Tribune: An Externally Consistent Database Common Access API for the Global Data Plane

Matt Weber, Shiyun Huang and Lawrence Supian
Department of Electrical Engineering and Computer Sciences (EECS)
University of California, Berkeley
Email: matt.weber, jane.huang, lsupian@berkeley.edu

December 16, 2014

Abstract

The Global Data Plane (GDP) is a distributed data storage and routing system with the objective of providing persistence to information for devices in the internet of things. The GDP has a variety of useful properties including data security, a flat namespace, and location independent routing, that improve the availability and efficiency of universal data storage. As a key component in the Terraswarm project and the SwarmOS, the GDP provides log based storage and routing between sensors and actuators embedded in the physical world. Logs are the fundamental primitive of the GDP, but some CAAPIs (a Common Access Application Programming Interface) require stronger semantics. In this paper we introduce Tribune, a multi-writer log CAAPI that enforces Google Spanner-like external consistency. We compare the trusted Paxos based replication scheme with attack tolerant Byzantine Agreement and evaluate Optimistic Concurrency against the lock table-based concurrency control scheme used by Google Spanner. In a performance evaluation we found the optimistic algorithm to be faster than the comparable lock table approach for some workloads, particularly when paired with Byzantine agreement. Our long term goal is to provide PTIDES style deterministic execution semantics for swarmlets in the SwarmOS that choose to access time-aware multi-writer logs.

1 Introduction

The Internet of Things (IoT) presents a new paradigm for computer systems, enabling technologies like context-aware apps, extensible cyber-physical systems, large scale sensor data collection for machine learning, and smart cities. But systems of the future will not come for free: engineers will need ubiquitous computing infrastructure in the form of platforms, services, and tools as a foundation for their work. The Terraswarm project [13] seeks to address the gap between the development resources of today and those needed to engineer the coming swarm of interconnected devices [17]. There are many dimensions of the problem to be considered, including verification tools, low-power sensors, programming models, and persistent universally available data storage. The last point in particular is of primary concern to swarmlet (an application in the swarm) developers who need data accessible across different usage modes and application contexts [7].

The Global Data Plane (GDP) [8] is a TerraSwarm project that seeks to extend upon the capabilities of the Cloud to meet the needs of decentralized and interoperable swarmlets. Oceanstore [19] provides a starting point, as data must be globally available, durable, private, secure, and efficiently accessible. However, the GDP must also support storage and distribution of streaming sensor data, which it achieves through log-based data storage in a flat address space. As such, the log is a principle component of the GDP, and may be used as the fundamental building block for arbitrarily complicated systems for information representation. This design choice does not limit the efficiency or expressiveness of the GDP, because the GDP supports a Common Access Application Programming Interface (CAPPI) for data representations with sophisticated semantics or complex structure. Although possible, the process of reconstructing a full database or key-value store from log data would be prohibitively expensive if every time the sophisticated data structure were needed it had to be built from logs and then immediately thrown away. A CAAPI can maintain the current state of the enhanced structure directly and function as a sort of GDP cache for a particular data representation.

Although CAAPIs can be used in the GDP to store data efficiently, they also have the capacity to enforce semantics on the behavior of data operations. This work, Tribune, is an example of a CAAPI that enforces external consistency along with standard ACID semantics on the behavior of a multi-writer log. The general concept of this style of CAAPI is illustrated in Figure 1. Tribune's name is a reference to the role of an ancient Roman Tribune, an elected official with the power to intervene on behalf of the common people by vetoing legislation from the senate. In a similar respect, Tribune can control transactions that attempt to write to a protected log in this multi-writer merge style, and

abort transactions that violate its semantics.

This paper is organized as follows: Section 2 provides background information on the database behavior we seek to emulate in Tribune. Section 3 gives an overview of the system architecture and design. Section 4 goes into the implementation details for Tribune's most interesting algorithms. Section 5 elaborates on our test environment setup and presents benchmark results. In Section 6 we address the direction of future work. We conclude with Section 7.

2 Background

Tribune's primary role is to enforce database semantics on a multi-writer log. We chose the semantics of Google Spanner's read write transactions [6] because Spanner provides a time-based external consistency guarantee and works at a global scale.

2.1 Google Spanner

External consistency establishes the invariant that if transaction A commits before transaction B begins as observed from the outside world, the timestamp associated with transaction B in the database must be later than than A's timestamp in the database. Spanner achieves this invariant through a combination of two phase locking within a Paxos group, time-aware two-phase commit between Paxos groups, and precise implementation of the TrueTime API as shown in Figure 2. TrueTime establishes an ordering between timestamps that reflects error in clock synchronization, i.e. it is unknown whether a timestamp proceeds another unless the difference in the timestamps exceeds the known upper bound on the offset between clocks.

Spanner's high level architecture, shown in Figure 3, shares components and terminology with Chubby [4], Megastore [2], and BigTable [5], all Google projects as well. In Spanner, the tablets that are associated with a particular data model run on a set of replicas in the same Paxos group. The primary responsibility of the Paxos leader is to replicate the state change of a write across the group. The leader also acts as a bottleneck for all read write transactions, which allows it to maintain a lock table that reflects the status of transactions currently in flight. Clients that intend to perform read write transactions must acquire locks through two phase commit. Deadlocks are prevented with wound wait [21]. When a read write transaction needs to commit across paxos groups (a "distributed transaction"), one paxos leader steps up for the role of participant leader and runs a transaction manager to organize the two-phase commit. Spanner also supports read only transactions and read transactions (the two are actually distinct types in Spanner) that can be performed at any replica without going through the paxos leader.

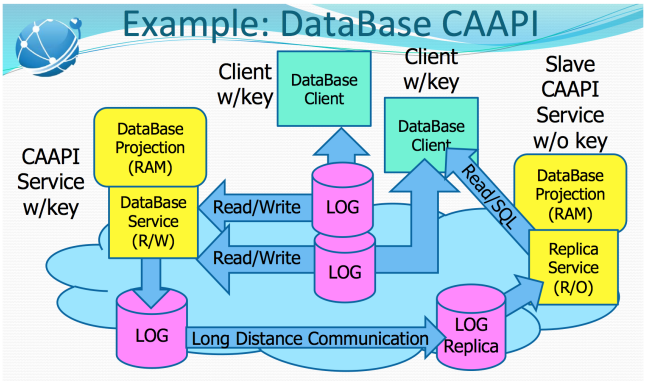

Figure 1. An illustration of a database CAAPI. Image from slide 17 of [8]

Method	Returns
$TT.now()$	$TTinterval$: $[earliest, latest]$
$TT.after(t)$	true if t has definitely passed
$TT.before(t)$	true if t has definitely not arrived

Figure 2. Spanner's TrueTime interface. Image from Table 1 of [6]. Note that t is a timestamp.

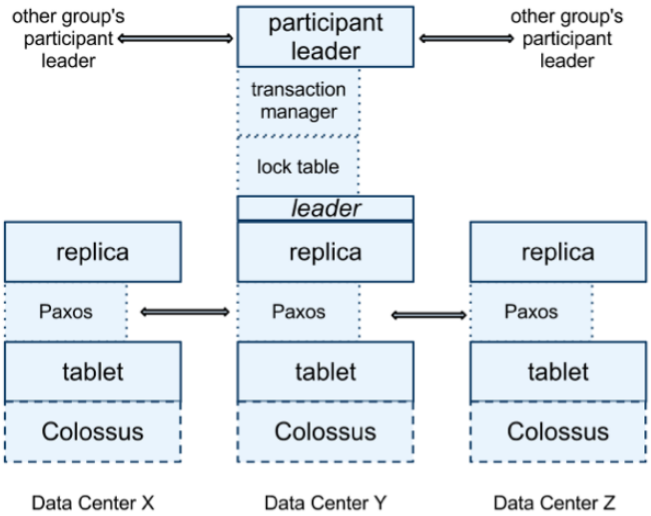

Figure 3. Spanner's Architecture. Image from Figure 2 of [6]

2.2 Time Synchronization

Spanner uses a global network of atomic and gps clocks to achieve time synchronization. An advantage of establishing very accurate clocks with very low drift is little communication is needed to preserve synchrony. IEEE 1588, the Precision Time Protocol [18] presents an alternative ip based mechanism to achieve synchronization on the order of microseconds across a wired local area network. Truetime is used in spanner to determine when to pick timestamps for transactions in two-phase commit and how long to hold locks after committing within a paxos group. Note that without any kind of synchronization between clients, relativity makes it impossible to evaluate external consistency. The definition of external consistency requires that transactions outside the system exist on a shared timeline (because they must be comparable) which implies some way to compare times at two locations, whether quantitative or logical [9].

2.3 Optimistic Concurrency

The work of [1] introduces an alternative to the lock table based concurrency control used in the majority of database systems including Spanner. Rather than officially communicate to the system as a whole that a transaction needs to read or write a particular value, optimistic concurrency performs reads and writes immediately wherever it is most efficient (potentially a local machine in a distributed system). Only when the transaction is finished does the entire system validate its correctness in the context of external consistency and serialization with two phase commit. Assuming conflicts are rare, Optimistic Concurrency offers the potential to be significantly faster than the locking approach. If conflicts are common, Optimistic Concurrency may be forced to abort most of its transactions.

2.4 Consensus Algorithms and Durability

Consensus algorithms like Paxos [10] and Byzantine Agreement [11] can be used to build a state machine with an unambiguous history across distributed, faulty machines. As a consequence, data is unambiguously replicated through the system – a desirable outcome for a global database that must remain consistent and available in the face of datacenter-wide failures.

Paxos and Byzantine Agreement address different failure modes for a distributed state machine. Byzantine agreement tolerates malicious faults while Paxos protocols assume the entire system is trustworthy. There is however, a performance tradeoff. For a system with 2N + 1 nodes, Paxos consensus allows for N nodes to simultaneously disappear. For a system with 3M + 1 nodes, Byzantine Agreement can tolerant M malicious nodes and requires expensive pairwise communication between all nodes to achieve consensus.

Note that availability, as provided by consensus algorithms, is not the same as durability [24]; the former relates to the presence of a backup-copy of a resource accessible in the short-term, where the latter refers to the capacity to protect data from accidental deletion. Reed-Solomon encoding [3] is a good approach for long-term data storage when the top objective is reducing the probability that data will be lost. But for short term data storage, using a consensus algorithm to achieve agreement on the state of a set of replicas is a much faster approach. Perhaps the Reed-Solomon encoding data might be verified by a Byzantine Agreement ring as in [19] but for low latency applications, other methods may be more practical. As a long-term objective for the GDP, the system ought to achieve a balance between both durable and responsive storage.

Another dimension of durability concerns the transfer of data from volatile to non-volatile memory such that a particular machine can recover from a crash or power outage. Although we didn't implement this part of a database in Tribune, algorithms that address this concern (such as [14] and [22]) are well known in the literature and are applied in most mature databases.

2.5 Routing in the GDP

Data must be routed to Tribune in the GDP through an overlay network. Oceanstore used Tapestry [26], although a distributed hash table approach such as Chord [23] or Bamboo [20] could also be used for routing. A project is currently underway in the GDP to efficiently route to the opaque identifiers that signify the location of a log server or a CAAPI.

3 System Architecture

The chief object of Tribune is to give Google Spanner's time stamp and external consistency semantics to a GDP CAAPI. Our design for Tribune is shown in Figure 4 and shows some high level similarities to Spanner. However, Tribune supports three modes in addition to Spanner's lock table and

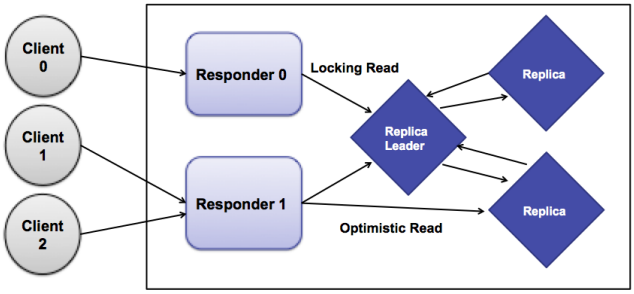

Figure 4. Tribune's Architecture.

Paxos configuration: an Optimistic Concurrency with Paxos mode, a lock table with Byzantine Agreement mode, and an Optimistic Concurrency with Byzantine Agreement mode. It isn't intended for Tribune to operate in multiple modes simultaneously, but Tribune could switch modes depending on application requirements.

3.1 Optimistic vs Locking Concurrency Control

The lock table with Byzantine Agreement mode was implemented with the expectation it would be a terrible design. We present it mainly as a Strawman Argument for purposes of illustrating how Byzantine Agreement is better used in conjunction with Optimistic Concurrency. This is why: trusting the Leader's lock table violates the Byzantine assumption that M components could be dishonest. The only option is to replicate the state of the lock table across the system, but the lock table is a highly concurrent data structure that is updated many times for each transaction. To make matters worse, Byzantine Agreement requires pairwise communication between Replicas whenever consensus must be reached.

The Optimistic approach need only check a transaction when it commits, and it need only compare it to previous transactions that were already replicated by the system for data storage redundancy. Since we already have timestamps paired with data values, the algorithm from [1] simplifies to a check of whether the read set of a transaction matches the current state of the database. Our experimental results in Section V confirm that Optimistic Byzantine Agreement is consistently faster than Locking Byzantine Agreement.

3.2 Tribune's Design

Our current design of Tribune implements a simplified version of Google Spanner without "distributed transactions". This decision means we do not implement the transaction manager which coordinates two phase commits between different Paxos groups. Our first design of the system assumes a one-paxos-group environment, so we only need a lock manager at the leader replica.

The Responder receives transactions from client applications. It parses each operation line by line and acquires read locks for all data required for the transaction. The Responder buffers writes locally and when all transaction operations finish, it tries to acquire write locks for the locally modified data. If the attempt turns out successful, the responder will try to commit the transaction at the leader.

We only implement the necessary components for read-write transactions. So all transactions would go through the leader replica to validate their respective lock leases before committing via Paxos protocol. If the validation is successful, the leader goes through all Paxos phases and returns the final transaction status (abort or commit) back to the Responder driving the transaction. The Responder will return the transaction status back to its client application.

3.3 Pseudo-Paxos and Pseudo-Byzantine Agreement

Our original intention for experimentation with Tribune was to integrate an open source Paxos and Byzantine Agreement package into the system, but in the final iteration of development we decided to simulate consensus algorithms instead. Although a variety of Paxos (and Raft [15] – a Paxos alternative) implementations exist, we were unable to find a Byzantine Agreement package. Even if we had found a project for the two algorithms, the full protocols would have features our Tribune implementation is unable to deal with. Specifically, our lock table makes the simplifying assumption that the leader does not change. Considering that Paxos leaders are long lived (10s) in Spanner, a permanent leader doesn't significantly impact the validity of our experimentation, but would leave a true Paxos/Raft library out of sync with Tribune. Instead, Tribune simulates the data replication and consensus work of Paxos and Byzantine agreement, a procedure we term Pseudo-Paxos and Pseudo-Byzantine Agreement.

As part of consensus simulation, we had to deal first-hand with a non-obvious implementation detail that arises in the practical implementation of Paxos: out-of-date replicas. We took inspiration from Chubby's solution to the problem [4] by using sequence numbers to catch a replica back up to the current state of the system when possible. It would be impractical to maintain a permanent map between paxos sequence numbers and changes to the database, so instead we truncate the log at a fixed threshold in the past. If a replica is further behind than the end of the log, it asks for a snapshot of the current state of the system from the leader, and achieves up-to-date status.

4 Algorithms and Implementation

In this section, we elaborate on the algorithms and interesting problem solutions we implement in Tribune. Benchmark results follow in Section V.

4.1 Programming Tools

We originally choose to implement Tribune in Java because the project team had familiarity with the language, and we felt a strongly typed, garbage collecting, object oriented language would reduce the frequency of programming errors in development. Java is known to perform "stop the world" garbage collection, in which execution periodically halts in all threads for 10s of milliseconds, but considering the low precision of our ultimate testing environment (three laptops) we decided it was an acceptable penalty given Java's other advantages.

As shown in Figure 4, client processes need to establish a connection with Responders, and Responders must communicate with Replicas, likely all across a network. Early on we experimented with basic java socket IO, but we realized we would have to repetitively implement the same programming idiom of a server that handles the connection and a parser thread that is responsible for reading a custom defined language of intended commands to be called on the receiving object. A little investigation found that this functionality was already available in the Java Remote Method Invocation framework [16], so we chose to use it instead of building a similar system from scratch.

However, RMI is not a perfect solution. The RMI server launched for each registered object does not use a thread pool; it simply forks off a new thread whenever it gets a connection. The Sun implementation of RMI exposes a parameter for the number of TCP connections that can be ongoing at once, but this doesn't limit the number of worker threads that can be alive in the system at a given time. Therefore with a sufficiently large number of clients spawning an arbitrarily large number of threads, the RMI server may be ill-conditioned. Switching to SEDA [25] for future work, would alleviate this problem, but significantly change our programming model.

RMI also presents a naming problem. Surprisingly, RMI does not support a remote rmiregistry where objects can be found directly by registration name. A remote object may only be registered on a local rmiregistry server with a name of the form "<ip address of host machine>/<registration name>". Since our performance evaluation was performed on machines without static ip addresses, we potentially faced an addressing nightmare where the ip address of remote objects would change sporadically. To address this issue, we implemented a naming service over RMI we called Remote Registry, that maps a "<registration name>" to a full network-ready string of the form "//<ip address of host machine>/<registration name>". With Remote Registry we could publish not only the location of an object when its ip address changed, but also the availability of resources during initialization.

4.2 Initialization

At first, our team underestimated the problem of initializing Tribune. Both Responders and the Leader replica from Figure 4 depend on other components to correctly respond to RMI calls. The problem we encountered was that it is insufficient for a component to be ready to respond to RMI calls, it must also be ready to service requests which themselves may involve RMI calls on uninitialized system components.

Tribune uses the remote registry described in the previous section to address this issue. The first step of initialization is to clear the remote registry. If we enforce the invariant that a service only register itself in the remote registry when it is fully initialized and ready to service requests, other components can poll the remote registry in initialization to check their dependencies. Fortunately, Tribune's dependency graph is acyclic because Replicas can always register with no dependencies. From that starting point, the leader replica can initialize (being cognizant of the fact it is already registered as a Replica), which allows Responders to begin. Finally when the appropriate Responder is registered, a Client application can begin requesting the Responder to perform its transactions.

4.3 Strict Two-Phase Locking

In our first design of the system, we implement a lock manager to enforce strict two-phase locking in the one Paxos group scenario. Locks have time-based leases that must be extended by the corresponding Responder if a transaction is not to abort. The lock manager keeps track of four data structures: a map from keys to the current lock holder(s) called lockMap, a map from transaction IDs to their birth dates called transactionBirthdateMap, a map from keys to all transactions waiting to get the lock for the key called waitingTransactionMap and a map from committing transaction IDs to the locks they are holding called committingTransactionLocksMap. We use these data structures to store information required for the wound wait deadlock resolution algorithm, the lock killer that deletes expired locks, validating locks before a transaction commits, releasing all associated locks of a committed transaction and extending current lock expiration times in lockMap.

Releasing associated locks of a committed transaction removes the lock entries in lockMap as well as the transaction entries from transactionBirthdateMap and committingTransactionLocksMap. The lock manager extends lock leases for non-aborted transactions after receiving heartbeats from the Responder. Validation of locks goes through all locks held by the transaction intending to commit and checks that the locks of the correct access mode are still present in the lockMap and have not expired. The lock lease killer goes through the lockMap at regular interval to kill already expired lock leases and wake up the waiting transaction(s) to get the lock for a specific key.

4.4 Wound Wait Deadlock Resolution

The most nuanced part of the lock manager is its implementation of the wound wait algorithm. The algorithm is called whenever a new waiting transaction is added to waitingTransactionMap or any entry of lockMap is removed to determine if any waiting transaction qualifies to acquire a lock it needs. We exploit a small optimization trick in our implementation by using transactionBirthdateMap to detect locks which are held by already dead transactions. The rationale is if there is no entry contained in transactionBirth-

dateMap for a specific transaction which holds a lock in lockMap, the transaction must already be aborted by the lock manager. The entry in the lockMap for the specific transaction is stale, so a waiting transaction on the same key could safely get the lock it needs. To keep this invariant, if a running transaction loses its ownership of any of its already acquired locks, it should be considered aborted by lock manager and its entry in transactionBirthdateMap will be eventually removed.

Also, we use committingTransactionLocksMap to "lock" the locks held by a committing transaction in the lockMap, so they will not be accidentally killed by the wound wait algorithm or lock lease killer while it is committing.

The psudo-code for wound wait is given in Figure 5. Notice "incoming transaction waits" is actually calling "return" in the method; "incoming transaction acquires the lock" involves setting the the lease expiration time of the new lock lease and waking up a waiting thread using a condition variable to notify it the lock is successfully acquired.

We omit a small yet important detail for the last case, i.e. when incoming transactions request a write lock and all current lock holders are reads. Since responders only acquire read locks for their locally buffered writes, they need to "upgrade" their read locks to write locks for the buffered writes before they attempt to commit. These upgrades would fall into the last case classified by Figure 5's branch conditions. If all other current lock holders are younger than the upgrading transaction and thus an upgrade is valid, the transaction birthdate entry for the upgrading transaction is not removed during the removal of lockMap and transactionBirthdateMap entries of killed lock holders.

4.5 Lock Acquisition

We implement an RMI method named "getReplicaLock" that is intended to either block until the lock has been acquired or fail. Internally, the lock manager either returns an abort status if the calling transaction is already aborted by lock manager or puts itself in waitingTransactionMap and blocks until acquiring the requested lock. We implement a container class called LockAndCondition which consists of information about a lock lease and an object serving as a condition variable called wakeUpCondition to wake up the waiting RMI method when the transaction acquires the lock it needs and leaseExpirationTime is set by lock manager. We rely on java explicit synchronization to guarantee the correct ordering of the executed code. Also notice that we start another thread in getReplicaLock to add the waiting transaction to waitingTransactionMap and invoke the wound wait algorithm to determine if the waiting transaction could supersede the current lock holder(s) to acquire the lock for the key. This guarantees the waiting transaction immediately gets the lock it needs without any actual

```
wakeUpNextTransaction(int key) {
define incoming transaction to be the
    oldest transaction among all
    transactions waiting to acquire lock of
    a specific key

if incoming transaction does not have an
    entry in transactionBirthdateMap:
    return abort status
else:
  If lock is free:
    incoming transaction acquires the lock
  else if incoming transaction requests a
      read lock and all current lock
      holders are reads:
    incoming transaction acquires the lock
  else if incoming transaction requests a
      read or write lock and current lock
      holder is a write:
    if current lock holder does not have
        an entry in
        transactionBirthdateMap:
      incoming transaction acquires the
          lock
    else if current lock holder is older:
      incoming transaction waits
    else if current lock holder belongs to
        a committing transaction:
      incoming transaction waits
    else removes lockMap and
        transactionBirthdateMap entries of
        current lock holder and incoming
        transaction acquires the lock
  else if incoming transaction requests a
      write lock and all current lock
      holders are reads:
    if any of the current lock holders
        belong to a committing transaction:
      incoming transaction waits
    else if any of the current lock
        holders who still have entries in
        the transactionBirthdateMap is
        older than the incoming
        transaction:
      incoming transaction waits
    else removes lockMap and
        transactionBirthdateMap entries of
        all current lock holders and
        incoming transaction acquires the
        lock
}
```

Figure 5. Psuedocode for the lock manager's wound wait algorithm.

waiting if it is already available.

5 Experiments and Results

In this section we present Tribune's performance when given a variety of transaction workloads. All transaction workloads are randomly generated and execute concurrently in Tribune. We first evaluate a suite of lengthy high conflict transactions, next we present the performance of a sample bank application that runs a high volume of short transactions between accounts, and finish with a high volume test of reads to a small set of memory locations to simulate a memory hotspot.

5.1 Experimental Setup

A graphic representation of our setup can be seen in Figure 6. We elected to run Tribune on three Macbook Pros as opposed to a cluster because we need easy physical access to the machines and admin status to run a Precision Time Protocol Daemon (PTPD). Specifically we run ptpd2, an open source implementation of IEEE1588. Using a Linksys router to establish a local area network over ethernet, we established a 10 μs average offset from the master clock to the slaves. Going six standard deviations from the mean, we determined a 700 μs value for the clock error in our implementation of TrueTime would upper bound the actual clock error in almost every case.

As for hardware and operating system specifications, the IEEE 1588 master machine in the upper left of Figure 6 has OS X Version 10.9.3, 2.4 GHz Intel Core i7, and 8 GB 1600 MHz DDR3. It runs the PTPD master and the consensus leader because it is our strongest machine. The laptop at the bottom of the diagram has OS X Version 10.10.1, 2.5 GHz Intel Core i5, and 4 GB 1600 MHz DDR3, and the one in the upper right has OS X Version 10.9.3, 2.3GHz Intel Core i7, and 16 GB 1600 MHz DDR3. Note that all Client programs run on the machine in the upper right. It evenly distributes client program requests between each of the two Responders. Transaction running time is measured on this laptop using a call to Java 8's `Time.Instant.Now()` function before sending the transaction to a Responder and after receiving a "commit" or "abort" message from Tribune.

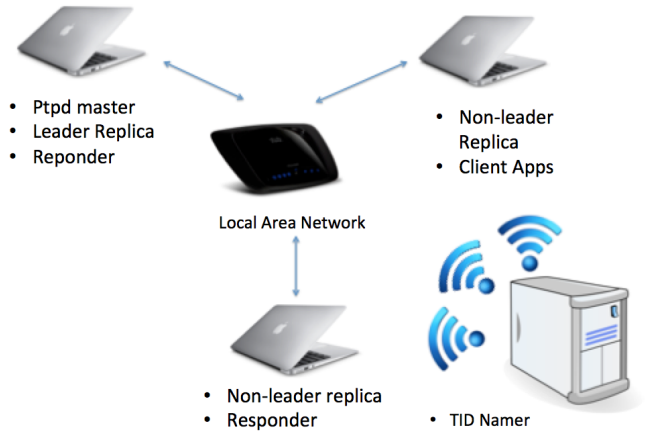

Figure 6. Experimental Setup for Evaluation

The final component of our test setup is a transaction GUID namer running on a server in the same building on UC Berkeley's EECS department network. Laptops connect to it through WiFi routers in our lab area. Ping time to the server is only slightly longer than to the LAN laptops so we expect this to have had a negligible impact on performance.

5.2 High Conflict Transactions

Our first benchmark consists of very long transactions with a high probability of accessing the same memory address for reads and writes. Therefore for both optimistic concurrency and locking control we expect to have a high percentage of aborts when many transactions are running on Responders. The specific format of the experiment shown in Figures 7 and 8 is to run a variable number of clients (iterating from 1 to 10) concurrently in Tribune. This means we begin the trial with one client running and when it finishes we start two clients running, and so on. The name "Locking Paxos" refers to lock table based concurrency control with Paxos replication, "Optimistic Byzantine" refers to Optimistic Concurrency control with Byzantine replication, and the other runs are configurations of those features that correspond to the names in the legend. We use the same naming convention for the other experiments. Each client runs a list of three transactions, each consisting of one thousand commands (i.e. read, write, add, etc.) to be executed . Because many transactions abort shortly after starting, we only report the running time in Figure 7 for transactions that commit.

As expected, the Byzantine configurations were slower on average than the Paxos setups. The mean transaction execution times were also consistent

Note that the abort rate in Figure 8 is quite low for larger numbers of concurrent clients dropping below 20% in some cases. As a general trend within a particular system configuration, the commit rate decreases as the contention increases. However, it's very surprising that the optimistic approaches resulted in fewer aborts than the lock based ones - as one would typically expect lock-free reads and writes in high conflict transactions to consistently step on each other. We give two possible explanations for why this happened. First, the failure mode of optimistic concurrency in high conflict scenarios is mitigated in our experiment because there are only two replicas where a read might take place, and given the assumptions of our experiment, neither one

is likely to be considerably out-of-date. Second, wound wait deadlock resolution in the lock table may overzealously abort transactions that ought to have been able to commit in an optimistic scenario.

5.3 Bank Application

The second benchmark focuses on a more realistic application where data values might be important: a bank scenario similar to an ATM or teller. Like the high conflict experiment, we generate random workloads, but for each client, we reduce the number of commands for each transaction inside the client from 1000 to 20. To compensate we start 10 times as many clients on each iteration of the experiment and run 10 transactions per client. We start with ten client apps and increment by ten for every run up to one hundred. The objective is to present a realistic use case where conflicts over data appear, without being forced to abort transactions as often as in the high conflict scenario.

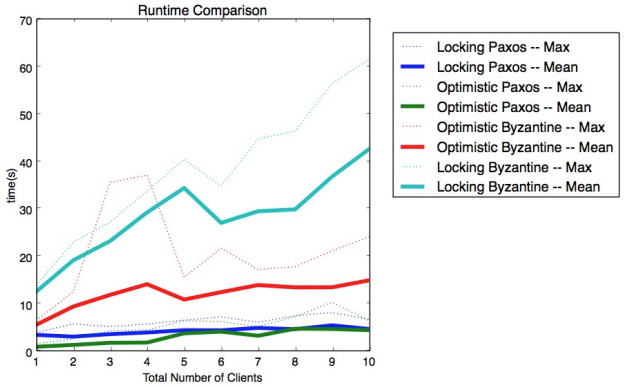

Figure 7. High Conflict Rate: Committing Transaction Execution Time

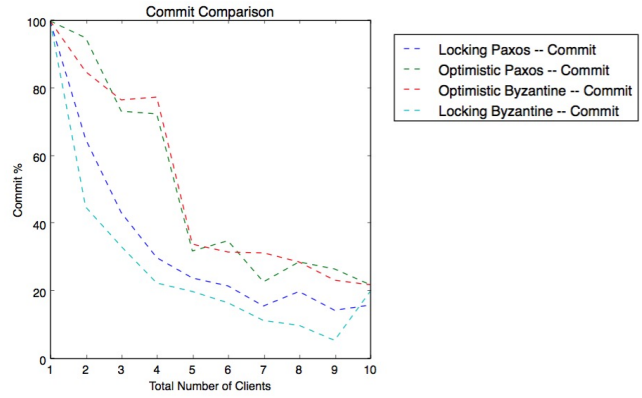

Figure 8. High Conflict Rate: Commit Percentage

It is very odd that the mean execution time of Optimistic Paxos is slower than Optimistic Byzantine in Figure 9, considering that Byzantine agreement is simulated in Tribune at this stage by doing strictly more work than Optimistic Paxos in the form of pairwise communication between replicas. Perhaps it was a fluke of the random test generation or nondeterministic network behavior. However it is no surprise that Locking Byzantine is consistently the slowest, considering that anytime the lock table is modified, Byzantine agreement requires pairwise communication to keep the Leader honest.

The commit percent is much higher for this experiment than for the high conflict run. Now, the lock-based schemes seem to have marginally higher commit rates than the optimistic ones.

5.4 Hot Spot Reads

Our last benchmark consists of many clients trying to read values that already exist in data storage. We are interested to know what will happen when there are many reads for a few memory addresses and no writes at all except during initialization. The idea for this kind of benchmark came from the common case on media websites like Youtube, Google Play, or Spotify where millions of people watch or listen to (but don't write) a specific memory location at the same time. Ideally, reads in this scenario should have minimal performance impact towards each other since reads alone cannot cause a transaction to abort.

In the test scenario, an initalization client writes data to one specific memory address, then the rest of the clients simultaneously read from that memory address once the first client is done. Like the Bank experiment, the simulation started with ten as the number of concurrent clients and iterated up to one hundred clients by intervals of ten. The result is shown in Figure 11 on a log scale graph to spread out the three runs near each other on the bottom. The run times are so similar, the log scale doesn't make much of a difference to differentiate them. The Locking Byzantine agreement configuration is an order of magnitude slower than the others because it must perform substantially more consensus work. Note that the commit percent graph is omitted for this trial because 100% of transactions committed successfully.

6 Future Work

Spanner-style semantics concerning time are most certainly not the only option for a CAAPI in the GDP. PTIDES [27] provides both an execution model and a simulation environment for time-aware computation. Rather than dealing with transactions, PTIDES determines the execution of atomic events and enforces that sensor-to-actuator deadlines are met. As an advantage of tight time synchronization

and atomic events, PTIDES can define a deterministic time-ordered merge between two input streams by simply waiting

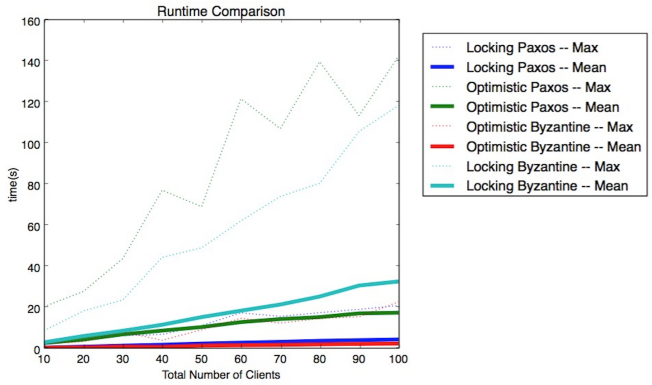

Figure 9. Bank Application: Committing Transaction Execution Time

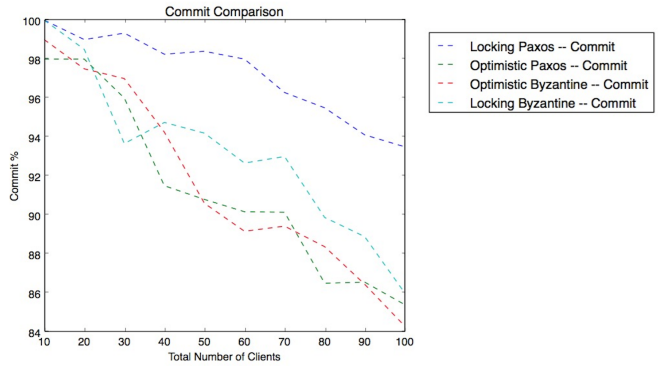

Figure 10. Bank Application: Commit Percentage

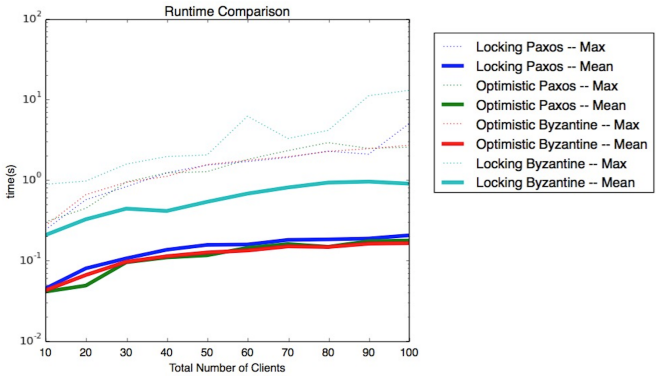

Figure 11. Hot Spot Reads: Committing Transaction Execution Time

out the clock uncertainty before writing. This policy guarantees no future event could be timestamped further in the past than the written value.

Reconsidering our implementation choices now the project is complete, Java and RMI may not be the best language and communication system for Tribune. C would be considerably faster in some of the computationally expensive bottlenecks such as the lock table. RMI gives a strange programming model and has a lot of computational overhead in it's operation. The biggest implementation challenges we faced had to do with strange interactions with threads arising out of RMI, which is no surprise considering that threads intrinsically convolute the model of computation [12]. In the future we will look into alternative message-based IPC.

7 Conclusion

We present Tribune, a CAAPI for the GDP that enforces strong semantics on the behavior of a multi-writer log. Tribune uses precise clock synchronization to establish external consistency. Data may be replicated in geographically disperse locations, increasing the odds that useful values are physically near their client machines. When it comes to concurrency control, Tribune exhibits better performance in some applications with Locking and better in others with Optimistic transactions. Our experiments show the combination of Locking and Byzantine Agreement to be a particularly slow choice, motivating the use of Optimistic concurrency control when Byzantine Agreement is desired.

8 Acknowledgements

Thanks to Dr. Patricia Derler for her input on Spanner and time synchronization. Also thanks to Prof. John Kubiatowicz for numerous discussions about the GDP and CAAPIs.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record*, 24(2):23–34, 1995.

[2] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Lon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[3] J. Blmer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An xor-based erasure-resilient coding scheme, 1995.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Princi-*

ples of distributed computing, pages 398–407. ACM, 2007.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, and P. Hochschild. Spanner: Googles globally-distributed database. In *Proceedings of OSDI*, volume 1, 2012.

[7] P. Dabbelt and J. D. Kubiatowicz. A case for the universal dataplane, September 2013. Presented at theFirst International Workshop on the Swarm at the Edge of the Cloud (SEC'13 @ ESWeek).

[8] J. D. Kubiatowicz. Enabling the swarm through the global data plane, November 2014.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[10] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[11] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[12] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[13] E. A. Lee, J. D. Kubiatowicz, J. M. Rabaey, A. L. Sangiovanni-Vincentelli, S. A. Seshia, J. Wawrzynek, D. Blaauw, P. Dutta, K. Fu, and C. Guestrin. The TerraSwarm research center (TSRC)(a white paper). Technical report, Technical report UCB/EECS-2012-207, 2012.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[15] D. Ongaro and J. Ousterhout. *In search of an understandable consensus algorithm (extended version)*. 2014.

[16] E. Pitt and K. McNiff. *Java.Rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[17] J. M. Rabaey. The swarm at the edge of the cloud-a new perspective on wireless. In *VLSI Circuits (VLSIC), 2011 Symposium on*, pages 6–8, 2011.

[18] R. Ratzel and R. Greenstreet. Toward higher precision. *Communications of the ACM*, 55(10):38–47, 2012.

[19] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *FAST*, volume 3, pages 1–14, 2003.

[20] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. *Handling churn in a DHT*. Computer Science Division, University of California, 2003.

[21] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, June 1978.

[22] R. Sears and E. Brewer. Segment-based recovery: write-ahead logging revisited. *Proceedings of the VLDB Endowment*, 2(1):490–501, 2009.

[23] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Networking, IEEE/ACM Transactions on*, 11(1):17–32, 2003.

[24] H. Weatherspoon, P. Eaton, B.-G. Chun, and J. Kubiatowicz. Antiquity: exploiting a secure log for wide-area distributed storage. *ACM SIGOPS Operating Systems Review*, 41(3):371–384, 2007.

[25] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.

[26] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.

[27] J. Zou, S. Matic, E. A. Lee, T. H. Feng, and P. Derler. Execution strategies for PTIDES, a programming model for distributed embedded systems. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 77–86. IEEE, 2009.