Replication for the Global Data Plane

Yucheng Yang
3031947231
tonyyang@berkeley.edu

Scott Numamoto
26342027
scott.numamoto@berkeley.edu

Steven Wu
26198048
stevenwuyinze@berkeley.edu

Abstract

There has been an explosion in the number of smart devices connected via the Internet to augment people's everyday life. The Global Data Plane (GDP) is a new infrastructure design which tries to address the challenges presented by such a vast and broad array of devices by introducing a higher layer of abstraction. In this paper, we present an anti-entropy mechanism which ensures that data is replicated on multiple GDP log servers, and describe the corresponding APIs for users to access data without knowledge of the underlying details of replication. We choose to support a highly write available system. Results show that our replication system can recover from large loss of log servers and has significantly better usage of bandwidth than a naive system.

1 Introduction

There has been an explosion in the number of smart devices connected via the Internet to augment everyday things. This shift is commonly referred to as the Internet of Things (IoT) [2]. The growth in IoT poses scalability and security challenges on its existing infrastructure based on cloud service. The Global Data Plane (GDP) is a new infrastructure design which tries to address these challenges by introducing a higher layer of abstraction [10].

In this paper, we present a anti-entropy mechanism which ensures that data is replicated on multiple servers in GDP, and describe the corresponding APIs for users to access data without knowledge of the underlying details of replication. GDP uses a single-writer append-only log model to organize data [10], and therefore requires different anti-entropy mechanisms and APIs than those of typical distributed databases.

Typical modern distributed databases use anti-entropy mechanisms that are either based on Merkle trees [9] or version management [11]. We discovered that neither is best for GDP.

Version management requires that updates on data are applied in a consistent order on every server. However, most IoT devices running on the GDP network, say security cameras and baby monitors, require high write availability and version management cannot satisfy such requirements.

Merkle tree based approaches impose the requirement that logs on each server can be ordered consistently across all servers. Nevertheless, in GDP, the possible identifiers for such logical ordering - one example is the timestamp of log generation - is provided by the writer; we cannot fully trust the writer to always behave correctly since faults on devices can happen. The leaf nodes of the Merckle tree are the ordered records. If one log record needs to be inserted in the middle of the linearized view, every log that is to the right of the insertion point needs to be shifted over. The re-position of logs after replication leads to changes in the Merkle tree structure and induces huge CPU workload on log server upon each record's arrival. Furthermore, the size of the Merckle tree digest will increase as the write continues, which is not preferable for GDP as an overlay for many write-frequent devices. By contrast, our anti-entropy algorithm is designed based on the unique data model of GDP, which we will further discuss in Section 2. It does not require log servers to keep a linear ordering of the data and sends out minimal information during exchanges of digest messages.

The remainder of this paper is structured as follows: In Section 2 we discuss the data model of GDP and how users can access data without detailed knowledge of the underlying systems . In Section 3 we present the design and implementation of our replication mechanism and how it interacts with the rest of the GDP infrastructure. In Section 4 we discuss the results from some evaluations of our system. In Section 5 we discuss related work. In Section 6 we talk about future work, and we conclude in Section 7.

2 Data Model and User Semantics

In this section we briefly summarize the GDP data model and describe choices we have taken to support a highly write available system.

2.1 The GDP Data Model

The Global Data Plane (GDP) is a widely distributed and federated storage and communication infrastructure. with the key interface of a single-writer append-only secure log [10]. A log consists of its metadata and an ordered list of records.

These GDP records are the unit for reading and writing to the log. A record includes header and body. The record body holds encrypted application level data and is opaque to the log server.

The record header contains meta-information about the log. Most critically for our replication service, the record header contains a hash of the record header of the previous record in the log. The record header can also contain a variable number of hashes to older records in the log and a monotonically increasing integer that refers to the order in which the records were written to the log. Other information in the record header includes the name of the log the record belongs to.

Log-servers are the nodes that provide the physical storage for logs. A log-server may host multiple logs.

The single writer of a log provides serialization for the log entries. The writer should maintain recent information about the log, including the headerHash of the most recent record and any records that have not yet reached satisfactory durability.

2.2 A Highly Write Available System

We choose to support a highly write available log. Ideally, a log can represented by an ordered list of records. Each record's prevHash is that of the record before it. Each record's hash serves as the prevHash of only the record succeeding it. This may not be the case in practice.

A writer should use the headerHash of the most recent record in the log to append a new record to the log. Writers should keep this headerHash stored in non-volatile memory. Log servers could enforce the linearity of the record graph by rejecting records that do not use the headerHash of the most recent record, but this would limit the speed at which the single writer could append logs and is further complicated by multiple replicas. We choose to support a greater write availability at the cost of a more complex topology for the graph of log records. However, this design choice also means we have to deal with branches and holes in the GDP log chain.

2.2.1 Branches

Consider a writer that cannot wait for a new headerHash to persist to disk before appending a new record. The headerHashes of new logs wait in volatile storage to be flushed to disk. The writer crashes before the flush is complete, and upon recovery loses some of the new records that have been sent to the log server.

Typically the writer could request the last n records from the log server to confirm the latest headerHash on disk is that of the log. Our writer cannot wait for a response from the server before writing. Thus, the writer continues appending to the log using a headerHash that is not the most recent headerHash of the log. The writer has created a branch and is illustrated by records A, B, C, and D in figure 1.

2.2.2 Holes

There are two types of holes that can exist in the graph of a log: permanent and transient.

A permanent hole is created in the following way. Consider writer with limited storage that is constantly appending to the log. A single record is lost due to network error. The writer continues appending to the log and soon removes the dropped record from the writer's limited storage. The log server continues to accept these records even though one of the records hash a prevHash to a record that is not stored in the log. A permanent hole is now in the log.

A transient hole is created in the following way. Consider a writer that is sending appended records to all log servers. Log server A experiences a network outage and does not receive the record. The writer receives confirmation of receipt of the record from all other log servers and does not resend the appended record. The writer continues appending new records to the server. Log server A regains network connection and receive the new records from the writer. Log Server A now contains a transient hole. The other log servers communicate with log server A to patch the hole. Some abnormalities in GDP's networking layer such as out-of-order packet arrival may also result in similar effect.

2.2.3 Read Semantics

The semantics of the read operation change slightly in the face of multiple replicas of a log. We choose to support the following operations for read and illustrate them in figures 1 and 2.

Readers can read a record specified by its GDP name through read(hash). Readers send this request to all log servers and accept the first response.

Readers can read the last record on a log with read(-1). Readers send this request to all log servers. Once R

Figure 1: Graph of Records in a Log

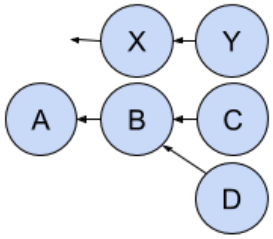


Figure 2: Example Read Calls

```
read(B) = record_B
read(-1) = [record_C, record_D, record_Y]
read(-2) = [
    [record_C, record_B],
    [record_D, record_B],
    [record_Y, record_X]
]
```

log servers respond, the reader reconciles the responses by discarding any records that are succeeded by a record from another log server's response. Log servers respond with the latest record on all branches and after any hole after a checkpoint. For the log in the figure 1, the log server would respond with records C, D, and Y.

Responses for the latest record could become cluttered with abandoned branches and holes if not tended to. Checkpoints are logs written by the writer that indicate all branches before this record may be ignored. We illustrate a checkpoint and examples function calls in figures 3 and 4. Checkpoints also indicate the canonical branch that should be used. Writers should repair any holes and branches necessary before appending a checkpoint record.

Readers can read the last n records on a log with read(-n). This functions similar to the reading the last record on a log, but returns the last record on a log and the $n-1$ logs that precede it.

2.2.4 Write Semantics

The single writer can write to the log with append record. The single writer sends this request to all log servers.

To guarantee the durability of a record, the single writer holds a record after sending it to the log server in storage until it receives acknowledgement from a number of log servers that is satisfactory to the single writer and less than or equal to the number of log servers.

We do not force the single writer to guarantee durability of records by holding them in storage. Nor do we force the single writer to wait until it receives a desired number of acknowledgements to continue appending records.

Figure 3: Graph of Records in a log with Checkpoint and example API calls

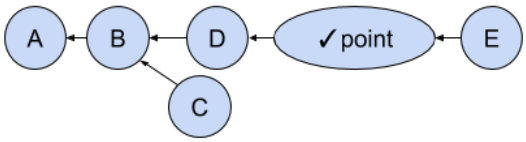


Figure 4: Example Read Calls with Checkpoint

```
read(-1) = [record_E]
read(-2) = [[record_D, record_E]]
```

Consider a situation in which a writer appends record C and receives acknowledgement of success from log server 1. The writer does not have the storage to hold record C. After sending the record to the log servers the writer discards record C.

Log server 1 then goes down for maintenance for several hours before the next heartbeat and opportunity to replicate record C. The writer does not receive acknowledgement of success from any of the other log servers. From the perspective of the writer, record C is unavailable and may be permanently lost. Our semantics support two main options for the writer.

Option 1: The writer may choose to continue appending logs that follow record C. All available records will contain a hole, but this hole will be patched upon the recovery of log server 1 in several hours.

Option 2: The writer may choose to discard the record C and continue appending with the hash of record B. Upon log server 1's recovery, record C will replicate to all log servers as a branch. The writer, upon realizing record C did fully replicate, may later choose to close off this branch by setting a checkpoint on the writer's chosen branch. A topology that could result from this option is shown in Figure 3.

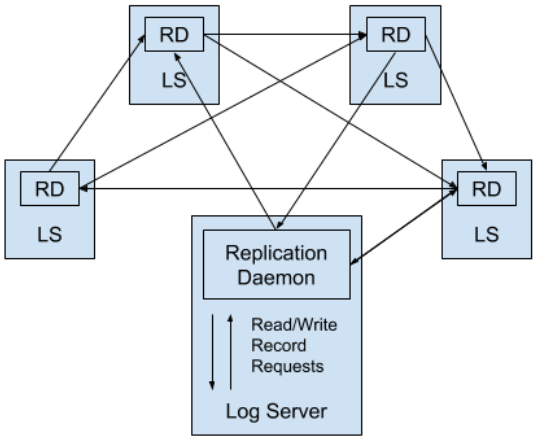
3 Design and Implementation

3.1 Architecture

Replication for each log is orchestrated through replication daemons that runs on the log's log servers.

Log daemons interact with the records of the log on their log server by reading and writing the records of the log on their log server.

Figure 5: Replication Daemons sending out heartbeat with fanout degree (f) = 2



Log daemons for one log communicate with each other through heartbeats to replicate logs to all log servers. Each log daemon periodically sends a heartbeat to f peers. In figure 5, one replication daemon is initiating a heartbeat with $f = 2$ peers.

A replication daemon may manage replication for multiple logs each with a unique set of peers. A log server might have one single replication daemon for the replication of all the logs it stores.

During a heartbeat two daemons exchange messages until both daemons have the same records, regardless of the presence of branches and holes. Our replication service can repair a log server that momentarily went down and is missing the last several records as well as a log server whose hard drive failed and lost all records.

Records exchanged by replication daemons are verified through the same methods used when appended logs are sent to the log server from the writer.

A peer may fail during a heartbeat, possibly by crashing and sending no message or sending a corrupted message. The corresponding peer will abort the rest of the heartbeat, and restart the heartbeat process the next time that peer is selected for a heartbeat.

3.2 Multicast

We previously described read and append functions for the user to interact with the log that sent requests to all log servers. Using traditional internet protocol, these requests could become burdensome when scaling to a large number of replication servers. The GDP routing layer, however, provides multicast that allows the client to send a single request that it delivered to all log servers via the GDP routing layer.

3.3 Anti-entropy mechanism

Since GDP's data model does not require strong consistency, anti-entropy protocol, also known as gossip protocol, is an attractive way to replicate data [5] across different GDP servers and address inconsistencies. A good anti-entropy protocol should spread updates across nodes quickly and use a minimal amount of network bandwidth.

Given the unique data model of GDP which involves holes and branches, we designed a novel, graph-based algorithm to generate messages used in the anti-entropy protocol, as discussed below in section 3.3.1. Our anti-entropy protocol allows any pair of servers in GDP to fully synchronize their state and address inconsistencies within only four messages exchanged. This ensures steady latency for spreading updates given a stable network connection between the two.

To evaluate our algorithm we also designed and implemented a naive approach that we call the simple set comparison algorithm.

3.3.1 Graph Comparison Algorithm

In an anti-entropy protocol, messages are generated to represent to a server some part of the state of one of its peers. Given that the data model of GDP can be seen as a directed graph, our algorithm uses the sources and sinks of the graph to minimize the information needed to be exchanged. Since our anti-entropy protocol includes four messages been exchanged between a pair of servers in GDP, there are four parts in the algorithm which generates the messages. A graph digest is used in some parts of the algorithm, which will be introduced below before we present the specification of the algorithm. Finally, we will give a proof of correctness and discuss the performance of this algorithm under different situations.

3.3.2 Graph Comparison Algorithm: Graph representation of the GDP data model, and digest of a graph

Since each record in GDP has a unique hash address and contains a hash pointer to a previous record, all the records in the same GDP hash namespace can be seen as nodes on a directed graph. Every directed edge from node X to Y in the graph describes that there are two records, X and Y, and the previous record of Y is X. If a record points to a previous log that does not exist locally, there is no directed edge corresponding to that. If a record specifies that is the root record in this namespace, there is also no directed edge that points to it. Formally, the graph can be defined as the following.

Definition 1. Record and State

A record in the GDP namespace L contains two hash pointers $(L_{address}, L_{prev})$. $L_{address}$ is the hash address of record L, and L_{prev} is the hash address of record L' which should be the previous record for L. When $L_{prev} = \text{null}$, L declares that it is a root record in its namespace. The state of all records in a GDP namespace on a server can be described as a set of records $S = L$.

Definition 2. Graph representation of a state

The graph representation $G = (V, E)$ of a state S can be defined as: there is a bijection between V (set of nodes) and S (set of records), where, for the sake of convenience, we will use a record L to refer to the node in V corresponding to L; for every record $L \in S$, there is a unique corresponding $e \in E$ if $L_{prev} \in S$, and such e starts from L_{prev} and ends at L.

Definition 3. Records ahead of and after a record

Given a record $L \in S$. The directed graph of S is directed graph G. The corresponding records of all nodes in G from which there is a path to L are called records ahead of L. Likewise, the corresponding records of all nodes in G to which there is a path from L are called records ahead of L. In neither case, L cannot be said to be a node ahead of or after itself.

Definition 4. Connected component of a record

Connected component of a record L is defined as the union of L, records ahead of L, and records after L.

A digest of the directed graph generated as above contains two lists: one list for all the sources and another list for all the sinks in the graph.

3.3.3 Graph Comparison Algorithm: Message generation

There are four messages sent between two servers in the state reconciliation mechanism as two round trip exchanges. The algorithm assumes that no external writers will change the state of the servers when this process is happening. We display Algorithms 1 through 4 on the current and next page.

Suppose the message exchange between servers A and B is initiated by A. A sends message 1 to B, and B replies with message 2. Then A sends message 3 to B, and B replies with message 4, which completes the state reconciliation.

Contents of message 1 contains the digest of the graph generated based on the state of A. Contents of the message 2 contains the digest of the graph generated based on the state of B, and a list of records generated according to Algorithm 1.

After A receives message 2 from B, it first adds all the records in message 2 to its local state, and refresh its digest. A then continues to generate message 3, which contains a list of records and a list of hash addresses. Both lists are obtained according to Algorithm 2.

B first adds all records in message 3 to its local state. Then B generates message 4, which includes a list of records, generated according to Algorithm 3.

Algorithm 1 Generation of Message 2

```
1:  procedure GETMISSINGDATAA
2:      nodeB ← all nodes of B's graph
3:      sourceA ← source nodes of A's graph
4:      sinkA ← sink nodes of A's graph
5:      sourceB ← source nodes of B's graph
6:      sinkB ← sink nodes of B's graph
7:      L ← []
8:      for v ∈ sourceA do
9:          if v ∈ nodeB and v ∉ sourceB then
10:             Add all connected nodes ahead of v in
    nodeB to L, until we reach a node in sinkA
11:     for v ∈ sinkA do
12:         if v ∈ nodeB and v ∉ sinkB then
13:             Add all connected nodes after v in nodeB
    to L, until we reach a node in soourceA
14:     return L
```

3.3.4 A Naive Algorithm

To compare and measure the efficacy of our algorithm, we also designed and implemented a naive anti-entropy protocol called Simple Set Comparison. Each log server that serve a log is viewed as a set of records. During each heartbeat, the two replication daemons compare their full record sets and exchange the records exclusive to only one peer. The replication daemons exchange three messages, as illustrated in figure 6.

To compare the two algorithms, simple set comparison will send all hashes from one peer to the other. Our graph comparison algorithm sends only a subset of the hashes according to the missing sinks and sources of the graph. Both algorithms then exchange information about the records exclusive to one peer. As the number of records on the log server grow, the amount of data communicated per heartbeat will also grow linearly with simple set comparison. This is not necessarily the case with our graph comparison algorithm.

4 Evaluations

In this section, we evaluate replication for GDP by simulation. The simulation utilizes the architecture described in section 3.1 and either the graph comparison algorithm or the simple set comparison algorithm for the communication protocol. We measure our success on the improvements of the graph comparison algorithm over the simple

Algorithm 2 Generation of Message 3

procedure GENERATEMSG3
    nodeA ← all nodes of A's graph
    sourceA ← source nodes of A's graph
    sinkA ← sink nodes of A's graph
    sourceB ← source nodes of B's graph
    sinkB ← sink nodes of B's graph
    $L \leftarrow []$ // records
    $H \leftarrow []$ // hash addresses
    for $v \in$ *sourceB* **do**
        if $v \in$ *nodeA* **and** $v \notin$ *sourceA* **then**
            Add all connected nodes ahead of v in *nodeA* to L, until we reach a node in *sinkB*
    for $v \in$ *sinkB* **do**
        if $v \in$ *nodeA* **and** $v \notin$ *sinkA* **then**
            Add all connected nodes after v in *nodeA* to L, until we reach a node in *soourceB*
    for $v \in$ *sourceA* **do**
        if $v \notin$ *sourceB* **and** $v \notin L$ **then**
            Add connected component of v to H
    for $v \in$ *sinkA* **do**
        if $v \notin$ *sinkB* **and** $v \notin L$ **then**
            Add connected component of v to H
    return L, H

Algorithm 3 Generation of Message 4

procedure FULFILLDATAREQ
    nodeB ← all nodes of B's graph
    req ← list of hash address in message 3
    $L \leftarrow []$
    for $v \in$ *req* **do**
        Add the connected component of v in *nodeB* to L
    return L

Figure 6: Simple Set Comparison Messages

```
  msg_1(peer_a_hashes)->
<-msg_2(
    peer_b_only_records,
    peer_a_only_hashes)
  msg_3(peer_a_only_records)->
```

Algorithm 4 Simple Set Comparison Pseudocode

```
peer_a.send(peer_a.hashes)

peer_a_only_hashes,
peer_b_only_hashes =
    peer_b.compare(
        peer_b.hashes,
        peer_a.hashes)
peer_b_only_records =
    peer_b.read(peer_b_only_hashes)

peer_b.send(
    peer_b_only_records,
    peer_a_only_hashes)

peer_a.write(peer_b_only_records)
peer_a_only_records =
    peer_a.read(peer_a_only_hashes)

peer_a.send(peer_a_only_records)

peer_b.write(peer_a_only_records)
```

set comparison algorithm in the metrics of scalability of recovery from churn and reduced bandwidth usage.

4.1 Model Scenario

We model our simulation on the following scenario. The single writer is a security camera that is monitoring the hidden entrance to Professor JK's office. This security camera has extremely limited storage capabilities and deletes records shortly after sending them to the log servers. The security camera and the log servers are connected to the Internet via a WiFi router with a typical speed of 10 Mbps and delay of 20ms [3]. The network drops a packet from the writer every 1 out of 100 records.

The security camera is recording video with a resolution of 320x240 at 15 frames per second with H.265 compression at all time. The security camera appends its recorded video to the log every 0.1 seconds. With these parameters, each record has a body of size 3 KB with 0.1 seconds of video [1].

The security camera is writing to n log servers. The log servers exchange heartbeats with each other every 0.5 seconds. The security camera writes to a limited number of log servers based on network constraints.

4.2 Simulation Environment

We implemented our replication service in Golang 1.11 in approximately 2,700 lines of code. We analyzed the results of our simulation using Python and Jupyter notebooks. Our implementation can be found at `https://github.com/tonyyanga/gdp-replicate`.

We ran our simulation on Google Cloud Platform with a n1-highcpu-4 with 4 virtual CPUs and 3.6 GB of memory. We simulated a network connection through Mininet an application that simulates a realistic network. We simulated each log server as a SQLite database and an accompanying replication daemon.

Replication daemons communicated with each other over a simulated switch network with packet drop rate of 0.01%. As the current implementation of the GDP routing layer does not permit communication to specific log servers, daemons communicated with each other through HTTP.

We emulated the behavior of the single writer by inserting records into W of the n SQLite databases every 0.1 seconds, where $1 \leq W \leq n$. These databases were selected randomly for each write. This creates missing records in some $n - W$ log servers that are distributed through replication. In each trial, log servers each began with zero records.

4.3 Scalability

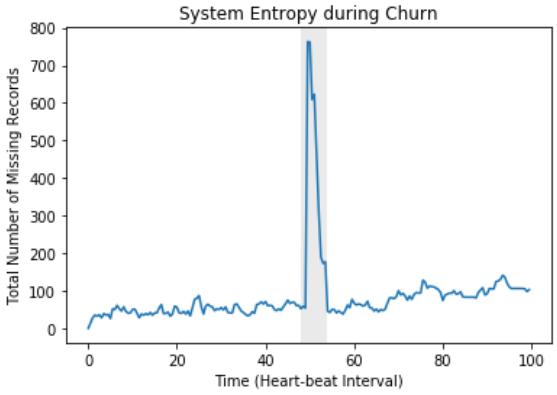
Figure 7: Recovery time from Multiple Node Failures

Num Log Servers	Simple Set (Num heartbeats	Graph to recovery)
3	1	1
5	2	1
15	8	5

We performed an experiment to measure our replication service's resiliency to large scale node loss. We continually wrote to a varying number of log servers. After appending 250 records, we churned the nodes by wiping all records from $\frac{n}{3}$ log servers and determined the amount of time necessary for the system to return normal.

We defined system recovery to normal as returning to a total number of records missing from all log servers consistent with before the churn. We display this metric for our trial with the graph comparison algorithm and $n = 15$ log servers in figure 8. The height of the large spike in the number of missing records make it easy to identify the amount of time.

Figure 8: Total number of missing records over log server churn



We performed the experiment with 3, 5, and 15 log servers. For each trial we used a fanout degree of $\log_2 n$. The single writer wrote to $W = \frac{n}{3}$ of the log servers for each record. During the experiment we appended 500 records to the log over the course of 100 heartbeats.

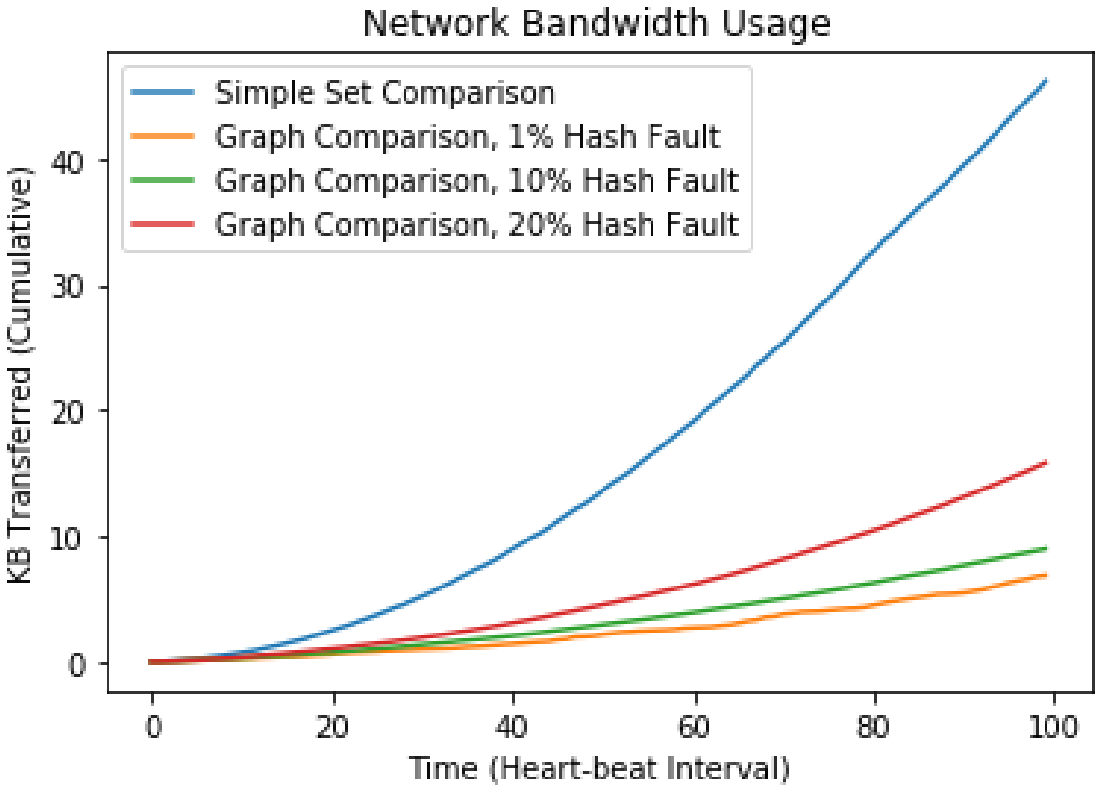
The graph comparison algorithm performs better than simple set comparison algorithm. Both algorithms are increasing quickly in recovery time with the increase in the number of log servers. With a small number of log servers, the graph comparison algorithm recovers immediately after a single heartbeat. The simple set comparison algorithm also recovers quickly, within 2 heartbeats. Some noise is expected in the time for recovery due to the random choice of peers. With $n = 15$ log servers, only 10 log servers held records immediately after the churn. We were unable to successfully simulate larger numbers of log servers.

4.4 Bandwidth

We performed an experiment to measure the total bandwidth usage of replication, shown in figure 9. We measured both the graph comparison algorithm and the simple set comparison algorithm for different rates of branch and hole creation. A 1% hash fault rate indicates that there is a 1% chance for a given written record to have the incorrect prevHash, creating a hole or branch.

We simulated the environment $n = 5$ log servers replicating the log, according to the parameters stated for the security camera scenario. The single writer wrote to $W = 3$ of the log servers for each record. We utilized a fanout degree of $f = 2$ for each heartbeat. During the experiment we append 500 records to the log over the

Figure 9: Cumulative Bandwidth Usage of Algorithms



course of 100 heartbeats.

We can make several observations about replication from the cumulative bandwidth usage in figure 9. The simple set comparison's usage of bandwidth that strongly aligns with polynomial curve. This is expected as the simple set comparison sends the entire list of all hashes of one log server at the start of each heartbeat. Each hash is roughly 1% the size of one record's payload - 3KB. This list consistently increases with the number of hashes in the log. We omit varying the hash fault rate for the simple set comparison algorithm, as the algorithm is not aware of the prevHash field.

The graph comparison algorithm utilizes much less bandwidth than the simple set comparison. At the end of their trials, the graph comparison algorithm used approximately nine times less bandwidth than the simple set comparison. Based on the graphs of each algorithm, this difference would continue to increase if the experiment was prolonged. Both algorithms have the same end result of transmitting all missing records between log servers, so the difference in bandwidth usage can be predominantly attributed to the difference in the number of hashes exchanged to determine which records are missing.

The amount of bandwidth used by the graph comparison algorithm increases with the rate of hash faults. This is consistent with the graph comparison algorithm as each heartbeat now includes more starts/ends in the graph. If possible, decreasing the number of branches or holes in a graph can improve the bandwidth usage of replication.

5 Related Works

Gossip protocols distribute information through members communicating with other randomly chosen members [15]. There are two classes of gossip: anti-entropy and rumor-mongering protocols. Anti-entropy peers gossip information until it reaches all members. To contrast, rumor-mongering peers gossip information for an amount of time such that it reaches all peers with some chosen probability [14]. We utilize the anti-entropy form of gossip.

Anti-entropy protocols utilize some merge operation between two states that create a new state. Periodically, each peer chooses another peer at random to gossip with and applies the merge operator on their states [14]. Sometimes, a peer will communicate with a subset of its peers instead of one. The size of this subset is usually fixed and called fanout (F) [6]. In protocols described in

[14], the fanout is 1 while in [7], it is shown that using a higher fanout can reduce the rounds needed for synchronization, although too high a fanout implies more network traffic and possibly performance loss due to excess message overhead. We fixed $F = log(N)$ in all of our trails, with N being the number of log servers.

There are three forms of gossip. In push-gossip, the receiving peer applies the results of the merge operation only to the receiving peers. In pull-gossip, the initiating peer applies the results of the merge operation to the initiating peer. Push-pull-gossip is the most efficient and applies the resulting state of the merge operation to both peers [14]. Both our graph comparison algorithm and simple set comparison algorithm utilize push-pull-gossip.

Anti-entropy provides some trade offs. The utilization of randomly choosing a partner prevents strong bounds on the convergence time of the members, but it does provide a low complexity to implement with simple data structures [5]. Anti-entropy theory and epidemic theory which it is a part of holds that a single infection site can spread to the entire population in time proportional to the size of the population [5].

Van Renesse et al. [14] evaluated an anti-entropy protocol that that adapts the amount of rate at which gossip is sent per peer through a TCP-inspired method.

While fanout is commonly held at 1 or another constant, Satish Verma et al. [16] examined dynamically changing fanout as the most important parameter for controlling latency of message delivery. The authors were able to allow the user to fine-tune fanout for their needs. Our trials experimented with multiple differetn fanouts, ranging from a degree of one to full broadcast. Our final trials utilized a fanout of $log_2 n$.

Van Renesse et al. [15] utilized a gossip protocol for the failure detection that was robust in the face of a message loss, host failures, and network partitions.

Gonçalves et al. [8] evaluated a distributed key-value store with inspiration from Dynamo that utilizes a logical clock framework and found that it utilized less memory and communication than an identical system utilizing Merkle Trees.

Dynamo is a key-value storage system with customizable semantics that can give the writer a high degree of write availability [4]. To establish such semantics, the database shifts some burden such as reconciliation from the time of write to the time of read. For example, multiple values may be presented to a user retrieving a key. At which point, the user may have have to select which version to keep. Our replication system proposes similar semantics taylored to GDP through the pruning of branches via checkpoints. As suggested in our semantics, the single writer is given the choice to stored appended logs until a desired level of durability is achieved from the confirmations of receipts from log servers. Dynamo similarly allows users to specify the desired degree for read and write quorums - if any.

Conflict-free Replicated Data Types (CRDTs) are data types that do not require complex synchronization to reconcile difference between replicas and converge to a common state [12]. A set is one of the examples of CRDTs proposed. The log in GDP can be thought of as an add-wins CRDT as no records are removed from the log once added. Our emphasis upon the high write availability of the log allows for replicas to temporarily diverge. The graph comparison algorithm and the simple set comparison algorithm serve as merge operations, and build a method specific to the data model set forth for GDP.

6 Future Work

We ran into limits with our simulation when trying to emulate large numbers of log servers above $n = 25$. Specifically, we ran into faulty performance with Mininet. Further testing of our replication system at with large numbers of log servers remains to be done.

Our design of the read and append semantics can still be implemented and optimized. Currently much of the burden of determining an answer for a query, such as the n most recent logs rests on the shoulders of the requester. Possiblities include having one of the replication daemons serve as a coordinator for a query.

Our simulation approximates the GDP log server as a SQLite database. Reads and appends communicated directly with the database. Operations such as verification of record signatures and the hash chain between log entries were omitted. The creation of signatures for the records are computationally expensive and could incentivize writers to batch their write or use larger payloads in records. Our approximation of log servers did not verify signatures either.

With a location independent naming scheme, GDP's current implementation does not have the ability to send messages to certain log servers. Our current implementation of replication interfaces with logs directly through the log's SQLite database to read and write to the log server. To communicate with other replicas, we use HTTP over IP as opposed to the GDP routing layer. More efficient compression of the record metadata for peer to peer communication is possible while still using HTTP. Future work includes utilizing an upcoming feature in GDP that can direct a request to a specific machine rather.

Our replication scheme utilizes a static set of peers for simplicity. The ability to add and remove peers is necessary for a system with high availability requirements.

Log servers might be retired or fail permanently without recovery. Similarly, a log should be able to increase the number of log servers it is replicated on as its usage changes.

Record headers may contain the hashes of older records beyond the prevHash. These hashes could help construct more highly-connected graph.

We create complete copies of the log at each log server. This creates a very high level of redundancy for each of the logs. Our replication service could achieve a higher degree of fault tolerance with a smaller storage and service burden per log server by having each log server store only certain records. Key distribution methodologies such as Chord hold promise to distributing the records and their retrieval for reads among long servers [13].

7 Conclusion

We have presented replication for the Global Data Plane. We have designed user semantics that support a highly write-available system. We have designed and implemented an architecture to carry out this replication and an anti-entropy algorithm to synchronize replicas. Results show that our graph comparison algorithm is able to utilize significantly less network bandwidth than our naive algorithm. Our graph comparison algorithm is able to recover quickly from large replica outages with a small number of log servers and recovers at least as fast as the naive algorithm. Results do not show the ability of our replication service to scale to large numbers of log servers. More work remains to be done in integration of our replication service with the current GDP implementation and testing at scale. We believe Replication for the Global Data Plane presents a promising basis for highly write available and fault tolerant systems on the Global Data Plane.

8 Acknowledgements

We would like to thank Professor John Kubiatowicz for introducing us to GDP and the many hours in his office discussing project direction. We would also like to give thanks to Nitesh Mor and Eric Allman for their time spent explaining to us the details of GDP and potentials paths to implementing replication.

References

[1] Security dvr and nvr record time calculator. `https://www.security-camera-warehouse.com/cctv-record-time-calculator.php`. Accessed: 2018-12-11.

[2] ATZORI, L., IERA, A., AND MORABITO, G. The internet of things: A survey. *Computer networks 54*, 15 (2010), 2787–2805.

[3] BAUER, S., CLARK, D., AND LEHR, W. Understanding broadband speed measurements.

[4] 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 *ACM SIGOPS operating systems review* (2007), vol. 41, ACM, pp. 205–220.

[5] DEMERS, A., GREENE, D., HOUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., AND TERRY, D. Epidemic algorithms for replicated database maintenance. *SIGOPS Oper. Syst. Rev. 22*, 1 (Jan. 1988), 8–32.

[6] EUGSTER, P. T., GUERRAOUI, R., HANDURUKANDE, S. B., KOUZNETSOV, P., AND KERMARREC, A.-M. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS) 21*, 4 (2003), 341–374.

[7] EUGSTER, P. T., GUERRAOUI, R., HANDURUKANDE, S. B., KOUZNETSOV, P., AND KERMARREC, A.-M. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS) 21*, 4 (2003), 341–374.

[8] GONÇALVES, R. J. T., ALMEIDA, P. S., BAQUERO, C., AND FONTE, V. Dotteddb: Anti-entropy without merkle trees, deletes without tombstones. In *Reliable Distributed Systems (SRDS), 2017 IEEE 36th Symposium on* (2017), IEEE, pp. 194–203.

[9] MERKLE, R. C. A certified digital signature. In *Conference on the Theory and Application of Cryptology* (1989), Springer, pp. 218–238.

[10] MOR, N., ALLMAN, E., PRATT, R., LUTZ, K., AND KUBIATOWICZ, J. D. An architecture for a widely distributed storage and communication infrastructure. Tech. Rep. UCB/EECS-2018-130, EECS Department, University of California, Berkeley, Aug 2018.

[11] PARKER, D. S., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B. J., WALTON, E., CHOW, J. M., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistency in distributed systems. *IEEE transactions on Software Engineering*, 3 (1983), 240–247.

[12] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free Replicated Data Types. Research Report RR-7687, July 2011.

[13] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 149–160.

[14] VAN RENESSE, R., DUMITRIU, D., GOUGH, V., AND THOMAS, C. Efficient reconciliation and flow control for anti-entropy protocols. In *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware* (2008), ACM, p. 6.

[15] VAN RENESSE, R., MINSKY, Y., AND HAYDEN, M. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (2009), Springer-Verlag, pp. 55–70.

[16] VERMA, S., AND OOI, W. T. Controlling gossip protocol infection pattern using adaptive fanout. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on* (2005), IEEE, pp. 665–674.