPs do not support locking.[10] We instead assign unique names to uploaded shares as described in Section 5.1 and let clients upload conflicting file updates. Clients identify and resolve the resulting conflicts when downloading files (line 7 in Algorithm 3).

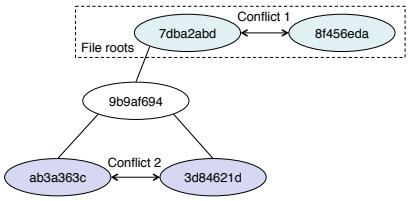


Figure 8: Two types of file conflicts.

Distributed conflict detection: We identify two types of file conflicts, as shown in Figure 8. First, two clients may create files with the same filename but different contents, resulting in different metadata file names, e.g., metadata `7dba2abd` and `8f456eda`. Second, one client can modify the previous version of a file due to delays in sync-ing metadata. We then find two child nodes from one parent, e.g., `ab3a363c` and `3d84621d` in the figure. Each node represents an independent modification of the parent node's file.

When new metadata is downloaded from the cloud, we check for conflicts by first checking if it has a parent node. If so, we check for the first type of conflict by searching for other nodes with the same filename. The second type of conflict arises if the new node has a parent. We traverse the tree upwards from this node, and detect a conflict if we find a node with multiple children.

File deletion and versioning: Clients can recover previous versions of files by traversing the metadata tree up from the current file version to the desired previous version. CYRUS also allows clients to recover deleted files by locating their metadata. When clients delete a file, CYRUS marks its metadata as "deleted," but does not actually delete the metadata file.[11] Shares of the file's component chunks are left alone, since other files may contain these chunks.

5.5 Adapting to CSP Changes

Over time, a user's set of viable CSPs may change: users may add or remove CSPs, which can occasionally fail. Thus, CYRUS must be able to adapt to CSP addition, removal, and failure. We consider these three cases individually.

Adding CSPs: A user may add a CSP to CYRUS by updating the list of available CSPs at the cloud. Once this list is updated, subsequently uploaded chunks can be stored at the new CSP. Moreover, the new CSP does not affect the reliability or privacy experienced by previously uploaded chunks. Since uploading shares to the new cloud can use a significant amount of data, we do not change the locations of already-stored shares. Shares of the file metadata can be stored at the new CSP, with appropriate indication in the list of available CSPs, if the user wishes to increase the reliability of the metadata storage.

[10] For instance, if two clients try to modify the same file at the same time, Dropbox allows both to create a locking file but changes one locking file's name. Thus, only one client can delete its locking file, allowing us to create a lock. On Google Drive, however, we cannot create a lock: both clients' locking files retain their original names and can be created and deleted.

[11] Since file metadata is very small, metadata from deleted files does not take up much CSP capacity.

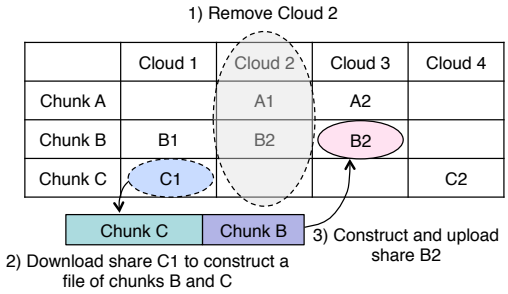


Figure 9: Share migration when removing a CSP.

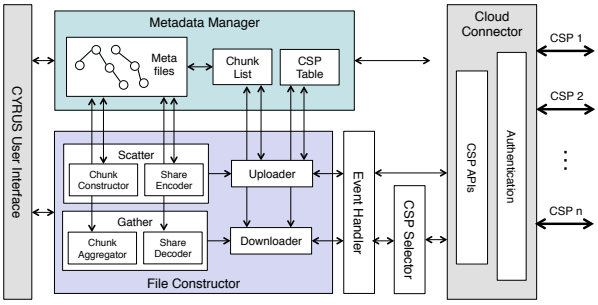


Figure 10: Client-based CYRUS implementation.

Removing CSPs and failure recovery: CYRUS detects a CSP failure if it fails to upload shares to that CSP; once this occurs, CYRUS periodically checks if the failed CSP is back up. Until that time, no shares are uploaded to that CSP. CSP removal can be detected by repeated upload failures or a manual signal from users. A failed or removed CSP is marked as such in the list of available CSPs.

Unlike adding a CSP, removing a CSP reduces the reliability of previously uploaded chunks and shares. Thus, to maintain reliability we must reconstruct the removed shares and upload them to other CSPs. We similarly migrate the file metadata that was stored on the deleted CSP. However, while the metadata is relatively small and can be migrated without excessive overhead, uploading all of the file chunk shares requires uploading a large amount of data. Indeed, from a user's perspective, a cloud may fail temporarily (e.g., for a few hours) but then come back up; it is therefore impractical to move all shares at once. Instead, we note that reliability matters most for frequently accessed shares and use a "lazy addition" scheme. Whenever a client downloads a file, we check the locations of its chunks' shares. Should one of these locations have been removed or deleted, we create a new share and upload it to a new CSP. Figure 9 shows the procedure of adding these shares: after a CSP is removed, a client downloads a file. If a share of one of the file's chunks was stored at the removed CSP, we reconstruct the share from the chunk and upload it to a new CSP.

6. CYRUS Implementation

We have prototyped CYRUS on three representative platforms (Mac OS X, Windows, and Linux) using Python and C, with the software architecture shown in Figure 10. The prototype has seven main components: 1) a graphical user interface, 2) C modules implementing content-based chunking and (t, n) secret sharing, 3) threads uploading and downloading contents to and from CSPs, 4) a metadata manager that constructs and maintains metadata trees, 5) an event handler for upload and download requests, 6) a cloud selector selecting CSPs for uploading and downloading shares, and 7) cloud connectors for popular commercial CSPs. Our CYRUS prototype consists of 3500 lines of Python code.

Our implementation integrates the CYRUS APIs (Table 3) and provides a graphical interface for people to use CYRUS. To increase efficiency, we use C modules to implement file chunking (Rabin's fingerprinting) and dividing chunks into shares (Reed-Solomon coding). We implement the cloud selection algorithm in Python.

To ensure transparency to different CSPs, we create a standard interface to map CYRUS's file operations to vendor-specific cloud APIs. This task involves creating a specific REST URL with proper parameters and content. We utilize existing CSP authentication mechanisms for access to each cloud, though such procedures are not mandatory. We have implemented connectors for Google Drive, DropBox, SkyDrive (now called OneDrive), and Box, with connectors to more CSPs planned in the future. These providers, as shown in Table 2, use standard web authentication mechanisms. Open source cloud connectors like Jclouds [4] can be used in place of CYRUS's own connectors.

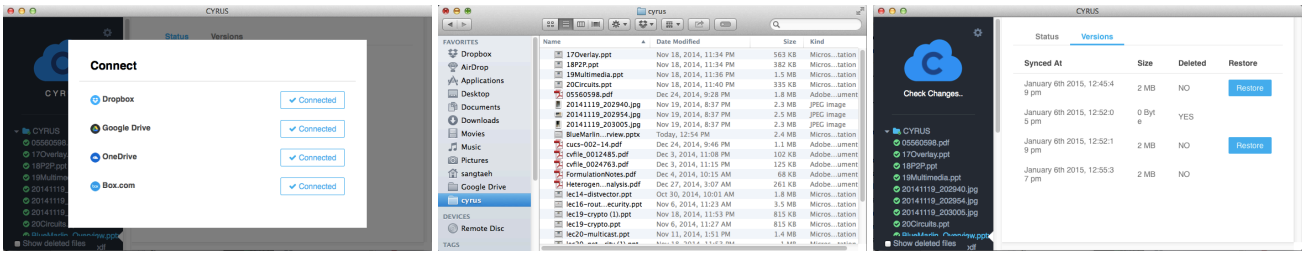
Figure 11 shows screenshots of our user interface. Figure 11a shows a list of CSP accounts connected to CYRUS, while Figure 11b lists the files and folders uploaded to CYRUS's directory. Figure 11c shows the history of a file within CYRUS; users can view and restore previous versions of each file. A demo video of the prototype is available [1].

7. Performance Evaluation

CYRUS's architecture and system design ensure that it satisfies client requirements for privacy, reliability, and latency. This section considers CYRUS's performance in a testbed environment before presenting real-world results from a comparison with similar systems and trial deployment.

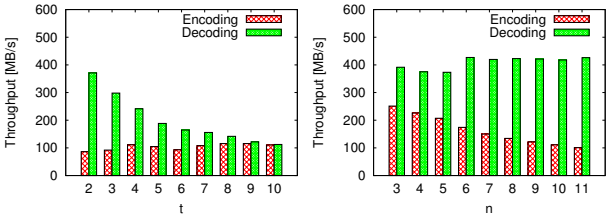
7.1 Erasure Coding

R-S decoding requires an attacker to possess at least t shares of a given file chunk as well as the dispersal matrix used to encode the chunk. Since we generate the dispersal matrix from hash values of the user's key string, recovering the dispersal matrix requires the user's key string, providing an initial layer of privacy. Even if the dispersal matrix is known, the attacker must still gain access to t of n shares of each file chunk. Since we store the shares on physically separate CSPs (Section 4.1), such a coordinated breach of users' CSP accounts is unlikely.



(a) Connected CSP accounts. **(b)** Files stored on CYRUS. **(c)** File history in CYRUS.

Figure 11: Screenshots of the CYRUS user interface.



(a) Changing t with n=11. **(b)** Changing n with t=2.

Figure 12: Empirical overhead of 100 MB chunk encoding and decoding while changing t and n.

Extension	# of files	Total bytes	Avg. size (bytes)
pdf	70	60,575,608	865,366
pptx	11	12,263,894	1,114,899
docx	15	9,844,628	656,309
jpg	55	151,918,946	2,762,163
mov	7	351,603,110	50,229,016
apk	10	4,872,703	487,270
ipa	4	47,354,590	11,838,648
Total	172	638,433,479	3,711,823

Table 4: Testbed evaluation dataset.

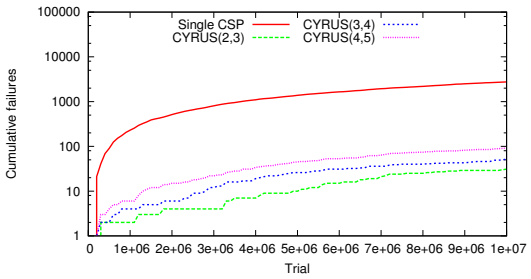


Figure 13: Simulated number of cumulative CSP failures.

If the attacker knows parts of the dispersal matrix or recovers $t' < t$ shares, it would be computationally expensive to guess the remaining information. The dispersal matrix is generated by a vector of length t, each of whose elements belongs to a finite field of size 2^m, where m is an integer encoding parameter chosen by the user. Similarly, the non-systematic R-S code can use t' shares to recover the original file up to $t - t'$ linear equations, or equivalently $t - t'$ unknowns. Since each unknown has 2^m possible values, an exhaustive search would be computationally expensive.

We implement the share encoding and decoding with the well-known zfec library [41]. Figure 12 shows the empirical overhead of the share encoding and decoding. As we would expect, a larger t leads to longer decoding times (i.e., lower throughput), with a minimum throughput of around 100MB/s for t=10. The encoding throughput depends more on n rather than t, reaching a minimum of around 100MB/s when n=11. In our experiments, we take (t, n) between (2,3) and (3,5), so the encoding and decoding throughputs are at least 200 and 300 MB/s respectively. The completion time bottleneck in our experiments below is therefore data transfer rather than encoding and decoding overhead.

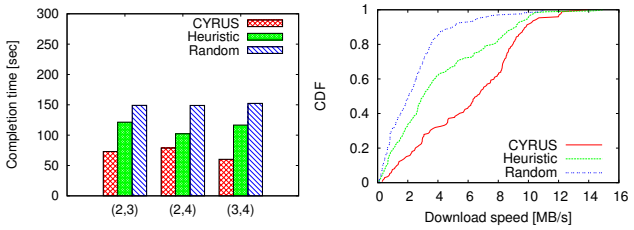
7.2 Privacy and Reliability

We first show that CYRUS improves CSP reliability in Figure 13, which shows the simulated number of cloud failures with single CSPs and CYRUS with different configurations. We base the simulations on real-world monitoring data from four commercial CSPs whose downtime varies from 1.37 to 18.53 hours per year [12]. At the end of a simulation with 10^7 trials, even the most reliable CSP returned approximately 1,500 failed requests, while CYRUS showed only 44 failures with $(t, n) = (3, 4)$ and no failures with $(t, n) = (2, 4)$. However, as we show in testbed experiments below, taking $(t, n) = (3, 4)$ improves upload and download completion times compared to other configurations.

Testbed setup: We construct a testbed with a MacBook Pro client and seven private cloud servers as our CSPs, connected with 1Gbps ethernet links. We emulate CSP cloud performance by using `tc` and `netem` to change network conditions for the different servers. We set maximum throughputs of 15MB/s for four cloud servers (the "fast" clouds) and 2MB/s for the remaining three clouds (the "slow" clouds).

We evaluate CYRUS's performance on a dataset of several different file types. Table 4 summarizes the number and size of the dataset files by their extensions. The total dataset size is 638.43 MB, with an average file size of 3.71 MB. We use content-based chunking to divide the files into chunks with an average chunk size of 4MB, following Dropbox [14].

Performance results: We first consider the effect of changing reliability and privacy on file download completion times. We test three configurations: $(t, n) = (2, 3)$, $(2, 4)$, and $(3, 4)$. Compared to the $(2, 3)$ configuration, $(t, n) = (2, 4)$ is more reliable, while $(t, n) = (3, 4)$ gives



(a) Mean completion time. **(b)** Throughput, $(t, n) = (2, 3)$.

Figure 14: Testbed download performance of random, heuristic, and CYRUS cloud selection.

more privacy. We compare the performance of three download CSP selection algorithms: CYRUS's algorithm from Section 4, random, and heuristic.[12] The random algorithm chooses CSPs randomly with uniform probability, and the heuristic algorithm is a round-robin scheme.

The download completion times are shown in Figure 14a. For all configurations, CYRUS's algorithm has the shortest download times. The random algorithm has the longest, likely due to the high probability of downloading from a slow cloud. Figure 14b shows the distribution of throughputs achieved by all files; we see that the throughput with CYRUS's algorithm is decidedly to the right of (i.e., is larger than) the random and heuristic algorithms' throughputs.

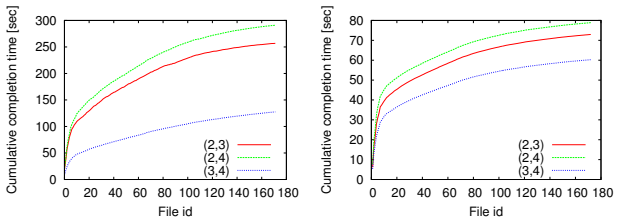
The completion time of CYRUS's algorithm when $(t, n) = (3, 4)$ is especially short compared to the other (t, n) values (Figure 14a). Since the share size of a given chunk equals the chunk size divided by t, higher values of t will yield smaller share sizes, lowering completion times for (parallel) share downloads. However, the random and heuristic algorithms' completion times do not vary much with the configurations. With $(t, n) = (3, 4)$, CYRUS must download shares from three clouds instead of two with the other configurations, increasing the probability that these algorithms will use slow clouds and canceling the effect of smaller shares.

Figure 15 shows the cumulative upload and download times for all files with CYRUS's algorithm. As expected, the more private $(3, 4)$ configuration has consistently shorter completion times, especially for uploads. The more reliable $(2, 4)$ and $(2, 3)$ configurations yield more similar completion times; their shares are the same size. The $(2, 4)$ configuration has slightly longer upload times since the shares must be uploaded to all 4 clouds, including the slowest ones.

7.3 Real-World Benchmarking

We benchmark CYRUS's upload and download performance on Dropbox, Google Drive, SkyDrive, and Box. We compare CYRUS to DepSky [7], a "cloud-of-clouds" system that also uses R-S coding to store files at multiple CSPs. DepSky's upload and download protocols, however, differ from CYRUS's: they require two round-trip communications with

[12] We did not compare the true-optimal performance, since finding the true optimum would require searching over $\sim C(t, n)^{300}$ possible download configurations, which is prohibitively expensive.



(a) Upload. **(b)** Download.

Figure 15: Testbed completion times of different privacy and reliability configurations.

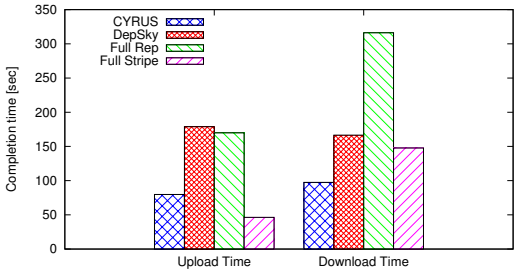


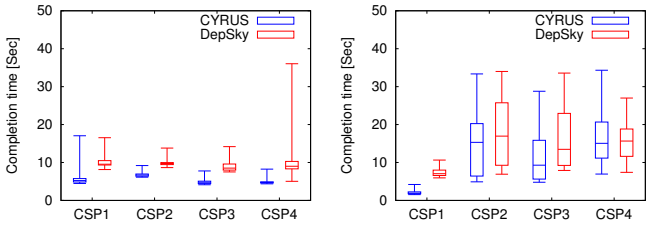
Figure 16: Completion times of different storage schemes.

CSPs to set lock files, preventing simultaneous updates, and a random backoff time after setting the lock. Moreover, DepSky uses a greedy algorithm that always downloads shares from the fastest CSPs, in contrast to CYRUS's optimized CSP selection. To compare the two systems, we implement DepSky within CYRUS.

Figure 16 shows the upload and download completion times for a 40 MB file with CYRUS, DepSky, and two other baseline storage schemes. Full Replication stores a 40 MB replica and Full Striping a 10 MB fragment at each of the four CSPs. Both CYRUS and DepSky use $(t, n) = (2, 3)$, so each share is 20 MB.[13] Since full striping uploads the least amount of data to each CSP, it has the shortest upload completion times. However, full striping is not reliable, as any CSP failure would prevent the user from retrieving the file. CYRUS has the second-best upload performance. DepSky's upload time is more than twice as long as CYRUS's and longer than Full Replication's, though Full Replication uploads twice as much data as DepSky to each CSP.

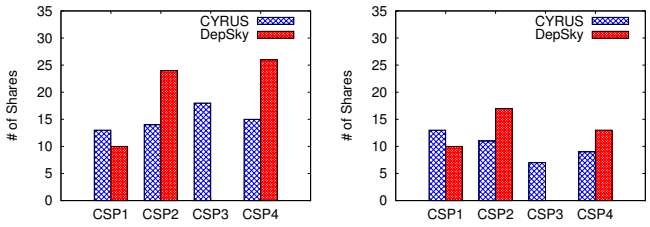
CYRUS's optimized downlink CSP selection algorithm allows it to achieve the shortest download completion time. DepSky shows longer completion times than either CYRUS or Full Striping, which must download from all four CSPs, including the slowest. Full Replication has the longest download time, since we averaged its performance over all four CSPs. Its download time would have been shorter (24.118 seconds) with the optimal CSP, but longer (519.012 seconds) with the slowest.

[13] For ease of comparison to full striping and full replication, we do not chunk the file with CYRUS or DepSky. We can thus verify that CYRUS's download CSP selection is perfectly optimal, as only $C(t, n)$ shares need to be downloaded from the CSPs.



(a) Upload. **(b)** Download.

Figure 17: Completion times with CYRUS and DepSky.



(a) Upload. **(b)** Download.

Figure 18: Share distribution with CYRUS and DepSky.

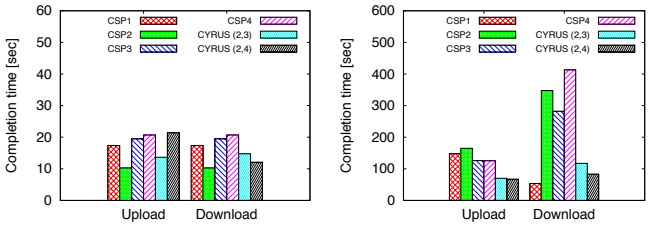
We next compare CYRUS's and DepSky's achieved upload and download completion times for a 1MB file, with upload and download measurements taken every hour for two days. Figure 17 shows box plots of the resulting completion times. CYRUS achieves significantly shorter completion times to all CSPs; DepSky's upload times are particularly large at nearly twice those of CYRUS.

Figure 18 shows the number of shares uploaded to each CSP. DepSky stores more shares at consistently faster CSPs: it starts uploads to all CSPs and cancels pending requests when n uploads complete, while CYRUS distributes shares evenly. CYRUS thus ensures that no one CSP will run out of capacity before the others, while DepSky's approach can cause one CSP to quickly use all of its capacity, hindering the distribution of shares to multiple clouds. Similarly, CYRUS spreads share downloads more evenly across CSPs.

7.4 Deployment Trial Results

We recruited 20 academic (faculty, research staff, and student) users from the United States and Korea to participate in a small-scale trial. We deployed trial versions of CYRUS for OS X and Windows that send log data to our development server. Over the course of the trial in summer 2014, we collected approximately 35k lines of logs. Since clients installed CYRUS on their laptops and desktops, we observed little client mobility within the U.S. or Korea; we thus report average results for each country and do not separate the trial results for different client locations within the countries.

We compare upload and download times for trial participants in the U.S. and Korea in Figure 19, which shows CYRUS's completion times to upload a 20 MB test file when connected to Dropbox, Google Drive, SkyDrive, and Box. We use $(t, n) = (2, 3)$ and $(2, 4)$, so that uploading a file



(a) U.S. **(b)** Korea.

Figure 19: Completion times during the trial.

to individual CSPs is neither as reliable nor as private as CYRUS.

In the U.S. (Figure 19a), CYRUS encounters a bottleneck of limited total uplink throughput from the client, slowing its connections to each individual CSP and lengthening upload completion time. When $(t, n) = (2, 3)$, CYRUS is still faster than all but one CSP. However, when $(t, n) = (2, 4)$, CYRUS uploads twice as much data in total than just uploading the file to one CSP; $(2, 4)$'s upload time is therefore longer than that of all single CSPs. In Korea (Figure 19b), connections to individual CSPs are much slower than in the U.S., so CYRUS does not encounter this client throughput bottleneck with either configuration. Since we upload less data to each CSP, both CYRUS configurations give shorter upload times than all individual CSPs.

CYRUS's download times for both configurations are shorter than those from individual CSPs. In both the U.S. and Korea, CYRUS has slightly longer download times than the fastest CSP, as it downloads shares from two CSPs. However, its download times are shorter than those from all other CSPs. Comparing the $(t, n) = (2, 3)$ and $(2, 4)$ configurations, we see that the (t, n) values affect download times more in Korea, due to the slower CSP connections there. Using $(t, n) = (2, 4)$ instead of $(2, 3)$ requires uploading an additional share, increasing the upload time in the U.S. by 7.78 seconds with little decrease in the download time. However, in Korea, using $(t, n) = (2, 4)$ does not appreciably increase the upload time and saves 33.805 seconds of download time. For any (t, n) values, CYRUS shortens the average upload time in Korea, where client bandwidth is not a bottleneck, and the average download time in both countries.

7.5 User Experience

After our trial, we informally surveyed the participants to understand their experience in using CYRUS and their thoughts on whether such a storage service would be useful in practice. While the survey represents a limited subset of academic users who are familiar with computer usage, their responses demonstrate CYRUS's potential to be practical and useful for the general population.

Most of our users had accounts at multiple CSPs but only used one or two regularly, likely due to the overhead in tracking which files were stored at which CSP. CYRUS thus allowed them to better utilize space on all of their cloud

accounts without additional overhead. In fact, two users in Korea thought CYRUS was faster than uploading files to individual CSPs, as is consistent with the upload results in Figure 19b. The remaining users in the U.S. and Korea did not notice any difference in upload or download speed with CYRUS. Users mainly used CYRUS to store documentation and image files, with almost all files $<$10 MB in size.

All but one user found CYRUS's interface easy-to-use. Users liked having a separate folder for CYRUS's files (Figure 11b), with one U.S. user saying that "*CYRUS is [as] good as any other application when we consider file dialogs.*" They also appreciated the ability to view files' history without accessing a web app (Figure 11c), with one user reporting that "*the fact that it[']s visible on the App [made] it easier to use and to access.*" Users reported that registering their CSP accounts with CYRUS was their main source of inconvenience; our prototype seeks to minimize this overhead by locally caching authentication keys so that users need only login to their CSPs once.

All users expressed interest in using CYRUS in the future, but made some suggestions for improvement. One user, for instance, suggested adding a feature to import files already stored at CSPs. The most common suggestion, made by nearly half of the users, was to implement CYRUS on mobile devices so that files could be shared between their laptops and smartphones. We plan to add mobile support in our future work.

8. Promoting Market Competition

While commoditization often decreases market competition by favoring large economies of scale, CYRUS may instead promote competition, as its privacy and reliability guarantees depend on users having accounts at multiple CSPs.

Without CYRUS, users experience a phenomenon known as "vendor lock-in:" After buying storage from one CSP, a user might not use storage from other CSPs due to the overhead in storing files at and retrieving them from multiple places. Since different CSPs enter the market at different times, vendor lock-in can thus lead to uneven adoption of CSPs and correspondingly uneven revenues. To combat vendor lock-in, many CSPs offer some free storage for individual (i.e., non-business) users. However, persuading users to later pay for more storage is still difficult, unless the new CSP offers much better service than the previous one. Business users, who do not receive free storage, have no incentive to join more than one CSP, exacerbating vendor lock-in.

CYRUS eliminates vendor lock-in for its users by removing the overhead of storing files at multiple CSPs. In fact, CYRUS encourages users to purchase storage at multiple CSPs, which increases its achievable reliability and privacy: users can store more chunk shares at different CSPs. Assuming comparable CSP prices, a given user might then purchase storage at all available CSPs, even-ing out CSP market shares. CSPs entering into the market would also be able to gain users. Some CSPs could gain a competitive advantage by providing faster connections or better coordination with CYRUS, but user demand would still exist at other CSPs.

Since CYRUS's secret sharing scheme increases the amount of data stored by a factor of n/t, users would need to purchase more total cloud storage with CYRUS. Total CSP revenue from business users might then increase, though it would likely be more evenly distributed among CSPs. CSP revenue from individual users, however, might decrease: some users could collect free storage from different CSPs without needing to purchase any additional storage. Thus, CYRUS may discourage CSPs from offering free introductory storage to individual users, in order to boost revenue.

9. Conclusion

CYRUS is a client-defined cloud storage system that integrates multiple CSPs to meet individual users' privacy, reliability, and performance requirements. Our system design addresses several practical challenges, including optimally selecting CSPs, sharing file metadata, and concurrency, as well as challenges in improving reliability and privacy. CYRUS's client-based architecture allows multiple, autonomous clients to scatter shared files to multiple CSPs, ensuring privacy by dividing the files so that no one CSP can reconstruct any of the file data. Moreover, we upload more shares than are necessary to reconstruct the file to ensure reliability. CYRUS optimizes the share downloads so as to minimize the delay in transferring data and simultaneously allows file updates from multiple clients, detecting any conflicts from the client. We evaluate CYRUS's performance in both a testbed environment and real-world deployment.

By realizing a client-defined cloud architecture, CYRUS flips the traditional cloud-servicing-clients model to one of clients controlling multiple clouds. CYRUS thus opens up new possibilities for cloud services, as similar systems may disrupt the current cloud ecosystem by commoditizing CSPs; thus, these services represent not just a technological, but also an economic change. These wide-ranging implications make client-defined cloud architectures an exciting area for future work.

Acknowledgments

This research was supported by the Princeton University IP Acceleration Fund and the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ICT/SW Creative Research program (NIPA-2014-H0510-14-1009) supervised by the NIPA (National IT Industry Promotion Agency).

References

[1] CYRUS demo video, 2014. http://youtu.be/DPK3NbEvdM8.

[2] Git merge, 2015. http://git-scm.com/docs/git-merge.

[3] ABU-LIBDEH, H., PRINCEHOUSE, L., AND WEATHERSPOON, H. Racs: A case for cloud storage diversity. In *Proc. of ACM SoCC* (New York, NY, USA, 2010), ACM, pp. 229–240.

[4] APACHE. What is apache JClouds?, 2013. http://jclouds.apache.org/.

[5] APACHE SOFTWARE FOUNDATION. HBase, 2014. http://hbase.apache.org/.

[6] ATTASENA, V., HARBI, N., AND DARMONT, J. Sharing-based privacy and availability of cloud data warehouses. In *9mes journes francophones sur les Entrepts de Donnes et l'Analyse en ligne (EDA 13), Blois* (Paris, Juin 2013), vol. B-9 of *Revues des Nouvelles Technologies de l'Information*, Hermann, pp. 17–32.

[7] BESSANI, A., CORREIA, M., QUARESMA, B., ANDRÉ, F., AND SOUSA, P. Depsky: Dependable and secure storage in a cloud-of-clouds. In *Proc. of ACM EuroSys* (New York, NY, USA, 2011), ACM, pp. 31–46.

[8] BESSANI, A., MENDES, R., OLIVEIRA, T., NEVES, N., CORREIA, M., PASIN, M., AND VERISSIMO, P. Scfs: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 169–180.

[9] BOWERS, K. D., JUELS, A., AND OPREA, A. Hail: A high-availability and integrity layer for cloud storage. In *Proc. of ACM CCS* (New York, NY, USA, 2009), ACM, pp. 187–198.

[10] CACHIN, C., AND TESSARO, S. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proc. of IEEE DSN* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 115–124.

[11] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[12] CLOUDSQUARE. Research and compare cloud providers and services, Mar. 2015. https://cloudharmony.com/status-1year-of-storage-group-by-regions.

[13] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proc. of ACM SOSP* (2007), ACM, pp. 205–220.

[14] DRAGO, I., MELLIA, M., M. MUNAFO, M., SPEROTTO, A., SADRE, R., AND PRAS, A. Inside Dropbox: Understanding personal cloud storage services. In *Proc. of ACM IMC* (New York, NY, USA, 2012), ACM, pp. 481–494.

[15] DUMON, M. Cloud storage industry continues rapid growth. Examiner.com, 2013. http://www.examiner.com/article/cloud-storage-industry-continues-rapid-growth.

[16] FINLEY, K. Three reasons why amazon's new storage service won't kill dropbox. Wired, 2014. http://www.wired.com/2014/07/amazon-zocalo/.

[17] GOODSON, G. R., WYLIE, J. J., GANGER, G. R., AND REITER, M. K. Efficient byzantine-tolerant erasure-coded storage. In *Proc. of IEEE DSN* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 135–.

[18] GUPTA, A. Outage post-mortem. Dropbox Tech Blog, 2014. https://blogs.dropbox.com/tech/2014/01/outage-post-mortem/.

[19] HU, Y., CHEN, H. C. H., LEE, P. P. C., AND TANG, Y. Nccloud: Applying network coding for the storage repair in a cloud-of-clouds. In *Proc. of USENIX FAST* (Berkeley, CA, USA, 2012), USENIX Association, p. 21.

[20] KARGER, D., SHERMAN, A., BERKHEIMER, A., BOGSTAD, B., DHANIDINA, R., IWAMOTO, K., KIM, B., MATKINS, L., AND YERUSHALMI, Y. Web caching with consistent hashing. *Comput. Netw. 31*, 11-16 (May 1999), 1203–1213.

[21] KEPES, B. At last some clarity about who is winning the cloud file sharing war... Forbes, 2014. http://www.forbes.com/sites/benkepes/2014/03/03/user-numbers-or-revenue-for-cloud-file-sharing-vendors-which-is-most-important/.

[22] LANGO, J. Toward software-defined slas. *Commun. ACM 57*, 1 (Jan. 2014), 54–60.

[23] LING, C. W., AND DATTA, A. InterCloud RAIDer: A do-it-yourself multi-cloud private data backup system. In *Distributed Computing and Networking*. Springer, 2014, pp. 453–468.

[24] MACHADO, G., BOCEK, T., AMMANN, M., AND STILLER, B. A cloud storage overlay to aggregate heterogeneous cloud services. In *Local Computer Networks (LCN), 2013 IEEE 38th Conference on* (Oct 2013), pp. 597–605.

[25] MALKHI, D., AND REITER, M. Byzantine quorum systems. In *Proc. of ACM STOC* (New York, NY, USA, 1997), ACM, pp. 569–578.

[26] MALKHI, D., AND REITER, M. K. Secure and scalable replication in Phalanx. In *Proc. of IEEE SRDS* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 51–.

[27] MARTIN, J.-P., ALVISI, L., AND DAHLIN, M. Minimal Byzantine storage. In *Proc. of DISC* (London, UK, UK, 2002), Springer-Verlag, pp. 311–325.

[28] MCELIECE, R. J., AND SARWATE, D. V. On sharing secrets and Reed-Solomon codes. *Communications of the ACM 24*, 9 (1981), 583–584.

[29] MU, S., CHEN, K., GAO, P., YE, F., WU, Y., AND ZHENG, W. μ-libcloud: Providing high available and uniform accessing to multiple cloud storages. In *Proc. of ACM/IEEE GRID* (Washington, DC, USA, 2012), IEEE Computer Society, pp. 201–208.

[30] PAPAIOANNOU, T. G., BONVIN, N., AND ABERER, K. Scalia: An adaptive scheme for efficient multi-cloud storage. In *Proc. of IEEE SC* (Los Alamitos, CA, USA, 2012), IEEE Computer Society Press, pp. 20:1–20:10.

[31] PATTERSON, S. Personal cloud storage hit 685 petabytes this year. WebProNews, 2013. http://www.webpronews.com/personal-cloud-storage-hit-685-petabytes-this-year-2013-12.

[32] PERLROTH, N. Home Depot data breach could be the largest yet. The New York Times, 2014. http://bits.blogs.nytimes.com/2014/09/08/home-depot-confirms-that-it-was-hacked/.

[33] RABIN, M. *Fingerprinting by Random Polynomials*. Center for Research in Computing Technology. Aiken Computation Laboratory, Univ., 1981.

[34] RAWAT, V. Reducing failure probability of cloud storage services using multi-cloud. Master's thesis, Rajasthan Technical University, 2013.

[35] RESCH, J. K., AND PLANK, J. S. AONT-RS: Blending security and performance in dispersed storage systems. In *Proc. of the 9th USENIX FAST* (Berkeley, CA, USA, 2011), USENIX Association, pp. 14–14.

[36] SHAMIR, A. How to share a secret. *Commun. ACM 22*, 11 (Nov. 1979), 612–613.

[37] STONE, B. Another Amazon outage exposes the cloud's dark lining. Bloomberg Businessweek, 2013. http://www.businessweek.com/articles/2013-08-26/another-amazon-outage-exposes-the-clouds-dark-lining.

[38] STRUNK, A., MOSCH, M., GROB, S., THOB, Y., AND SCHILL, A. Building a flexible service architecture for user controlled hybrid clouds. In *Proc. of ARES* (Washington, DC, USA, 2012), IEEE Computer Society, pp. 149–154.

[39] WAKABAYASHI, D. Tim Cook says Apple to add security alerts for icloud users. The Wall Street Journal, 2014. http://online.wsj.com/articles/tim-cook-says-apple-to-add-security-alerts-for-icloud-users-1409880977.

[40] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proc. of OSDI* (Berkeley, CA, USA, 2006), USENIX Association, pp. 307–320.

[41] WILCOX-OHEARN, Z. zfec package 1.4.24. https://pypi.python.org/pypi/zfec.