DCS: Secure Distributed Storage with DataCapsules

Ted Lin
UC Berkeley

Samuel Berkun
UC Berkeley

Saketh Malyala
UC Berkeley

ABSTRACT

DataCapsules are a recently proposed data container model with built-in security properties, intended to be a standardized abstraction unit for secure portable storage over untrusted, heterogeneous distributed infrastructure. We present *DCS*, a DataCapsule server design and implementation. DCS supports efficient integrity proof generation through the use of cryptographic hash chaining, client-server mirrored verification caching, and a simple globally monotonic heuristic. DCS also supports replication for durability, availability, and scalability. We introduce a novel anti-entropy algorithm for faster convergence toward consistency across DataCapsule replicas. Finally, DCS supports two alternative client APIs, including a commit-based API that can enable accessing and verifying "sub-records" with improved performance.

DCS demonstrates performance that is not drastically different from the production-quality key-value store RocksDB on target workloads such as YCSB. DCS also scales to handle 1000s of concurrent clients.

1 INTRODUCTION

In the modern computing world, there are many options for distributed storage, ranging from services managed by large-scale cloud businesses such as Amazon Web Services and Microsoft Azure [4, 15] to decentralized "edge" or "fog" resources administered by individuals, municipalities, and other organizations [6, 21]. These storage options, particularly those on the edge, have varying levels of security guarantees [21], often relying on "border security" mechanisms that fail to fully protect against the myriad potential attack vectors [17]. Furthermore, these storage options are often incompatible with each other due to the heterogeneous nature of the edge [21] and the "vendor lock-in" economic incentive of cloud providers [22]. For example, when migrating between different cloud storage services, one would need to manually configure each service separately to achieve the same desired security properties, which is an inefficient and error-prone process. The above issues suggest that, although there is universal demand for it, shareable/portable secure distributed data storage is currently not straightforward for the average user to correctly set up and maintain.

To address these problems, a *data-centric* approach to security and distribution has recently been proposed [16, 17]. The key abstraction behind this approach is the *DataCapsule*, a general data container with built-in security properties. DataCapsules are analogous to the standardized physical freight containers that are transported worldwide by cargo ships, trains, and trucks. Each DataCapsule includes all metadata required for globally unique identification of the DataCapsule, access control over its contents with confidentiality guarantees, and verification of the integrity and provenance of its contents. A DataCapsule's metadata and contents are fully integrated with each other by immutable cryptographic hash linking and are stored as a single entity, which means the DataCapsule effectively serves as a standalone portable unit of storage that can maintain security guarantees over untrusted, heterogeneous distributed infrastructure.

At a high level, the DataCapsule model consists of an append-only directed acyclic graph of signed immutable *records* linked together by cryptographically-secure hashes. This is a thin and flexible low-level data model that many higher-level systems and applications can be structured around, including key-value stores (e.g. CapsuleDB [18]) and "Function as a Service" platforms (e.g. Paranoid Stateful Lambdas [7]).

While DataCapsules are a well-developed idea in theory and have already been used as the basis for several experimental systems, there does not yet exist a concrete realization of a production-level DataCapsule storage plane that DataCapsule-based systems and applications can be built on top of. Challenges for such a storage plane include:

- supporting low-power devices like sensors and mobile gadgets as clients.
- minimizing the performance overhead of generating proofs of integrity and provenance.
- supporting replication of DataCapsules across servers for increased durability, availability, and scalability.

2 BACKGROUND AND RELATED WORK

In this section, we describe the DataCapsule model based on prior work [16, 17]. We also make connections with other existing work.

2.1 Threat Model

At creation time, each DataCapsule defines its *authorized writers*, i.e. entities that have a private key that is associated with one of the public keys attached to the DataCapsule. Authorized writers are assumed to be trusted. Note that if an authorized writer is found to be malicious, then the provenance guarantees of the DataCapsule facilitate verifiably linking their authored records back to them.

Readers who are given decryption keys for the content of a DataCapsule are assumed to be trusted. If they are not, then the basic confidentiality guarantees of the DataCapsule break down. Note that the DataCapsule preserves integrity and provenance guarantees even in the presence of malicious readers with the ability to decrypt its content.

The infrastructure over which DataCapsules are served is generally assumed to be untrusted. For example, servers are free to reorder incoming requests, serve corrupt data to clients or other servers, and maliciously respond to control messages.

One important property that we do not address in this paper is protection against denial of service. We do not guarantee forward progress in the presence of network attacks and malicious servers. In practice, the economic incentives of most service providers, the use of a federated network such as the Global Data Plane [16, 17], and *freshness queries* (which we support as described in Section 3) help mitigate this problem. However, making *guarantees* against

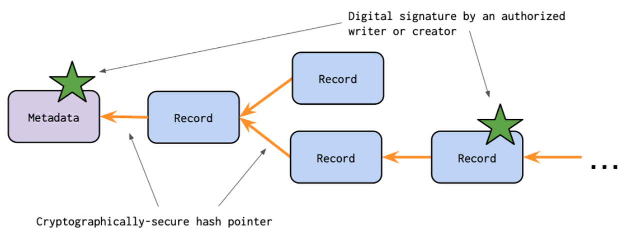

Figure 1: Model of internals of a DataCapsule.

denial of service is out of scope for this paper, and we leave it to future work.

2.2 DataCapsule Model

As mentioned in Section 1, each DataCapsule includes immutable metadata that is sufficient to globally identify the DataCapsule, manage confidential access to its contents, and anchor proofs of integrity and provenance. Metadata may include a human-readable description of the DataCapsule, the public keys that authorized writers use to digitally sign contents, a description of how readers can securely request and receive content decryption keys (a process that is external to the DataCapsule model), a physical timestamp of creation, and more. Each DataCapsule is globally identified by the 256-bit cryptographically-secure hash of its metadata - we refer to this as a *DataCapsule name*. Note that the use of a cryptographically-secure hash here helpfully creates a tamper-proof association between the DataCapsule name and its authorized writers.

The base unit of content that clients can write to or read from a DataCapsule is the *record*. Records are individually encrypted for confidentiality and digitally signed by authorized writers for integrity and provenance. A record itself has no inherent structure; the higher-level systems and applications built on top of DataCapsules can decide what they want individual records to represent. For example, an individual record can represent anything from a partial database index segment to an entire large file (though we note this is probably undesirable for performance reasons).

An important property of records is that they are immutable - i.e. records that have already been made durable cannot be modified or deleted. This helps streamline verification of integrity and provenance. This also makes handling replication easier, as we describe in Section 2.3.

As records are immutable, a DataCapsule can only be modified by *appending* records. Assuming a single designated writer at any given time and absence of failures, this results in a linear totally-ordered log history of records. However, the requirement of designating a single writer at any given time (e.g. through a separate coordination or serialization service) is overly restrictive for many applications and may be impractical in edge / distributed settings [8]. Furthermore, failures are hard to avoid, especially in the context of untrusted distributed infrastructure. For example, if a writer client crashes and loses track of its most recently written record, the untrusted infrastructure may unintentionally or maliciously return stale records. Subsequently appending to such stale records would result in an inconsistent state under the strict linear model. Therefore, DataCapsules explicitly allow branching in record histories, creating a well-defined partial-ordering over

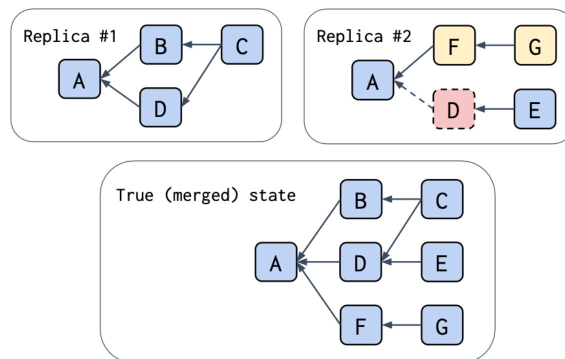

Figure 2: Two replicas of the same DataCapsule can become inconsistent. When compared against Replica #1, Replica #2 has a *hole* (record "D"). When compared against Replica #2, Replica #1 has a *stale branch* (record "E") and a *missing branch* (records "F" and "G").

written records in the form of a directed acyclic graph (DAG) and allowing for strong eventual consistency guarantees that we describe in detail in Section 2.3.

Records are linked by cryptographically-secure hashes. This has two useful consequences:

- Causal relationships in the partially-ordered history of records are efficiently represented, as opposed to alternatives e.g. vector clocks [13] which introduce problems related to size.
- Proving the integrity of a given record does not require a direct digital signature for that specific record. A cryptographically-secure hash chain to any previously verified record acts as a sufficient proof of integrity [14]. This forms the basis of several important performance optimizations that we explore throughout the rest of this paper.

2.3 Replication and Anti-Entropy

The ability to replicate DataCapsules across servers opens doors to increased durability, availability, and scalability. However, replication also introduces classic distributed consistency problems. In particular, when a server crashes or loses connection to the rest of the network, its physical DataCapsule copies may become out of sync with other replicas. A desirable guarantee for such situations is *strong eventual consistency*, which requires that "correct replicas that have (eventually) delivered the same updates have equivalent state" [20]. Following terminology established in other seminal work [8, 19], we refer to algorithms that progress towards convergence across replicas as *anti-entropy* algorithms.

Conveniently (by design), the immutability of records and append-only nature of DataCapsules implies that achieving strong eventual consistency is logically trivial - the order in which updates arrive to a recovering replica does not matter since set-unioning is a commutative operation. Therefore, a trivially-correct naive anti-entropy algorithm for any given DataCapsule would be to: (1) periodically pair up replicas, (2) have each replica send their entire local copy of the DataCapsule to the other server in its pair, and (3) have each replica union its local copy with the remote copy received by the other server. Note that since servers are assumed to be mutually untrusted, honest servers should verify the authenticity of

data received from other servers to avoid wasting storage space on corrupted data.

In this naive algorithm, the back-and-forth sending of entire DataCapsule copies is clearly inefficient, especially when considering that (1) many of the potential use-cases for DataCapsules involve large record histories, and (2) inconsistencies across replicas only arise as a consequence of infrastructure failures, which means that replicas are expected to have almost the same state most of the time. Prior work on DataCapsules shows that the naive algorithm can cause significant stress on the underlying network and slow convergence times, resulting in undesirable "staleness" of the data being returned from read requests [12].

Existing work on general anti-entropy (not specific to DataCapsules) offers alternative approaches. One approach, adapted from Amazon Dynamo [8], is to treat all records in a DataCapsule as a flat set and build a separate Merkle tree over this set. To achieve convergence between two replicas, start by having the replicas exchange and compare the roots of their respective Merkle trees. If their roots are the same, then we know the two replicas have exactly the same state, and we are done. Otherwise, we know that the two replicas are inconsistent, and we move on to have the replicas exchange and compare the tree nodes on the next level of their respective Merkle trees. By recursively repeating this process, we drill down to find which records are missing from each replica. When compared to the naive algorithm, this approach has the advantage of sending less data overall to achieve convergence. However, a key disadvantage is that it can require many rounds of network communication in the worst case, which is exacerbated by the fact that DataCapsules are intended to operate over untrusted infrastructure.

Prior work on DataCapsules includes an anti-entropy algorithm that aims to address the issues mentioned above [12]. Since our own algorithm builds on this one, we include its summary and our interpretation of it as part of Section 3.4.

3 DCS DESIGN

3.1 DataCapsule Representation

Here we describe how the DataCapsule model introduced in Section 2.2 is materialized in DCS. Note that from this point forward, any mention of "hash" assumes a cryptographically secure hash function e.g. SHA-256.

Each logical record consists of two physical components: a *header* and a *body* of (encrypted) application content. We make this physical distinction for performance reasons, e.g. integrity verification through hash chaining should not require downloading potentially-large application contents.

We define a *record name* as the hash over a record header. A record's name can also be thought of a *hash pointer* to that record. "Signing a record" just means creating a digital signature on the record's name.

A record's header consists of the DataCapsule name, a hash pointer to the associated body, and a variable number of *backpointers* i.e. hash pointers to a selection of earlier records. The hash pointer to the associated body "completes the chain" of integrity from signature to application content. The backpointers to earlier records serve two purposes: (1) establishing causality and (2) enabling hash-chained proofs of integrity (for those earlier records). Allowing the number of backpointers to be variable similarly serves two purposes: (1) creating well-defined points at which branches in the record history "merge" according to application-specific semantics and (2) improving the efficiency of integrity proof generation and verification by giving earlier records an opportunity to "reach" a record with a signature in fewer hash link "hops".

3.2 API

The intended clients of DCS include a range of higher-level systems and applications with different requirements. With this in mind, we define a low-level API with the goal of being a flexible base, noting that middleware optimized for application-specific ergonomics and performance can be built on top of DCS in the future.

Writer API

- *Create*(Metadata): creates a new DataCapsule.
- *Init*(DataCapsuleName): initializes a session/batch of requests that refer to the given DataCapsule name.
- *Write*(Record): persists the given record to the DataCapsule referred to by the containing context/session/batch. The "Record" input here refers to a logical record that is fully materialized as described in Section 3.1, including both body and header. Note that this is exactly the "append" operation with respect to the record history DAG.
- *Sign*(RecordName, WriterSignature): persists the given signature for the given record name to the DataCapsule referred to by the containing context/session/batch.

For a write request of any type, each server is expected to digitally sign (with the server's own asymmetric key pair) an acknowledgement in response. These acks are for durability purposes. As in other quorum-based systems [8], clients can configure the threshold of acks required for a write request to be considered complete. Higher thresholds result in increased durability at the cost of increased latency or time spent blocking. Lower thresholds represent optimistic writing (with little to no blocking) at the risk of introducing logical holes with higher probability.

As an aside, prior work proposes offloading the collection and verification of acknowledgements to proxies in secure enclaves [12]. In this work, we treat those proxies as an implementation detail of the client and do not discuss them further.

Reader API

- *Init*(DataCapsuleName): initializes a session/batch of requests that refer to the given DataCapsule name.
- *Read*(RecordName) -> Record: reads the logical record with the given name.
- *Prove*(RecordName) -> Proof: generates a proof of integrity for the named record. A proof can come in multiple forms, including (1) a direct signature for the named record, (2) a hash chain of record headers beginning at a record with a direct signature and ending at the named record, or (3) a hash chain of record headers beginning at a previously verified record and ending at the named record.
- *QueryFresh*() -> Set<RecordName>: returns the set of heads in the record history DAG. The reader can collect the responses of a high quorum of servers in order to be as up-to-date as possible.

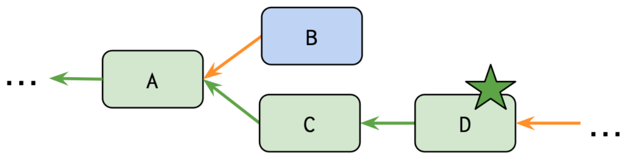

Figure 3: The secure hash chain from (unsigned) record-to-prove "A" to a proven descendant record "D" is a sufficient proof for the integrity of "A".

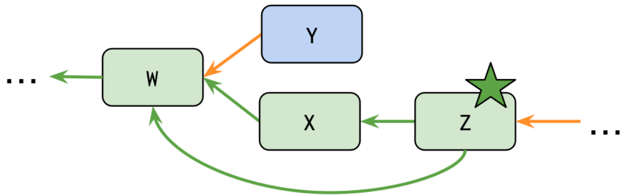

Figure 4: Since record "Z" has an additional backpointer directly to record "W", there are two equally valid hash chain integrity proofs for "W" from "Z". The obvious choice is to use the shorter hash chain.

3.3 Integrity Proofs

A straightforward approach to ensuring integrity is to sign and verify every individual record. However, as our measurements in Section 6 show, this approach comes with a nontrivial performance overhead, especially in contexts where data is being streamed at a fast rate. Though this performance overhead may be tolerable for some applications, it is unacceptable for many of the potential types of writer clients that will interact with DataCapsules, including low-power / Internet-of-Things devices such as sensors and mobile gadgets. By building on the fundamental idea of secure hash chaining, DCS supports the ability to efficiently generate and verify proofs of integrity while only having periodic records be writer-signed.

To prove the integrity of any given record, a secure hash chain of record headers from the given record to any verifiable (e.g. through a signature) record is sufficient. Recall from Section 3.1 that records can have multiple backpointers. In particular, a record can have additional backpointers whose sole purpose is to shorten the length of hash chaining required for earlier records to reach some record with a signature. This feature, along with the presence of branches and holes due to replication (see 2.3), implies that there can be multiple valid hash chains from the record-to-verify to some record with a signature. Upon receiving a *Prove* request, the server must choose one of these valid hash chains to return.

An obvious heuristic is to choose the shortest hash chain, which can be accomplished by a breadth-first-search on the reversed record history DAG starting from the record-to-prove. However, as intuition suggests, we observe that executing a breadth-first-search on every *Prove* request induces significant overhead, particularly for applications that need to support many concurrent readers or common OLAP workloads that make up a significant population of potential DataCapsule users.

To address this, DCS explicitly tracks the shortest proof for every record in a DataCapsule. This is materialized in implementation by associating every written record with a *witness*, which is a variant type that either (1) contains a direct signature for the record, (2) contains a hash pointer (opposite to the direction of flow in the record history DAG) to the next record in the shortest path to some directly verifiable record, or (3) represents a record whose integrity cannot yet be proven. Variant (2) also explicitly includes the length of said shortest path, so that it can be updated efficiently. Variants (1) and (3) imply shortest path lengths of zero and infinity, respectively. This tracking is eagerly updated on every *Sign* request, e.g. by dispatching to background worker threads after making the signature itself durable. Updating consists of a pruned breadth-first-search on the (unreversed) record history DAG starting from the record-just-signed.

The heuristic of shortest path length (in an append-only DAG) also has the nice property of having updates that are logically monotonic. For any given record, the shortest path from that record to a signed record can only ever get shorter as more records get signed. By the CALM Theorem [3], this implies that the heuristic is guaranteed to be eventually consistent without needing coordination, i.e. it is safe to propagate updates across servers using whichever methods are most efficient, e.g. gossip protocols [9, 11].

We next observe that it does not matter *how* the previously verified record was verified - it could have been signed itself, but it also could have been recursively verified by another hash chain proof. This motivates the idea of reader-side caching of records that the reader has already verified. For space efficiency reasons, note that the reader does not need to store entire records in this *verification cache* - just the names (secure hashes) of the previously verified records are sufficient.

In order to fully realize the performance improvement potential offered by reader-side verification caching, the server should be aware of what is already in the requesting reader's cache. DCS takes the approach of *client-server mirrored caching*, where the server maintains complete mirrors of each active reader's verification cache in order to generate shorter proofs tailored to the specific requesting reader. By "complete mirror", we mean that all cache characteristics (including size, eviction policy, and other implementation details) are exactly identical on both the client and server, such that if the client and server apply the same sequence of operations, their cache states will end up consistent. This can be achieved by having the reader send a detailed specification of its verification cache configuration when initializing a connection to a server. The reader can continue using the same verification cache across different sessions by also sending the current set of cached record names to the server upon session initialization.

The basic concept of cache mirroring does not specify synchronization between client and server (other than at initialization time). In particular, without additional mechanisms in place, the server-side cache can "run ahead" of the reader-side cache by adding a record for which it has successfully generated a proof before the reader is able to receive and verify the proof. Under the assumption of no failures, this lack of synchronization is acceptable since mirrored execution guarantees that the server will never send a proof rooted at a record that the client does not have in its verification cache. However, in cases e.g. where a client temporarily failed to receive proof responses from the server, the client indeed may not have verified the records that the server uses as the root of subsequent proofs. Assuming the absence of such failures is impractical, so we add a client acknowledgement step which lets the server know when the client has successfully finished verifying a record

and added it to their cache, at which point the server can apply the update to its own mirroring cache.

As shown in Section 6, client-server mirrored caching (when functioning as intended) achieves the clearest potential performance gain in DCS by far. However, we note a couple of important disadvantages of this approach that were discovered over the course of its ideation and development. First, cache mirroring only works with an honest server implementation. Malicious servers can cause denial of service by intentionally messing with their own execution of the protocol (though note that denial of service is outside of our threat model established in Section 2.1). Even with the assumption of an honest server, fully guaranteeing the correctness of cache mirroring in practice is tricky due to the common issues that arise in distributed systems and the fact that cache operations are not commutative (assuming bounded cache size). In future work, we look to establish correctness in a more complete and formal way. Note that the correctness of cache mirroring only affects the performance of integrity verification - an incorrect mirrored cache never violates the security guarantees provided by DataCapsules, since ultimately the reader has control over which records it considers as verified.

3.4 Anti-Entropy

Building on our study of existing anti-entropy work in Section 2.3, we make the following observation: Given how straightforward it is to merge missing records into local DataCapsule state, the problem of anti-entropy for DataCapsules reduces to how a recovering replica "A" can create a compact digest of its local state to send to its paired replica "B", such that "B" can quickly figure out which records it has that "A" is missing. Indeed, the current state-of-the-art anti-entropy algorithm for DataCapsules, which we now refer to as *Lu's algorithm*, fits into this perspective.

First, we establish terminology that helps us be more precise in our specifications and subsequent implementations. Note that the rest of our discussion of anti-entropy is in the context of a single (arbitrary) DataCapsule. We say a record hash pointer (in the record history DAG) is a *physical pointer* w.r.t. a given replica if both the record it originated from and the record it points to are present in the replica. A replica's *heads* are defined as records that are present in the replica and have no incoming physical pointers. Similarly, a replica's *roots* are defined as records that are present in the replica and have no outgoing physical pointers. We emphasize the "physical" distinction here since it is the source of most of the critical bugs we discovered in prior implementations of Lu's algorithm.

We now describe Lu's algorithm:

1. Have replicas periodically pair up to perform pairwise synchronization.
2. Let "A" and "B" be the names of two arbitrary replicas that have been paired up. Have "A" and "B" exchange their respective heads and roots.
3. Having received the heads and roots of "A", "B" can perform traversals over its own record history DAG to determine which records "A" is missing. These can be categorized as:
   - (a) *Stale branches* in "A", detected when heads of "A" are present in "B" but are not heads in "B". Note that patching these is similar to the "fast-forward merge" in Git [10] and requires "B" to traverse its DAG with reversed edges.
   - (b) *Holes* in "A", detected when roots of "A" are present in "B" but are not roots in "B".
   - (c) *Missing branches* in "A", detected when heads/roots of "B" are neither heads/roots of "A" nor records collected in the previous two cases.

Upon creating our own implementation of Lu's algorithm, we observed that the time-to-converge for any given pair of replicas was bottlenecked by the need to retrieve local records sequentially as part of the DAG traversals. This is especially apparent when servers are taken offline for significant periods of time, resulting in long chains of holes and stale/missing branches. Since this situation is expected to be a common mode of failure, we developed a novel anti-entropy algorithm that takes this into account.

Our algorithm views all records in the DataCapsule as a flat set, similar to the Dynamo-based approach described in Section 2.3. This allows us to do all record accesses in a batched or parallel-mapped fashion and sidesteps the performance bottleneck of long graph traversals in Lu's algorithm.

Our algorithm is based on the observation that a compact *estimated* digest of the local state of "A" can be formed by: (1) having "A" keep track of which records "A" has received since the last time it fully synchronized with "B", and (2) having "A" make a *Bloom filter* over those records. A Bloom filter is a data structure that efficiently solves the approximate set membership problem with the convenient guarantee of zero false negatives [5]. "B" can then cross-check the records "B" has received since its last sync with "A" against the Bloom filter from "A". Any record that "B" has that the Bloom filter returns false for is a record that "A" is guaranteed to be missing. Note that, with small probability, the Bloom filter may return a false positive, leading "B" to incorrectly believe that "A" is not missing the record. This implies that our algorithm does not guarantee full pairwise synchronization, but instead takes a large step towards convergence with high efficiency.

In practice, we run a multi-stage anti-entropy protocol that combines the best properties of multiple ideas and algorithms discussed thus far. The full protocol is summarized as follows:

- Have each replica keep track of the records it has received since its last pairwise synchronization with every other replica. This can be done efficiently e.g. by associating every record present in this replica with the local physical timestamp of when this replica received that record, and doing range queries keyed on the timestamp.
- Have replicas periodically pair up to perform pairwise synchronization. Perform the following in order:

1. Perform a "quick check" by having each replica hash over their respective "new records since last sync" and exchange this hash. If the hashes are the same, the replicas are already consistent (i.e. there have been no failures since the last sync), and we are done.
2. Perform our Bloom-based algorithm to take a large but efficient step towards convergence.
3. Perform Lu's algorithm to patch remaining inconsistencies and reach full pairwise synchronization.

4 ALTERNATIVE SUBRECORD-BASED DESIGN

4.1 Model

In all previous work, records are the main unit of information for both reading, writing, and signing. This makes sense when records are written one-at-a-time, but as discussed in the background, can be suboptimal.

We propose a novel design that includes a mechanism for sub-records within one record. With this model, records become the main unit of information for signing and linking, while sub-records become the unit of information for reading, writing, and proving integrity. This is particularly efficient when large numbers of writes can be performed at once. For example, in a key-value store, a batched update may consist of updates to many keys, which should be stored in individual sub-records (to allow them to be read individually). Storing the entire batch as one record, with the updates as sub-records, allows the entire batch to be signed with one signature.

Internally, each record is stored as a Merkle tree, with the leaves being the subrecords, and the root hash becoming the name of the record. The Merkle tree may have additional leaves to link the record to other records; this allows the overall structure of the DataCapsule to become a DAG.

Writer API

- *Create*(Metadata): creates a new DataCapsule.
- *Init*(DataCapsuleName): initializes a session/batch of requests that refer to the given DataCapsule name.
- *Write*(Subrecord): persists the given subrecord to the DataCapsule referred to by the containing context/session/batch. The "Subrecord" input here only includes encrypted data, since subrecords do not have individual hash pointers.
- *Commit*(Backpointers): Creates a record out of the latest written subrecords. The client signs the root of the merkle tree for use in integrity proofs; the server sends back a signed acknowledgement that the record has been persisted.

Note that signatures are only exchanged on each "commit" operation, so the total amount of overhead in the system depends on the average number of subrecords per record. If each record only contains one sub-record, then this system matches prior DataCapsule designs where each record is signed.

Reader API

- *Init*(DataCapsuleName): initializes a session/batch of requests that refer to the given DataCapsule name.
- *Read*(SubrecordName) -> Subrecord: reads the subrecord with the given name.
- *Prove*(SubrecordName) -> Proof: generates a proof of integrity for the subrecord. A proof can come in multiple forms, including (1) a hash chain of leading up to the merkle root, or (2) a hash chain ending at a previously proved node in the merkle tree.
- *QueryFresh*() -> Set<RecordName>: returns the set of heads in the record history DAG. The reader can collect the responses of a high quorum of servers in order to be as up-to-date as possible.
- *Subrecords*() -> Set<SubrecordName>: returns the set of subrecords within a given record.

This is very similar to the API presented in section 3.2; the main difference is that operations which previously applied to records now apply to sub-records.

4.2 Integrity Proofs

Integrity proofs work very similarly to the mechanism described in 3.3. However, since each record stores a Merkle tree with a known depth, heuristics are not needed to find the shortest path. Instead, the server may simply walk up the edges of the Merkle tree until either the root is reached, or a node in the mirrored cache is reached. This is overall more efficient for the server to compute, especially when there are few signatures.

Sequential proofs of subrecords within a record are very efficient, since nodes that are next to each other in the Merkle tree have many nodes in common, and therefore have a high probability of a node being cached. This may be achieved in the other design as well, if the client decides to put the records in a Merkle tree structure, but this design has a lower overhead because the Merkle tree structure is assumed.

5 IMPLEMENTATION

5.1 Overview

We implemented both of the designs described in section 3 (which we will refer to as the "Flexible API" design) and section 4 (which we will refer to as the "Subrecord API" design). Although the designs have different APIs and algorithms, the engineering for both of them is very similar and they share most of their components. The implementations and tests total to around 4900 lines of Rust code, excluding the TCP alternative implementations (discussed in 5.3).

5.2 Peer-to-peer network

DataCapsules were originally designed with the Global Data Plane, a federated network, in mind. However, the Global Data Plane is not ready for use at the time of writing. To simulate the Global Data Plane, we implemented a simple peer-to-peer network in Rust, that internally uses TCP sockets for communication. The performance characteristics of our peer-to-peer network may not match that of the real Global Data Plane, so we attempted to control for network latencies as much as possible in our benchmarks.

5.3 TCP Alternative network

While integration the Global Data Plane is the "end goal" for DataCapsule-related research, most elements of the design should function independently of what is used to transport information between clients and servers (for example, DataCapsule records could be sent as handwritten letters delivered via the US Postal service). One particularly useful networking model is that of a TCP server. Deploying TCP servers is particularly simple in modern society; in addition, the TCP model provides some particularly useful features that make implementation and scaling easier.

- Message ordering
- Ability to detect when clients connect and disconnect
- Separate socket per client

- Abundance of libraries for implementing applications on top of TCP

So in addition to the peer-to-peer networked versions of our clients and servers, we also implemented TCP server versions of both designs. These implementations can be deployed without a peer-to-peer network, making them more easily usable.

However, server-to-server gossiping is much harder in the TCP model (each server would have to know all the other servers hosting each datacapsule). We did not have time to implement a useful server-to-server communication model on top of TCP, so the TCP implementations are missing replication and anti-entropy features.

5.4 Wire Format

Each peer-to-peer message contains a sender, destination, some content, and some optional metadata. In our implementation, the content of each message represents one or more operations, as described by the server API. The operations are serialized in the Postcard Wire Format [1], chosen for its efficiency. However, the wire format is not fundamental to our design, and may be switched to an alternative (such as Protocal Buffers) in the future if needed.

5.5 Storage

We used sled, an experimental high-performance key-value store [2] written in Rust. Sled has the ability to create separate namespaces, called "trees". We store the data for each datacapsule in a separate tree. Within each datacapsule, we also have separate trees for different types of storage.

For the base design, it has separate trees for:

- Metadata: the datacapsule's metadata is stored separately from its records.
- Record Bodies: the hash of the record is used as the key, and the encrypted content of the record is the corresponding value.
- Record Headers: the hash of the record is used as the key, and the corresponding value is the record header (consisting of the hash of the body, and the hash of previous records that this record points back to).

The experimental design is similar, but has a tree for Merkle tree blocks instead of record headers.

5.6 Server Networking

The server makes heavy use of green threads, for scalability reasons. It starts a green thread for every client it sees, keeping a small amount of per-client state to allow it to efficiently answer to queries.

5.7 Client Library

We implemented a small client library for clients to use. It handles cryptography and networking details, but is otherwise very minimal. It exposes APIs for:

- Creating requests (one function for each request in the server API)
- Sending batches of requests
- Waiting for batches of requests to be received

The client library verifies cryptographic hashes and signatures, and encrypts/decrypts data on write/read operations.

6 EVALUATION

6.1 Hardware

All benchmarks were ran on a laptop, specifically a Lenovo Thinkbook 14 Gen 4. The laptop has an 8-core CPU (AMD Ryzen 7 5825U) and a 512 GB SSD with a PCIe gen 4 interface.

All tests were run locally, with both the client and the server implementations running on the laptop. The client and the server connected to each other via a simulated peer-to-peer network that internally used TCP sockets. This simulated network has a much lower latency than any real-world network. It is unknown whether the simulated network has lower or higher computational overhead than a real-world network.

6.2 Benchmarking Goals

The main concern of these benchmarks is the throughput of the server. Latency is a concern, but processing time contributes very little latency compared to the total latency of each operation. For reference, real-world network latency is usually in the 10-100ms range, but our server had an internal latency of a few microseconds per operation (not counting networking cost). Therefore, it is more fruitful to focus on the server's total throughput as a metric of its performance.

To minimize networking costs as much as possible, we usually combine many operations into a single peer-to-peer message. This may be realistic for some use cases (For example, if a client is trying to read an entire datacapsule's history all at once), but unrealistic for others. Regardless, our main goal is to test the throughput of our algorithms, not the throughput of our peer-to-peer network, so we view this optimization as a necessity.

Finally, many of our benchmarks use 16 byte records, which is unrealistically small. However, this aligns with the goal of testing the overhead of our algorithms. Using 16 byte records make any differences in overhead more apparent, since per-record overhead takes up the majority of the computation time. In contrast, if we ran our benchmarks with kilobyte-sized or megabyte-sized records, the majority of the time taken would be in less interesting parts of the system, such as networking.

6.3 Sequential Benchmarks

To evaluate the maximum throughput of our integrity proofs, we create sequential benchmarks that wrote 100,000 records and asked for proofs sequentially. The cache performs very well, butting down proof times by an order of magnitude. This is the best case for the cache, but the cache still performs well in the later YCSB benchmarks.

6.4 YCSB

Methodology. In order to test real-world performance, we use realistic workloads generated by the Yahoo Cloud Serving Benchmark. YCSB is a benchmark suite that has been used to evaluate several other Datacapsule-related projects.

We use traces generated from YCSB, which are a mixture of reads and writes. We then simulated a simple key-value store on top of our DataCapsule client, that works as following:

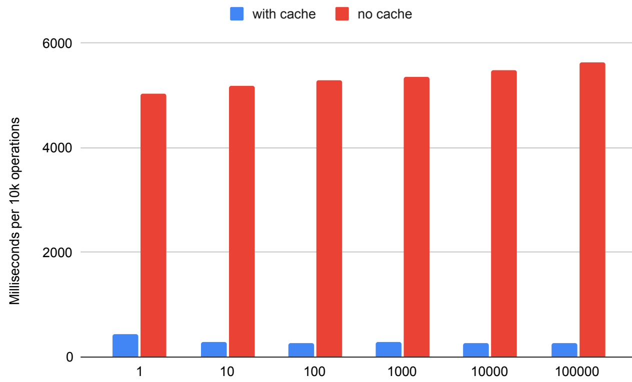

Figure 5: Times with and without proof caching. Each bar is the time it took to do 10k operations, in milliseconds. The x-axis is how many records there are per signature.

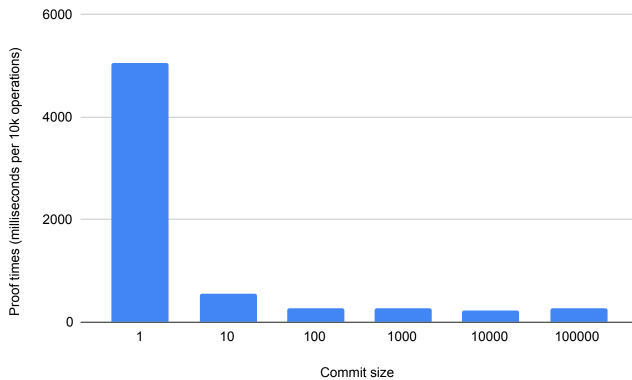

Figure 6: Proof times vs commit size. Each bar is the time it took to do 10k operations, in milliseconds. The x-axis is how many records there are per signature.

- The key-value store stores a hashmap and a hashset in-memory. The hashmap is maps keys to record hashes. The hashset store the hashes of any records that have not been signed.
- For each write, it performs a datacapsule write (with a random 16-byte value), then stores the hash of the new record in the hashmap. It also stores the hash in the hashset.
- For each read, it looks up which record hash it corresponds to. If the hash is in the hashset, it is unsigned. In that case, the client signs the latest hash (which allows the server to prove all the currently unsigned hashes), then clears the hashset. The client then reads the record hash, and requests a proof for the hash.

This algorithm allows the key-value store to avoid making signatures as long as possible, allowing it to have much higher throughput than if it needed to sign every write. Signatures are still needed, however, to allow the DataCapsule server to prove the integrity of each record.

YCSB Workloads. We use four different workloads generated by YCSB: Workload A is a random write-only benchmark. Workloads

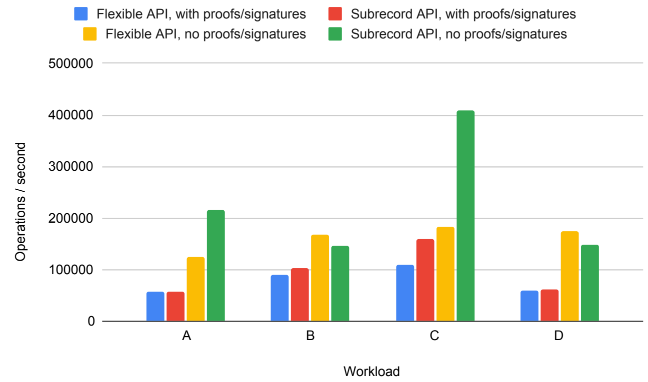

Figure 7: YCSB Benchmarks on both implementations.

B is a random read-heavy benchmarks, with approximately 95% reads and 5% writes. Workload C is a random read-only benchmark. Workload D is a "read latest" benchmark, where reads are more likely to correspond to the latest writes.

Main Results. Our implementations perform well in the YCSB Benchmarks, with almost as high throughput as in the sequential benchmarks. The blue and red bars in figure 7 represent overall the performance of the Flexible API implementation and the Subrecord API implementation respectively. They perform very similarly, with the Subrecord API performing better in workload C due to being more efficient for integrity proofs.

The Effect of Cryptography. Much of the overhead of our system comes from signatures and integrity proofs. To quantify how much, we turned off the signatures and integrity proofs, which are the yellow and green bars in figure 7 respectively. This increases the throughput by more than double, indicating that cryptography (and related algorithms) is a majority of our total overhead.

Comparison against RocksDB To see if our implementation stands up to the state-of-the-art, we also compared its performance to RocksDb. For a fair comparison to RocksDb, an embedded database, we ran our system locally with one client. We batched both systems' operations. The comparison against our implementations against the orange bars representing RocksDb is shown in Figure 8. RocksDb's batching generally performed poorly on mixed workloads. Our implementation performed better on workloads B and D, with only 5% writing, than workload A with 50% writing. RocksDb also performed better on these workloads by a much greater factor, suggesting that RocksDb cannot batch as effectively when the workload contains significant writing and reading.

RocksDb is heavily read-optimized, as evidenced by its stellar performance in workload C.

6.5 Concurrent clients

To test the scalability of our DataCapsule servers, we tested the total throughput of both implementations with different numbers of clients connected. Each test consists of 1 million writes, 1 million reads, and 1 million integrity proofs spread out among N different clients. Figure 9 shows the results for the flexible API: it has a higher throughput when multiple clients are connected, although its total

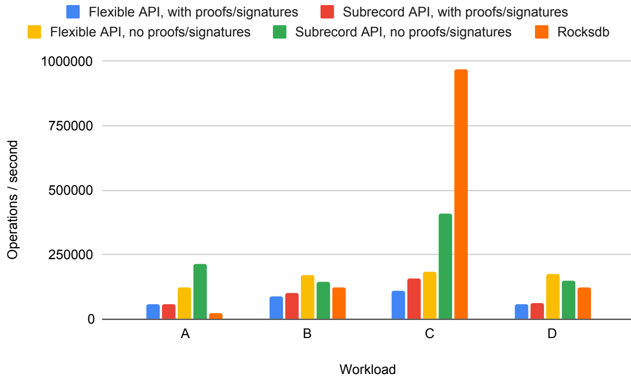

Figure 8: YCSB Benchmark comparison against RocksDb.

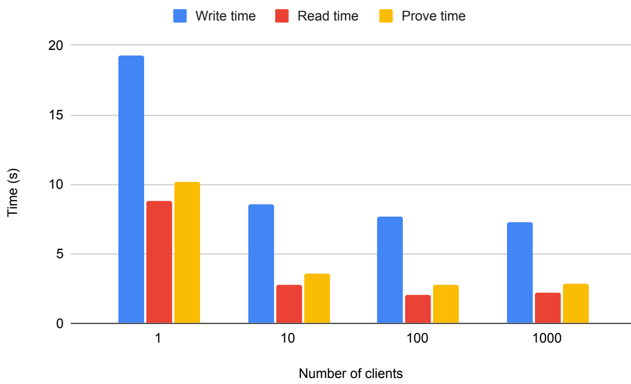

Figure 9: Concurrent benchmarks for Flexible API

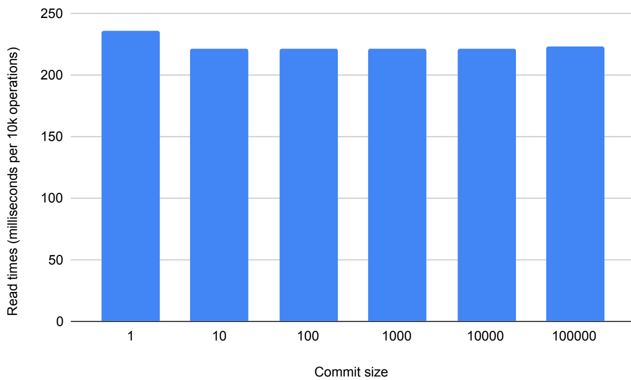

Figure 10: Read performance vs commit size

throughput is limited by the underlying hardware. Importantly, its throughput does not degrade when large numbers of clients are connected. Figure 14 shows the results for the subrecord API, which are very similar.

7 CONCLUSION AND FUTURE WORK

DCS represents a step towards a production-level DataCapsule storage plane that DataCapsule-based systems and applications can be built on top of. We show that the security guarantees of

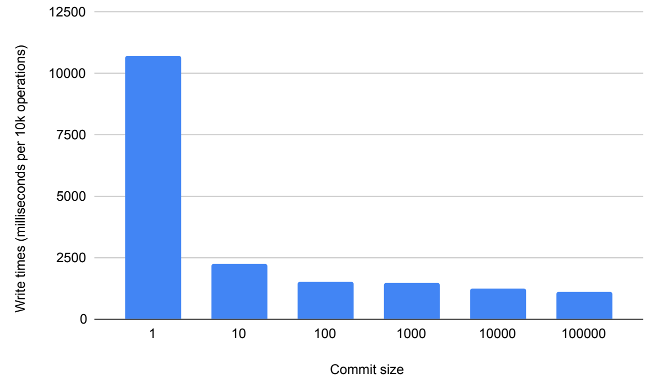

Figure 11: Write performance vs commit size

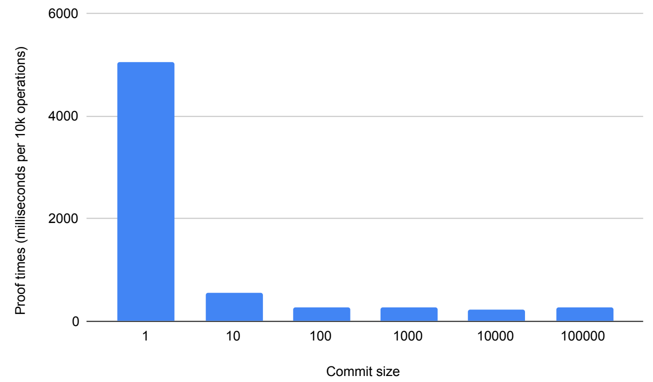

Figure 12: Sequential proof performance vs commit size

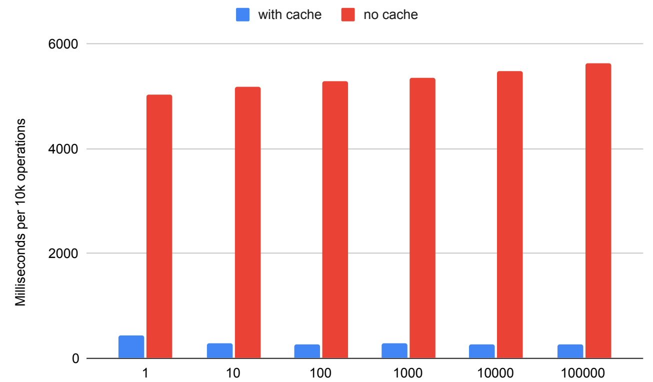

Figure 13: Sequential proof time with and without a cache.

DataCapsules can be provided with acceptable overhead for high-stress workloads.

While our implementations are functional, they are far from ready for real-world heavy usage. In particular, implementing support for replication across servers and anti-entropy among replicas is a work-in-progress. As mentioned in Section 3.3, we are also looking to improve our cache mirroring technique in terms of both performance characterization and correctness guarantees.

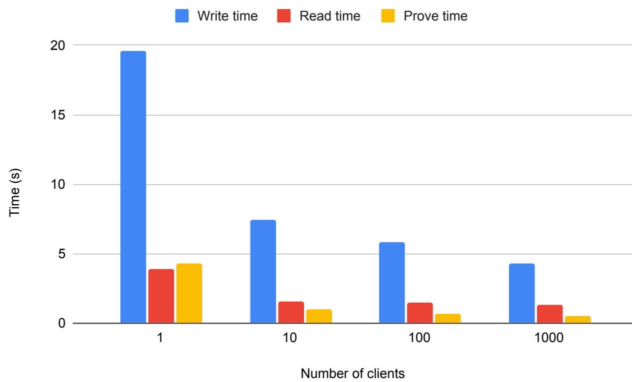

Figure 14: Concurrent benchmarks for Subrecord API

Furthermore, we note that our evaluations lack workloads from a diverse set of real-world applications and systems. This is in part due to the current ecosystem around DataCapsules being relatively small and scattered. We hope that this ecosystem and DCS can be co-designed and co-developed into the future.

REFERENCES

[1] The Postcard Wire Specification. (????). https://postcard.jamesmunns.com/

[2] Sled: the champagne of beta embedded databases. (????). https://github.com/spacejam/sled

[3] Peter Alvaro, Neil Conway, Joe Hellerstein, and William Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. *CIDR 2011 - 5th Biennial Conference on Innovative Data Systems Research, Conference Proceedings*, 249–260.

[4] Amazon. 2023. Amazon S3: Object storage built to retrieve any amount of data from anywhere. (2023). https://aws.amazon.com/s3/

[5] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (jul 1970), 422–426. https://doi.org/10.1145/362686.362692

[6] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing (MCC '12)*. Association for Computing Machinery, New York, NY, USA, 13–16. https://doi.org/10.1145/2342509.2342513

[7] Kaiyuan Chen, Alexander Thomas, Hanming Lu, William Mullen, Jeffery Ichnowski, Rahul Arya, Nivedha Krishnakumar, Ryan Teoh, Willis Wang, Anthony Joseph, and John Kubiatowicz. 2022. SCL: A Secure Concurrency Layer For Paranoid Stateful Lambdas. (2022). arXiv:cs.CR/2210.11703

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[9] Alan J. Demers, Daniel H. Greene, Carl H. Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. 1988. Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Oper. Syst. Rev.* 22 (1988), 8–32. https://api.semanticscholar.org/CorpusID:1889203

[10] Git. 2023. Fast-Forward Merge in Git. (2023). https://git-scm.com/docs/git-merge#_fast_forward_merge

[11] David Kempe, Alin Dobra, and Johannes Gehrke. 2003. Gossip-based computation of aggregate information. *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* (2003), 482–491. https://api.semanticscholar.org/CorpusID:5689705

[12] Hanming Lu. 2022. *DCR: DataCapsule Replication System.* Master's thesis. University of California, Berkeley. Advisor(s) Kubiatowicz, John. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-267.pdf

[13] Friedemann Mattern. 1988. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, M. Cosnard (Ed.). Elsevier, Chateau de Bonas, France.

[14] Ralph Merkle. 1989. A Certified Digital Signature, Vol. 435. 218–238. https://doi.org/10.1007/0-387-34805-0_21

[15] Microsoft. 2023. Azure Blob Storage: Massively scalable and secure object storage for cloud-native workloads, archives, data lakes, high-performance computing, and machine learning. (2023). https://azure.microsoft.com/en-us/products/storage/blobs

[16] Nitesh Mor. 2020. *Global Data Plane: A Widely Distributed Storage and Communication Infrastructure.* Ph.D. Dissertation. University of California, Berkeley. Advisor(s) Kubiatowicz, John. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-10.pdf

[17] Nitesh Mor, Richard Pratt, Eric Allman, Kenneth Lutz, and John Kubiatowicz. 2019. Global Data Plane: A Federated Vision for Secure Data in Edge Computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1652–1663. https://doi.org/10.1109/ICDCS.2019.00164

[18] William Mullen. 2022. CapsuleDB: A Secure Key-Value Store for the Global Data Plane. (2022), 33 p. http://digicoll.lib.berkeley.edu/record/269485

[19] Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. 1997. Flexible update propagation for weakly consistent replication. *Proceedings of the sixteenth ACM symposium on Operating systems principles* (1997). https://api.semanticscholar.org/CorpusID:6455497

[20] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[21] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

[22] Ion Stoica and Scott Shenker. 2021. From Cloud Computing to Sky Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 26–32. https://doi.org/10.1145/3458336.3465301