Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores

Faisal Nawab Vaibhav Arora Divyakant Agrawal Amr El Abbadi

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{nawab,vaibhavarora,agrawal,amr}@cs.ucsb.edu

ABSTRACT

Cross datacenter replication is increasingly being deployed to bring data closer to the user and to overcome datacenter outages. The extent of the influence of wide-area communication on serializable transactions is not yet clear. In this work, we derive a lower-bound on commit latency. The *sum* of the commit latency of any two datacenters is at least the Round-Trip Time (RTT) between them. We use the insights and lessons learned while deriving the lower-bound to develop a commit protocol, called Helios, that achieves low commit latencies. Helios actively exchanges transaction logs (history) between datacenters. The received logs are used to decide whether a transaction can commit or not. The *earliest* point in the received logs that is needed to commit a transaction is decided by Helios to ensure a low commit latency. As we show in the paper, Helios is theoretically able to achieve the lower-bound commit latency. Also, in a real-world deployment on five datacenters, Helios has a commit latency that is close to the optimal.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Transaction processing; Distributed databases*

Keywords

Cloud computing; geo-replication; multi-datacenter

1. INTRODUCTION

A key requirement for today's Internet services is availability. Complete outages of datacenters have been a major cause of disruption to availability. These outages can be due to technical problems [1], unforeseen natural disasters [39], or due to a sudden surge in traffic [2]. Datacenter outages can be overcome with the aid of geo-replication. However, replicating data at such large distances accentuates the trade-off between consistency and performance. Many solutions in the last decade investigated sacrificing consistency to deliver better performance [5, 15, 19, 48, 52]. However, in the last few years, interest in providing transactional guarantees has been increasing in both industry [9, 16] and research [27, 44, 47, 54] communities. This interest is due to the need for transactional guarantees for many applications, such as e-commerce. Furthermore, developing applications on top of an eventually consistent data store is complex [42]. Pushing the problem of handling consistency to the application developer is error-prone and debugging these problems for eventually consistent data stores is a complex task.

We focus on systems that provide serializable transactions. Committing arbitrary transactions requires coordination between datacenters mainly for detecting conflicts [6]. This coordination leads to higher commit latencies. Communication latency between datacenters can be in the order of hundreds of milliseconds. Studies show that a response time of more than 200ms can drive customers away [46]. This has motivated a plethora of research work that focuses on designing protocols that provide serializable transactions with the objective of achieving better commit latency [27, 38, 40, 54].

We develop a lower-bound on transaction *commit latency* that is due to the coordination needed to detect conflicts. The commit latency is defined as the time it takes the datacenter to decide whether a transaction executing can be committed or not. A transaction t cannot commit unless the datacenter is certain that there can be no other concurrent transactions that conflict with t, which leads to the need of coordination between datacenters. As we will demonstrate, *for any two datacenters to maintain serializability, the summation of their commit latencies cannot, in any case, be lower than the RTT between them.* The lower-bound result is applicable to a group of datacenters each maintaining a full replica of the data. Any pair of datacenters must satisfy the lower-bound commit latency.

We also propose Helios, an optimistic commit protocol influenced by the lower-bound study. It allows manual tuning of commit latency as long as the commit latency does not violate the lower bound. Each transaction is timestamped by a local loosely synchronized clock. The transaction commits by waiting for transaction information from other datacenters. The amount of received information needed to commit a transaction depends on the transaction's timestamp. Datacenters exchange their *preparing transactions* (transactions trying to commit) and *finished transactions* (committed or aborted transactions) using an ordered shared log. As logs arrive, the datacenter decides which local preparing transactions can be committed given the new information. He-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2723729.

lios judiciously decides the earliest point in the received logs that will enable committing a transaction. Recognizing the *earliest* point in the received logs to commit a transaction will lead to lower commit latency. We focus in this paper on achieving the lowest *average* commit latency. However, Helios allows tuning the commit latencies of individual datacenters to achieve other objectives.

Helios cleanly separates the protocol to guarantee serializability from the mechanism to ensure liveness in the presence of failures. This design is followed by other geo-replicated protocols such as Spanner [16] and Scatter [23] that use Two-Phase Commit (2PC) to guarantee consistency and Paxos [30, 31] to perform state replication. Helios leverages a separate synchronous replication component and augments it with the Helios commit protocol. This clear separation allows the flexible use of any replication protocol such as Paxos. Paxos, however, requires two rounds of communication. Also, it replicates to a majority of datacenters and does not provide any flexibility in setting the number of tolerated datacenter outages. We design a state replication protocol that utilizes the replicated log used for the commit protocol to ensure liveness in the presence of failures while allowing the flexibility of setting the number of tolerated datacenter outages.

Related work is presented in Section 2. Section 3 derives a lower bound on transactions running on replicated data. The Helios commit protocol and replication are described in Section 4. Then, evaluation results are presented in Section 5. The paper concludes in Section 6.

2. RELATED WORK

There is a large body of work on the management of replicated data [26]. Here we focus on work that targets or is related to wide-area replication.

Geo-replication. Early solutions for geo-replication proposed weakening consistency guarantees in favor of performance [5, 15, 19]. At one end of the spectrum are systems that provide eventual consistency [7] like Cassandra [5] and Dynamo [19]. Other systems provide different forms of consistency that are stronger than eventual consistency. Causal consistency, inspired by the causal ordering concept [29], preserves causal relations between transactions in a replicated data environment [8, 28, 36, 37]. COPS/Eiger, for example, define Causal+ consistency, which adds a data convergence guarantee to causal consistency. However, they do not provide general transactions. Snapshot Isolation [11, 26] is another notion of consistency. It prevents concurrent write-write conflicts and ensures that a transaction always reads a consistent snapshot of the data. Solutions in this category often rely on exchanging write-sets of transactions to decide whether to commit or abort [18, 33, 34, 47].

Relaxed consistency guarantees, however, are unsuitable for a wide-range of applications that require strong consistency. Also, developing applications on top of data stores that provide relaxed consistency guarantees is complex [42] and debugging consistency problems of these applications is error-prone.

Serializability. The need for stronger guarantees inspired the design of systems such as Megastore [9] and Paxos-CP [44] that use Paxos [30, 31] to serialize a replicated log of transactions. The use of Paxos, however, leads to higher commit latency due to the multi-round nature of the protocol. This also applies to distributed transaction managers that provide cross-datacenter replication via a Paxos replication of storage, such as Calvin [50]. Other systems use traditional commit protocols to manage transactions. Spanner [16] and Scatter [24] use Two-Phase Commit to manage transaction consistency and underneath it Paxos is used to ensure storage durability. The use of 2PC and Paxos incur multiple cross-datacenter messages making the commit latency high. Recent works identified this trade-off between latency and consistency and designed systems that will adapt accordingly to manage this trade-off. Pileus [48] allows developers to prioritize the level of consistency and latency requirements. SPANStore [52], on the other hand, optimizes for cost function given an application's consistency, latency, and fault tolerance requirements along with its workload characteristics. However these systems still incur a latency penalty similar to other systems when providing strong consistency. Bailis et al. [6] propose invariant confluence analysis, which determines whether a coordination-free execution is safe, thus allowing committing a transaction with no wide-area latency. Safety is determined according to application-level invariants provided by the programmer. The wide-area latency is still observed by transactions that are not invariant confluent.

Low latency. Large wide-area communication latency affects the cost of coordination and thus dramatically increases the commit latency of transactions. This resulted in a large body of research work that focuses on achieving lower commit latencies for serializable transactions. Transaction Chains [54] leverages *a priori* knowledge of transactions to perform a static analysis of conflicts in order to structure commitment plans that result in low latency. Not having this *a priori* static analysis will limit the benefits of Transaction Chains. MDCC [27] uses Fast Paxos [32] to enable committing transactions in one round of communication to a single, but larger than a majority, quorum. Replicated Commit [38] uses Paxos to drive inter-datacenter communication and 2PC and 2PL to manage transactions within a datacenter with the objective of minimizing commit latency. Message Futures [40] reserves commit points for future transactions to achieve low commit latency. Message Futures and Replicated Commit are part of the evaluation study in Section 5. More information on Message Futures and Replicated Commit are provided in Section 5.2.

Clock synchronization for conflict detection. The use of loosely synchronized clocks has been proposed for conflict detection [3, 4, 16, 21, 35]. The Thor project proposes using loosely synchronized clocks for conflict detection [3, 4]. Clients execute transactions optimistically and transactions are assigned a commit timestamp from a physical clock that will translate into a serial order. Other systems use a similar approach for different consistency guarantees like Clock-SI [21] for snapshot isolation, and Orbe [20] and GentleRain [22] for causal consistency. Spanner [16] uses synchronized clocks with bounded uncertainty by TrueTime to achieve external consistency. Spanner, unlike Helios and the aforementioned systems, *requires* high-precision clocks for the correctness of external consistency.

Log-structured data stores. Building a distributed data store using a shared log has been investigated in the literature [12, 40, 41, 43, 51]. Hyder [12] builds a multi-version log-structured database over a distributed shared log storage. A transaction executes optimistically on a snapshot of the database and broadcasts the record of changes to *all* servers and appends a record of changes to the distributed shared log. The servers then commit the transaction by detecting conflicts in the shared log. LogBase [51] and RAMCloud [43] are also multi-version log-structured databases built over a shared log. Helios, like these systems, is also an optimistic concurrency control manager that leverages a shared log for validation. However, Hyder, LogBase and RAMCloud are not designed for a geo-replicated environment. Message Futures [40], mentioned above, uses a geo-replicated shared log. Like Helios, Message Futures uses Replicated Dictionary [53] for validating transactions. Replicated Dictionary is a causally ordered shared log for single atomic events. It has several features that distinguish it from other log replication techniques that maintain causal order [10, 28, 45, 49]. Replicated Dictionary maintains a timetable representing the extent of each replica's knowledge. This timetable is used for various tasks like garbage collection and transitive propagation between replicas. It does not, however, provide transactional support. Helios extends the basic data structures of Replicated Dictionary to achieve transactional geo-replication with minimal commit latencies. Chariots [41] is a design of a scalable geo-replicated shared log, influenced by Replicated Dictionary [53]. Chariots might provide a more efficient log replication framework for Helios.

Helios is a system that supports **serializable transactions** on **geo-replicated** data stores with the objective of achieving **low latency** by leveraging **loosely synchronized clocks** and **log replication**. Helios starts with a clear understanding of the true limits of achievable commit latencies, which leads to a design that is theoretically able to achieve the optimal commit latency and that is practically able to achieve latency close to the optimal. Helios allows tuning the commit latency of individual datacenters and allows the flexibility to set the desired fault-tolerance level in terms of the number of datacenter outages to be tolerated.

3. COMMIT LATENCY LOWER-BOUND

The objective of this section is to develop a lower-bound on commit latency of transactions on replicated data stores *while maintaining serializability* [13]. Maintaining serializability requires coordination between replicas (datacenters in our case). The communication latency necessary for this coordination imposes a limit on *commit latency*, which is the time duration to decide whether a transaction commits or aborts. Achieving low commit latency is the focus of this study. Consider two datacenters A and B with unique commit latencies L_A and L_B, respectively. We show in this section that the *summation* of L_A and L_B must be at least the Round-Trip Time (RTT) between A and B. Note that this is a summation which means that the commit latency of a datacenter can be lower than RTT.

The lower-bound result extends to larger groups of datacenters by applying the lower-bound to all pairs in the group. This will allow us to judge whether the group of datacenters can commit with a certain set of commit latency values. We are particularly interested in minimizing the average commit latency of all datacenters. We call the minimum average latency a *Minimum Average Optimal* (MAO) latency or optimal latency for short.

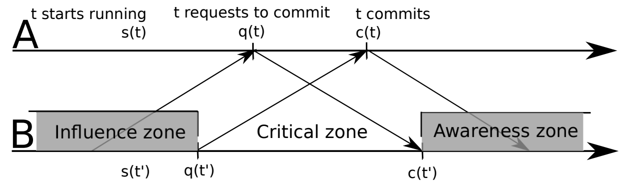

Figure 1: Two transactions, t and t', executing in a scenario with two datacenters

3.1 Theoretical model and assumptions

We consider a theoretical model that consists of datacenters with communication links connecting them. Each transaction undergoes two phases. First, the transaction is issued and it becomes visible to the datacenter. At that stage it is called a *preparing* transaction. Then, at a later time the datacenter decides whether it commits or aborts and it becomes a *finished* transaction. The time spent as a preparing transaction is the *commit latency*.

The following are the assumptions on communication and computation for this model. *These assumptions are made for the theoretical development of this section only and are not part of the Helios system design in Section 4.*

- *Compute power:* Infinite compute power is assumed in the model. The datacenter does not experience any overhead in processing and storing transactions. We make this assumption to focus our attention on communication overhead.

- *Communication links:* Sending a message through a link takes a specific latency to be delivered to the other end. Links are symmetric and take the same amount of time in both directions. Note that different links could have different latencies. However, triangle inequality must hold.

- *Arbitrary read-write transactions:* All datacenters have no restrictions on their choice or order of objects to be read or written in a transaction. Additionally, each transaction must have at least a single write operation. Thus, the model does not apply to optimizations for read-only transactions and disjoint data manipulation techniques. Also, transactions must try to commit, hence aborting all transactions is not allowed.

- *Knowledge:* Each datacenter A knows precisely every preparing and finished transaction that exists at another datacenter B up to the current time minus half the RTT between them, *i.e.*, *now* - $\frac{RTT(A,B)}{2}$. This reflects the fastest time a datacenter knows about any event in another datacenter. In a realistic setting this is a lower bound of such knowledge.

- *Commit latency:* We assume that the commit latency at each datacenter is fixed. This assumption simplifies the presentation. The discussions can be extended to the general case by taking each point in time in isolation.

3.2 Lower-bound proof

Intuition. For any two concurrent conflicting transactions, at least one of them must be able to detect the other before committing. Otherwise, both transactions will commit, which could result in incorrect executions. Here, we show that there is a lower-bound on commit latency. If the commit latency is lower than the lower-bound, then two conflicting transactions could commit without detecting each other, thus possibly violating correctness.

Formulation. Consider two datacenters A and B and a transaction t executing at datacenter A and transaction t' executing at datacenter B that could be conflicting with t. Figure 1 shows these transactions. In the figure, $s(t)$ is the transaction's start time, $q(t)$ is the commit request time, and $c(t)$ is the commit time. Transaction t's read and write-set are visible at the commit request time. Given the knowledge assumption, B knows about t starting from time $q(t) + \frac{RTT(A/B)}{2}$. Transaction t is preparing from time $q(t)$ until time $c(t)$ when it is committed. Three *zones* are defined at datacenter B with respect to t: (1) The *awareness zone* where B can possibly know about t, (2) The *influence zone* where B's transactions can be known to t, and (3) the *critical zone* where B is neither in the awareness nor influence zone.

Lemma 1. *The sum of the commit latencies of two datacenters is greater than or equal to the RTT between them,* i.e.*,* $L_A + L_B \geq RTT(A, B)$*, where* L_X *is the commit latency at datacenter* X*.*

Proof: Let the time when t requests to commit be $q(t)$ and the time it commits be $c(t)$, *i.e.*, $c(t) - q(t) = L(t)$. These times are illustrated in Figure 1. The earliest time that B can be aware of the commit request made by t is at time $q(t) + \frac{RTT(A,B)}{2}$, since half the RTT is needed to get a message from A to B. Thus, t can affect B only at the *awareness zone* which is at any time greater than or equal to $q(t) + \frac{RTT(A,B)}{2}$. Likewise, t cannot be affected by events at B that happened after time $c(t) - \frac{RTT(A,B)}{2}$ since they cannot be received at A before the commit decision is made. We denote the times when an event at B can affect the outcome of t as the *influence zone* which is any time less than or equal to $c(t) - \frac{RTT(A,B)}{2}$.

Now consider the time duration that is neither in the awareness zone nor in the influence zone. Call this time duration the *critical zone*. Consider a transaction t' at B that requests to commit and commits in the critical zone. Transaction t' will not affect the outcome of t, since t' is not in the influence zone. Also, t will not affect t', since t' is not in the awareness zone. Assume that t' can successfully commit. However, t' can conflict with t. Since t' is not aware of t and t is, likewise, not aware of t', both transactions successfully commit. This potentially results in an inconsistency, a contradiction to the assumption that t' can successfully commit. This means that a transaction that starts at the beginning of the critical zone at B cannot commit with a commit latency smaller than the duration of the critical zone. This duration is equal to $RTT(A, B) - L(t)$.

Repeating the same steps above for each point in time at datacenter A will yield that the commit latency at A, L_A, is equal to $L(T)$ and the commit latency at any point in B, L_B, is larger than or equal to $RTT(A, B) - L_A$. Thus, the sum of L_A and L_B must be greater than or equal to $RTT(A, B)$. ■

Protocol	L_A	L_B	L_C	Average
Master/Slave (A master)	0	30	20	16.67
Master/Slave (C master)	20	40	0	20
Majority	20	30	20	23.33
Optimal (MAO)	5	25	15	15

Table 1: Possible commit latencies, L_A, L_B and L_C, for three datacenters with Round-Trip Times $RTT(A, B) = 30$, $RTT(A, C) = 20$, and $RTT(B, C) = 40$.

The previous lemma shows that there is a direct trade-off between the commit latencies of two datacenters. Given this lemma we are now able to judge whether a set of commit latencies are *achievable* or violates the lower-bound for scenarios with more than two datacenters by applying the lower-bound to each pair of datacenters.

Example. Consider an example of three datacenters, A, B, and C. The RTTs between the datacenters are: $RTT(A, B) = 30$, $RTT(A, C) = 20$, and $RTT(B, C) = 40$. Table 1 shows four achievable commit latencies and the average commit latency of the datacenters. The first two represent a master-slave replication approach, where a single master is responsible for committing transactions. In this approach, the master commits immediately, and the other datacenters commit latencies are the RTT to the master. Note how each pair of datacenters satisfies the lower-bound, *e.g.*, when A is the master $L_A + L_B = 30 = RTT(A, B)$. The third row represents a majority replication approach. For the case of three datacenters, the commit latency of a datacenter is the RTT to the nearest datacenter. These replication protocols experience different average commit latencies: 16.67, 20, and 23.33. However, the minimum average commit latency (MAO) that is achievable for this scenario is 15. The fourth row in the figure show the commit latencies, L_A, L_B, and L_C, that achieve an average commit latency of 15 while not violating the lower-bound.

Deriving the achievable minimum average commit latency for a given set of datacenters is outlined next.

3.3 Minimum average optimality

A MAO set of commit latency values minimizes the average commit latency of all datacenters without violating the lower-bound condition (Lemma 1) for any pair of datacenters. The MAO solution can be derived using the following linear programming formulation:

Problem 1. ***(Minimum Average Optimal)*** *The Minimum Average Optimal commit latencies for* n *datacenters is derived using a linear program with the following objective and constraints:*

$$\begin{array}{lll} \text{Minimize} & \sum_{A \in R} & L_A \\ \text{subject to} & \forall_{A,B \in R} & L_A + L_B \geq RTT(A, B) \\ \text{and} & \forall_{A \in R} & L_A \geq 0 \end{array}$$

where R is the set of datacenters. This formulation follows directly from Lemma 1. Minimizing the latency is our objective and the constraints are the correctness conditions that commit latencies are not negative and Lemma 1 is satisfied. We will use this methodology to derive the commit latency values used with the Helios commit protocol. This linear program can be adapted to other objectives. In Section A.2, we discuss optimizing for throughput.

4. HELIOS COMMIT PROTOCOL

In this section we propose Helios. The protocol design and operation are developed followed by a discussion on handling datacenter outages.

4.1 Helios architecture

System model. We consider a multi-datacenter system consisting of datacenters and clients. Each datacenter contains a *full copy of the data* and runs an instance of Helios, which is an optimistic concurrency control manager. Read operations are performed by clients first, and write operations are buffered. When the client is ready to commit, it sends a commit request containing its read and write-sets to the closest Helios instance. Helios replies to read requests with the current version and version timestamp of the requested data object. The version timestamp is the timestamp of the most recent write operation that wrote the data object. Blind writes are allowed, meaning that an object can exist in the write-set without being read. The commit request contains the read-set with the read version timestamps and the buffered write-set. Helios upon receiving the commit request will start the commit protocol to commit the transaction and propagate all updates to other datacenters. After committing the transaction, the commit decision is sent back to the client. *The time spent by the client from sending the commit request to receiving the commit decision is called the* ***commit latency***.

Communication. Helios uses a *log replication protocol* to exchange transaction information between datacenters. The log is continuously being propagated between datacenters. Each record in the log contains the information of either a *preparing transaction* that is trying to commit or a *finished transaction* that is either committed or aborted. Each transaction has two records in the log, one added when it starts as a preparing transaction and one record when it becomes a finished transaction. The transaction information includes the read and write sets. Each transaction is timestamped. The log is ordered according to transaction timestamps. Furthermore, *records are received by other datacenters according to their order in the log.* This means that receiving a record of a transaction with timestamp τ will follow all transactions with lower timestamps. Timestamps reflect the local clock of the datacenter. Clocks are loosely synchronized.

Helios conducts this replication using a replication protocol that is similar to Replicated Dictionary (RDict) [53]. RDict is an efficient protocol to replicate logs while maintaining their order. A $N \times N$ timetable, T_A, is maintained at each datacenter A where N is the number of datacenters. Each entry in the timetable is a timestamp representing a bound on how much a datacenter knows about another datacenter's records. For example, entry $\mathbf{T_A}[\mathbf{B}, \mathbf{C}] = \tau$ means that datacenter A knows that datacenter B is aware of all events at datacenter C up to timestamp τ. This notation will be used while describing Helios.

4.2 Helios overview

4.2.1 Intuition

To provide an intuition of the Helios commit protocol, consider the scenario in Figure 1. The figure shows the timeline of two datacenters, A and B. At A, a transaction t is issued at time $q(t)$ and committed at time $c(t)$. Transaction t commits immediately after receiving a log from B that is shown as an arrow going from B to A. This log carries transactions that were issued up to the time of sending the log, including transaction t' (assume t' is issued at the time of log transmission). The time the log was sent from B is $q(t')$. $q(t')$ is also the commit request time of t'. Helios receives the log in order, meaning that all transactions, preparing or finished, at B prior to or at time $q(t')$ are known to A at time $c(t)$.

Detecting conflicts. Transactions at B must not conflict with t. The approach to avoid conflicts is influenced by the way the lower-bound latency was developed in Section 3. However, here we do not make any assumptions regarding clock synchronization or communication. Rather, we rely on the exchanged logs and received transaction timestamps.

A transaction, t', at B is either issued during the influence zone, critical zone, or awareness zone. If t' starts during the influence zone, then transaction t will detect it because the log will contain a record of t'. If t' starts in the awareness zone, then it will detect t. Thus, for these two cases, conflicts will be detected. An undetected conflict can arise only if t' starts *and* commits within the critical zone. Thus, if t' is issued in the critical zone, Helios must ensure that it does not commit until it is in the awareness zone, which means that B will detect the conflict between t and t'.

Commit offsets. A *commit offset* is a time duration that represents the extent of knowledge needed by a transaction from other datacenters prior to committing. This duration ensures that a transaction t' that is issued in the critical zone will commit in the awareness zone (more on how to optimally assign these commit offsets in Section 4.5). Each datacenter maintains a commit offset for all "other" datacenters. For example, in the figure, A has a commit offset for B, denoted co_A^B, and B has a commit offset for A, denoted co_B^A.

Committing. When a transaction, t, requests to commit, it is assigned a timestamp, $q(t)$. Helios uses $q(t)$ and co_A^B to calculate a timestamp called the *knowledge timestamp* (*kts*). There is a *kts* value for every other datacenter B:

$$kts_t^B = q(t) + co_A^B \tag{1}$$

The commit condition for t is the following: *t can commit if its datacenter, A, knows about the transactions at every other datacenter B that were issued up to time kts_t^B at B.*

Consider Figure 1. When t is issued, Helios records that $kts_t^B = q(t) + co_A^B$. Assume that kts_t^B is equal to $q(t')$. Thus, A waits until it receives the log sent at $q(t')$ to commit. In the figure, t receives that log at time $c(t)$ and is then able to commit.

4.2.2 Condition on commit offset

In order to guarantee that a transaction t' that is issued in the critical zone will commit in the awareness zone, Helios needs to enforce a condition on the assignment of commit offsets.

Consider the scenario in Figure 1. Assume that the commit request time of t', $q(t')$, is in the beginning of the critical zone, as is shown in the figure. Transaction t', similar to t, is assigned a knowledge timestamp when it requests to commit. This value for t' is: $kts_{t'}^A = q(t') + co_B^A$. Thus, t' will know about all transactions at A up to time $kts_{t'}^A$ at A. For t' to be aware of t before committing, the value $kts_{t'}^A$ must be greater than or equal to the time t was issued, *i.e.*, $q(t)$. By expanding the value of $kts_{t'}^A$, the inequality to guarantee

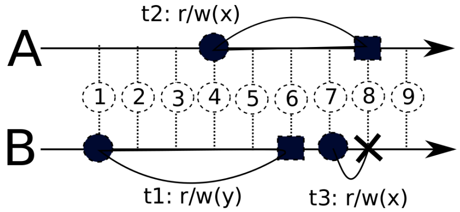

Figure 2: A scenario of Helios. Commit latencies are 3 and 5 for A and B. RTT is 8. A circle is a commit request, the square is a commit, and an X sign is an abort.

detecting conflicts is: $q(t') + co_B^A \geq q(t)$. $q(t)$ is equal to $q(t') - co_A^B$. Substituting this into the inequality and rearranging, the inequality becomes: $(co_A^B + co_B^A) \geq 0$. This is summarized by the following rule:

Rule 1. *For the Helios commit protocol to be able to detect conflicts, the sum of any two symmetric commit offsets (*e.g., $co_A^B + co_B^A$*) must be greater than or equal to 0.*

The requirement above specifies what is necessary to ensure correctness. However, a range of possible commit offset assignments might be used. We show later in Section 4.5 how Helios assigns commit offsets to minimize commit latencies. This assignment by Helios can theoretically achieve the lower-bound commit latency. Note that although timestamps are used, time synchronization is *not* required for correctness. Nonetheless, better synchronization will yield better performance as we demonstrate in Section A.

4.2.3 *Example scenarios*

To better illustrate how Helios works, consider the example in Figure 2 with two datacenters A and B. Time is denoted by the number between A and B's timelines. A commit request is denoted by a black circle and is connected to commit or abort time represented as a black square for a commit or an X sign for an abort. Transaction information is displayed as the read and write-sets. For example, transaction t_1 reads and writes y. Assume that the read version is the latest available version prior to requesting the commit. Log transmissions are omitted from the timeline, but assume for this example that they are being propagated at each tick of the clock. The RTT between A and B, $RTT(A, B)$, is equal to 8 time units. To simplify the presentation of the example, assume that the log takes exactly 4 time units to be delivered. Thus, at time 5, B knows about all events at A up to time 1.

For this example we pick the commit offset values to be -1 for co_A^B and +1 for co_B^A, hence $co_A^B + co_B^A$ is greater than or equal to 0. Now, follow the example scenario. Transaction t_1 reads and writes y and requests to commit at time 1, hence $q(t_1) = 1$. The knowledge timestamp for t_1 is given by: $kts_{t_1}^A = q(t_1) + co_B^A$ which is equal to 2. The transaction waits until B receives the log sent from A at time 2, which is the value for $kts_{t_1}^A$. At time 6, t_1 successfully commits after receiving the log from A that was sent at time 2 (the log takes 4 time units to be received). t_2, which reads and writes x, requests to commit at A at time 4. The knowledge timestamp for t_2 is given by: $kts_{t_2}^B = q(t_2) + co_A^B$ which is equal to 3. Assume that the log transmission from B at time 3 took more time than usual and was received at A at time 8, one time unit late. For this reason, t_2 commits at time 8 when the history of datacenter B is received up to time 3,

Algorithm 1: Processing commit requests at A

1: t := local transaction requesting to commit at A
2: **if** t conflicts with any $t' \in$ PTPool $\cup$ EPTPool **then**
3:     Abort t; exit
4: **for each** object o in $t.readset$ **do**
5:     **if** o is overwritten **then**
6:         abort t; exit
7: $t.timestamp = get\_time()$
8: **for each** datacenter $X \in datacenters$ **do**
9:     $t.kts^X = t.timestamp + co_A^X$
10: PTPool.$append(t)$; $Log_A.append(t)$

Algorithm 2: Processing transactions in the log at A

for each transaction t *in* Log_A **do**
    if t is local **then**
        skip to next
    if t conflicts with any t' $\in$ PTPool **then**
        abort t'; $t'.timestamp = get\_time()$
        $Log_A.append(t')$
    if $t.type ==$ preparing **then**
        EPTPool.$append(t)$
    else // $t.type ==$ finished
        if t.committed==true **then**
            for each object o in $t.writeset$ **do**
                Apply o to data store
        Remove t from EPTPool
    $T_A[A, host(t)] = t.timestamp$

which is the value of $kts_{t_2}^B$. Now consider t_3. It requests to commit at B at time 7. The knowledge timestamp value is given by: $kts_{t_3}^A = q(t_3) + co_B^A$, which is equal to 8. However, one time unit after its request, it receives t_2's information that was sent at time 4 from A. A conflict is detected and t_3 aborts immediately.

4.3 Concurrency control protocol

In this section we discuss the design of Helios. The main tasks performed are: (1) process commit requests (Algorithm 1), (2) process remote transactions received in the shared log (Algorithm 2), and (3) commit preparing transactions (Algorithm 3). We also briefly discuss read-only transactions in Section B.

4.3.1 *Commit requests*

When Helios receives a commit request for a transaction t at datacenter A, it checks whether it conflicts with preparing transactions (Lines 2-3). Preparing transactions are maintained in the *Preparing Transactions Pool* (PTPool) for local transactions and the *External Preparing Transactions Pool* (EPTPool) for remote transactions. Transaction t aborts if a conflict exists, which is an intersection between the read or write-set of t with the write-set of any preparing transaction. Then, the read-set of t is verified to have not been overwritten (Lines 4-6). If no conflicts are detected and the read-set is not overwritten, the knowledge timestamps are calculated as defined in Equation 1 (Lines 7-9). A preparing record of t is appended to both the PTPool and the local log, Log_A (Line 10). Appending to the log includes adding the record and updating the timetable so that $T_A[A, A]$ equals to t's timestamp.

4.3.2 *Log processing*

Helios processes transactions in the log in order (Algorithm 2). Local transaction records are not processed

Algorithm 3: Committing preparing transactions at A

1: **for each** transaction t *in* PTPool **do**
2:     **for each** datacenter $X \in datacenters$ **do**
3:         **if** $T_A[A, X] < t.kts^X$ **then**
4:             skip to next transaction
5:     Apply t.write-set to local data store at A
6:     commit t; $t.timestamp = get\_time()$;
7:     $Log_A.append(t)$

(Lines 2-3). For a coming transaction t, conflicts are detected with local preparing transactions in PTPool. A conflict exists if the read or write set of a local preparing transaction, t', intersects with the write-set of t (Lines 4-6). A conflicting transaction in PTPool is aborted by changing its state to aborted and updating its timestamp. An abort record is added to the log (Lines 5-6). Remember that adding to the log includes updating the timetable to reflect the addition of a new transaction record.

Remote transactions are either preparing or finished. When a preparing transaction is received from another datacenter, it is added to EPTPool (Lines 7-8). A finished transaction contains a flag to indicate whether it has committed or aborted. If it is committed, then the write operations in the write-set are applied to the local data store (Lines 11-12). However, whether a finished transaction is committed or aborted, it is removed from the EPTPool (Line 13). Finally, the timetable is updated to reflect that t is processed (Line 14).

4.3.3 Committing preparing transactions.

A preparing transaction, t, can successfully commit by satisfying two conditions: (1) *External knowledge:* Helios must have processed transactions from other datacenters up to the *knowledge timestamp* (*kts*) calculated according to Equation 1. (2) *Conflict freedom:* no conflicts were observed with t up to the point when the first condition is satisfied. Algorithm 3 checks whether the external knowledge condition is satisfied for transactions in PTPool (Lines 2-4). If the condition is satisfied for a transaction t, then it can successfully commit. The write-set of t is applied and a record is added to the log (Lines 5-7). Conflicts are already detected when the transaction requested to commit (Algorithm 1) and while the log is being processed (Algorithm 2). Thus, there is no need to detect conflicts at this point.

Commit condition. The commit condition can be summarized as the following:

Rule 2. *A transaction t in A commits if no conflicts are detected, and*

$$T_A[A, B] \geq kts_t^B, \quad \forall_B (B \in R)$$

where R is the set of other datacenters and kts is the knowledge timestamp defined in Equation 1.

4.4 Liveness

Intuition. Helios needs information from other datacenters to be able to commit its preparing transactions (see Rule 2). An outage of a datacenter will cause other datacenters to block waiting for its transaction log. The blocking will continue until the datacenter is back up again and Helios is recovered. This is similar to blocking scenarios in 2PC. State machine replication (SMR) is used to overcome these blocking scenarios. Here, we present the way Helios achieves liveness while enabling the flexibility to set the number of tolerated datacenter outages. Thus, Helios enables controlling the trade-off between liveness and performance.

The main idea is to ensure that a transaction, t, at datacenter A does not commit before its information exists at f other datacenters, where f is the number of datacenter outages to be tolerated. Verifying that t's information exists at another datacenter, B, can be done by waiting for an acknowledgment of the receipt of t. B should keep the information about t until it is completed and its information are propagated to other datacenters. This is important to enable B to propagate the information about t in case A experiences an outage.

Committing. A datacenter can know whether another datacenter received the record of a transaction by examining the transaction's timestamp and comparing it to the extent of the other datacenter's knowledge using RDict (see Section 4.1). If the number of datacenter outages that are to be tolerated is f out of n total datacenters, then the transaction waits for its information to be received by f other datacenters.

Failure case. When a datacenter B fails, a datacenter A might have to delay the commitment of a transaction until it can ascertain the identity of all previously finished transactions at the failed datacenter B by communicating with other datacenters, C, where backup information for B exists. However, it is not always possible to conclude that a datacenter has failed since a network partition might render the messages sent from B undelivered. Consider the following scenario: a datacenter A is waiting for information from B. A network partition makes information from B unable to be delivered to other datacenters. Given that no information is received at A from B, datacenter A consults C for information about B's finished transactions. Datacenter A can commit transactions since it knows that B cannot commit any transactions without getting an acknowledgment of its receipt from either B or C.

Grace time. The subtlety here is about knowing when a datacenter C can be certain about the state of another datacenter (B). Helios adopts a *time-based invalidation* technique to enable a datacenter to ascertain the state of a failed datacenter. Helios makes the commitment of a transaction dependent on the reception time at the other datacenters. We call this time the *Grace Time* (GT). A datacenter C will acknowledge the receipt of transaction t from B if and only if the transaction information is received at C at time τ that is smaller than the commit request timestamp of t plus GT, *i.e.*, $\tau \leq (q(t) + GT)$. Otherwise, C will not acknowledge the reception of t.

The implication of this bounded acknowledgment is that at time τ at datacenter C, we have a guarantee that no transactions with a request timestamp less than (τ-GT) will be able to synchronously replicate to C. Thus, B and C will be able to infer information about A. Specifically, B and C will be confident that no transactions (unknown to them) will commit at A with a timestamp less than $min\{now_B, now_C\} - GT$, where now_X is the current time at datacenter X. This is because even if a transaction with an earlier timestamp existed at A it will be invalidated since it was not acknowledged by B and C. This is very useful information for Helios because it allows transactions at B and C to commit even if A fails or cannot communicate with other datacenters.

Integration. This acknowledgment mechanism can be easily incorporated in Helios by extending commitment Rule 2 to reflect the acknowledgment requirement and invalidation. First, the extent of A's knowledge of B's events, $T_A[A,B]$, can be inferred by other participants. Thus, when $T_A[A,B]$ is used in Rule 2 it can be substituted by:

$$\hat{T}_A[A,B] = max\{T_A[A,B], \eta\} \tag{2}$$

where η is the extent of knowledge of A about B that can be inferred using a set of any $n-f$ other datacenters called κ. η is calculated as the following:

$$\eta = min\{T_A[C,C]\}\forall_{C\in\kappa} - GT \tag{3}$$

The other needed extension to Rule 2 is to restrict the commitment of a transaction until it has been successfully acknowledged by a number of datacenters equal to the number of outages to be overcome. A transaction t at A is considered acknowledged by B if $T_A[B,A]$ is greater than or equal to t's timestamp: $q(t)$. Thus, what is needed is to ensure that the transaction was acknowledged by a set of $f+1$ total datacenters. However, remember that for a transaction to be successfully acknowledged it needs to be received by the other datacenter in a bounded time, hence $q(t)+GT$. This is ensured by examining $ts_C(t)$, which is the time the transaction record of t was received at C. Given the aforementioned extensions, the commitment rule can now be written as the following:

RULE 3. *A pending transaction t in A commits if no conflicts are detected, and*
(1) $\hat{T}_A[A,B] \geq kts_t^B, \quad \forall_B(B \in R)$
(2) $T_A[C,A] \geq q(t), \quad \forall_C(\ C \in \kappa')$
(3) $ts_C(t) < q(t) + GT, \quad \forall_C(\ C \in \kappa')$

where κ' is a set of any $n-f$ datacenters where n is the number of datacenters and f is the number of tolerated outages.

Grace time. The choice of the value of GT is controlled by the trade-off between minimizing the effect of datacenter outages on progress and minimizing the number of aborted transactions due to delays in the acknowledgment process. A datacenter outage increases the latency because rather than waiting for the knowledge timestamp (Equation 1) from the failed datacenter, a datacenter has to wait for an additional duration of GT, to accumulate knowledge of the transactions in the failed datacenter from all the running ones. Also, having a small GT might lead to some transactions unnecessarily aborting due to message delays or drops while waiting for the acknowledgment.

4.5 Commit offsets assignment

The assignment of commit offsets directly influences the commit latency of each datacenter. It is not straightforward to assign commit offsets to minimize commit latencies. However, it is possible to estimate commit latencies given the assigned commit offsets. To simplify this estimation we assume that clocks are synchronized and that messages take exactly $\frac{RTT(A,B)}{2}$ time to be sent from A to B. This is of course not true of real systems and will introduce an estimation error of the commit latency.

Estimating commit latency. A transaction t at A waits for information from other datacenters. For every other datacenter B, t can commit only after receiving information from B up to time $(q(t) + co_A^B)$. This information is received at A at time $(q(t) + co_A^B + \frac{RTT(A,B)}{2})$. Since, t commits after receiving information from all datacenters, the commit latency of t is delayed until the time it receives the information from the last datacenter; if that datacenter is C, then the commit latency of transactions at datacenter A is

	V	O	C	I	S
V	-	66 (11)	78 (10)	84 (9)	268 (7)
O	66 (10)	-	19 (1)	175 (7)	210 (4.4)
C	78 (9)	19 (1)	-	175 (7)	182 (6)
I	84 (8)	175 (7)	175 (6)	-	194 (4)
S	268 (6)	210 (4)	182 (6)	194 (4)	-

Table 2: RTT latencies between different datacenters in milliseconds and the standard deviation inside parentheses.

$$L_A = co_A^C + \frac{RTT(A,C)}{2} \tag{4}$$

Setting commit offsets. Helios can assign arbitrary values to commit offsets, thus targeting the desired commit latency as long as they do not violate the commit offset correctness requirement (Rule 1). The linear program in Section 3.3 can be used to derive the commit latency values that will minimize the commit latency. Thus, we can set the commit offsets by the following rearrangement of Equation 4

$$co_A^C = L_A - \frac{RTT(A,C)}{2} \tag{5}$$

where L_A is the target commit latency calculated using the linear program in Section 3.3 and $RTT(A,C)$ is an estimation of the RTT.

Correctness. The optimal assignment is guaranteed to satisfy the correctness requirement of commit offsets shown in Rule 1. This can be verified by substituting Equation 5 into the sum requirement (Rule 1) that $(co_A^C + co_C^A)$ must be greater than or equal to 0. This will yield to the requirement that ($L_A + L_C$ - $RTT(A,C)$) must be greater than 0. Since the linear program of Section 3.3 has a constraint that the sum of two commit latencies must be greater than the RTT between them, this yields that ($L_A + L_C$ - $RTT(A,C)$) is always greater than 0. Thus, the assignment guarantees detection of conflicts.

Estimation vs. reality. In a real deployment the assignment in equation 5 will not yield the exact lower-bound commit latency because of communication links variability and lack of perfect synchronization. We quantify some of these effects in Section A and show in the evaluation (Section 5) that a real-life deployment is able to achieve a commit latency close to the optimal.

5. EVALUATION

5.1 Evaluation framework

The evaluation of Helios is performed using Amazon's AWS. Helios is evaluated for three different liveness levels: Helios-0 tolerating no datacenter outages, Helios-1 tolerating a single outage, and Helios-2 tolerating two outages. We compare the performance of Helios with Message Futures [40], Replicated Commit [38], and an implementation of Two-Phase Commit (2PC) over Paxos (2PC/Paxos) that is inspired from Spanner [16]. The calculated lower-bound commit latency is also shown for reference.

Objective. The evaluation focuses on quantifying and testing the performance of Helios in addition to its resilience to a lack of strict clock synchronization and RTT estimation errors. The performance metrics we report are commit latency and throughput. These are calculated for transactions that successfully commit. Helios resiliency is tested by artificially reducing the level of synchronization and accuracy of RTT estimation.

Highlights. The highlights of the results of this evaluation are the following:

- Helios-0 achieves a commit latency that is within 54ms from the calculated lower-bound commit latency. This overhead increases as the level of liveness increases (Figure 3).
- All variants of Helios achieve lower commit latency and higher throughput compared to Message Futures and 2PC/Paxos. Replicated Commit achieves a lower throughput when compared to Helios but it experiences a commit latency within what Helios-0 and Helios-2 achieve (Figures 3 and 4).
- Inaccurate synchronization and RTT estimation lead to a higher commit latency (Figure 5). The overhead recorded for this set of experiments is up to 52% higher average commit latency.

Setup. Amazon Elastic Compute Cloud (EC2) machines used are High-CPU Extra Large (c3.x2large) which have eight CPU cores and 7GB of RAM memory. Five datacenters were used in the following locations: California (C), Virginia (V), Oregon (O), Ireland (I), and Singapore (S). The average RTT latencies observed are shown in Table 2. These RTT numbers are sampled over 24 hours. The standard deviations of the samples are shown inside parentheses. Each datacenter hosts a full replica of the data. One machine in each datacenter runs Helios, serving transactions issued by clients. Helios uses HBase [25] as the underlying data store. Clock synchronization is achieved by running the command "ntpdate ntp.ubuntu.com" before each experiment run.

Workload. Dedicated machines in each datacenter are used to simulate clients and evaluate Helios. We use Transactional YCSB (T-YCSB) [17], a multi-record transactional benchmarking framework, for our evaluations. T-YCSB, an extended version of YCSB [14], generates transactional workloads. It issues transactions that consist of a set of read and write operations, where each operation accesses a different record of the data store. Each client can have one outstanding transaction at a time. Clients issue transactions as fast as they can unless mentioned otherwise. An operation is either a read or a write to a key from a pool of 50000 keys. The key is chosen using a Zipfian distribution. Each transaction contains five operations. Half of these operations are reads and the other half are writes. Each experiment runs for a duration of 10 minutes.

Configuration. Helios assigns the commit offsets to target achieving the lower-bound commit latency. The RTT average values in Table 2 are used to derive the optimal commit latencies using the method shown in Section 3.3. The calculated optimal commit latencies are: 69ms, 10ms, 10ms, 166ms, and 200ms for V, O, C, I, and S, respectively. These are then used to derive commit offsets to be used to commit transactions in Helios using Equation 5. As a baseline, we also display results of Helios without performing RTT estimations and optimal latency calculations by setting all the commit offsets to 0. This baseline is called **Helios-B**.

5.2 Systems for comparison

We report Helios results compared with results obtained from Message Futures [40], Replicated Commit [38], and a Two-Phase Commit over Paxos protocol (2PC/Paxos) that is inspired by Spanner [16].

Message Futures, like Helios, uses replicated logs that are causally ordered to exchange transactions information and detect conflicts. In Message Futures, partial logs are continuously being propagated between datacenters [53]. Transactions, t^i, which request to commit at A between the transmissions of partial log i and $i+1$ are assigned a reservation number i. Partial log transmissions are acknowledged by other datacenters. The acknowledgment contains a log of all transactions (not known to A) up to the time, τ, of the transmission of the acknowledgment. Transactions t^i can commit when log i is acknowledged by all other datacenters and no conflicts are detected. Helios follows a different approach where a transaction commits by waiting for partial logs from other datacenters up to calculated timestamps that minimizes the commit latency while preserving correctness.

Replicated Commit uses Paxos for cross datacenter replication and Two-Phase Locking (2PL) to avoid conflicts. Clients first perform read operations by trying to lock the read object in a majority of datacenters. Write operations are buffered. Then, committing the transaction is done by replicating it using Paxos to all datacenters. As the transaction is received by datacenters, locks are acquired for buffered write operations and read locks are validated. If the locks were acquired and validated at a majority of datacenters, then the client will commit the transaction. Thus, its commit latency should be in the order of a single RTT to the closest majority, which is the time required by Helios-2 to commit. However, Replicated Commit performs poorly in terms of throughput due to the read strategy. The client spends more time reading before the commit request is issued.

2PC/Paxos uses 2PC to avoid conflicts and Paxos for replication. In this evaluation the datacenter in Virginia (V) is assigned to be the 2PC coordinator. Clients, scattered across all five datacenters issue reads to the coordinator. The coordinator maintains a lock table. When a read is received, a read lock is placed on the corresponding key. Write operations are buffered. When a transaction is ready, a request to commit is sent to the coordinator with information of all operations. Once the coordinator receives the commit request it tries to acquire the write locks and verifies that the read locks are still held. If successful, the transaction commits. Then, the coordinator replicates the log to a majority of datacenters using Paxos [30]. The coordinator is assumed to have a lease so that it will not need to go through the leader election phase, hence reducing the required time to replicate the log by one RTT. After replicating the log, the output of the transaction is sent to the client. Transactions that are detected to be involved in a deadlock are immediately aborted.

Replicated Commit and 2PC/Paxos replicate to a majority before committing a transaction. Thus, in the setup used for the following experiments with five datacenters, they tolerate two datacenter failures. Thus, they tolerate the same number of failures as Helios-2. Message Futures does not inherently overcome datacenter outages, thus its liveness guarantee is equivalent to Helios-0.

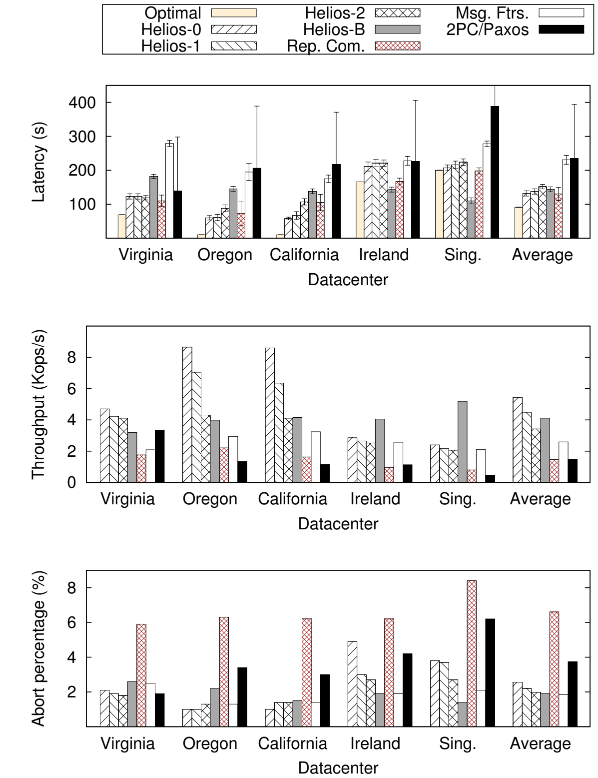

Figure 3: The commit latency, throughput, and abort rate of a scenario with 60 clients and 5 datacenters.

5.3 Helios performance

In the following experiments, 60 clients scattered across all datacenters issue back-to-back transactions to Helios instances. We report the results for each datacenter separately in addition to confidence intervals for the commit latency numbers.

Latency. The commit latency results are shown in Figure 3(a) for Helios with varying liveness levels. We also show the commit latency of the baseline systems. In the figure, the average latency of clients that are running at each datacenter is shown. Helios-0 has the closest latencies to the optimal since it does not need to synchronously replicate any transactions before committing them. The overhead of Helios-0 over the optimal commit latency is within 7 to 54ms. This overhead is caused by different factors such as execution time, I/O access, network variability, and the network latency between the client and server. Some of these factors are analyzed in Section A.

Increasing the level of fault-tolerance results in an increase in commit latency (see Section A.2 for more details). This increase is caused by the difference between the commit latency needed to commit consistently and the required time to get the acknowledgment of the transaction's reception by the number of tolerated datacenter failures. This difference can vary, which explains why the overhead of tolerating one datacenter failure by using Helios-1 to Helios-0 ranges from 0-1ms for Virginia and Oregon while the overhead is 9-10ms for the remaining datacenters. Virginia and Oregon do not experience a significant overhead because their commit latencies are equal to or greater than the RTT to the closest datacenter, thus receiving the acknowledgment of the replication happens while waiting for transaction commitment. Similar behavior is observed when increasing the number of tolerated datacenter failures from 1 to 2, which results in an increase of commit latency ranging from no observed overhead for Virginia, Ireland, and Singapore to 27ms and 40ms overhead for Oregon and California. The average commit latency for Helios-2 at Virginia is actually 4ms lower than the latency of Helios-1. This does not mean that less time is required to commit for Helios-2, rather it is due to the variability of the compute and network conditions.

Replicated Commit commits transactions with a latency equal to the closest RTT majority. The achieved average latency is 20ms lower than the one achieved by Helios-2. This is because Helios experiences more compute overhead to process and send the log, while Replicated Commit's communication process is lightweight. The average commit latency of Replicated Commit is also close to Helios-0. The evaluation topology and RTT values for this set of experiments make the overhead for tolerating two datacenter outages not so large. Coupled with the higher compute overhead of Helios, this results in observing similar commit latency for Helios-0 and Replicated Commit. However, in topologies where the overhead to tolerate two datacenter outages is higher, Helios-0 should show a more significant advantage over Replicated Commit.

Message Futures requires roughly a RTT to all other datacenters to commit transactions. This causes an overhead compared to Helios-0 that ranges from 17ms for Ireland to 181ms for Singapore. The average overhead is 99ms. 2PC/Paxos requires more time to commit when compared to Helios-2, which has the same degree of fault tolerance. The difference in commit latency between 2PC/Paxos and Helios-2 ranges from 15-17ms for Virginia, Ireland, and Singapore to 146-159ms for Oregon and California. In 2PC/Paxos, a transaction experiences a latency equal to the time required to send the transaction to Virginia in addition to a RTT to a majority from Virginia. This gives an advantage to Virginia and datacenters close to it. Helios-B, which is Helios-0 but without assigning optimal commit latencies, has an average overhead of 12.2ms compared to Helios-0.

Helios and Message Futures are stable with standard deviation values that are less than 10. Replicated Commit and 2PC/Paxos on the other hand show unstable performance with standard deviation values of up to 35 for Replicated Commit and values ranging from 154 to 278 for 2PC/Paxos. The stability observed for Helios and Message Futures is due to the use of a log to exchange transactions between datacenters. If a transmission i was affected by the variability of the communication network, the next transmission, $i+1$, will include i because it is part of the log that was not acknowledged [53].

Throughput. The throughput results are presented in Figure 3(b). Helios-2 achieves a throughput 37% lower than what is achieved by Helios-0. The throughput achieved by Replicated Commit, Message Futures, and 2PC/Paxos is lower than what is achieved by Helios. The commit latency causes this lower throughput for Message Futures. Replicated Commit and 2PC/Paxos have a larger overhead due to their read strategy. Replicated Commit reads from a majority and 2PC/Paxos directs reads to Virginia which increases the amount of time spent by a client prior to requesting to commit. Remember that the commit latency is the time from the client's commit request till a decision is received and does not include the read latency incurred prior to the commit request. Replicated Commit and 2PC/Paxos achieve an average throughput that is 56-57% lower than Helios-2. Note that Virginia in 2PC/Paxos achieves the closest throughput to Helios-2 although it was not the closest to Helios in terms of commit latency. This illustrates the advantage of 2PC/Paxos clients at the master datacenter ridding them from the overhead of wide-area reads. Message Futures's throughput is 52% lower than the throughput of Helios-0. For Message Futures, the achieved throughput by each datacenter compared to Helios is directly correlated to the overhead in commit latency. This is because Message Futures, like Helios, reads from the closest datacenter. Helios-B achieves a throughput that is 24% lower than Helios-0. Like Message Futures, the throughput overhead correlates directly with the latency overhead.

Contention. The abort rates are shown in Figure 3(c). The abort rate is a product of many factors, such as the amount of contention, the number of concurrent transactions, the lifetime of the transaction, among others. However, we can observe a pattern in the average abort rates of different systems. Increasing the liveness level of Helios causes the abort rate to decrease, which is a sign of less contention that results from the decrease in throughput. Message Futures has the lowest abort rate although it has a higher commit latency compared to Helios. Replicated commit and 2PC/Paxos on the other hand achieve worse abort rates. 2PC/Paxos has the highest commit latency and additionally holds locks for an extended period of time, beginning from the read operations prior to the commit request until releasing locks after the commit phase. This increases contention and is a factor causing this high abort rate. Replicated Commit locks data object in lock tables scattered across datacenters which increases the chance of experiencing deadlocks. Abort rates of individual datacenters illuminate how having a larger commit latency is a disadvantage causing more transactions to be aborted; Singapore experiences larger abort rates when compared to other datacenters.

Peak throughput. Now, we measure the peak achievable throughput and the number of clients required to converge to that throughput for Helios. We increase the load on the system by gradually increasing the number of clients issuing transactions. In Figure 4(a), the cumulative achievable throughput is plotted with the number of clients in the system increasing from 15 clients to up to 285 clients in 30-clients increments. Note how the Helios protocols converge to a throughput between 6000 and 7000 operations per second. Helios-0 and Helios-1 are the fastest to converge as soon as the number of clients is 195. Helios-2 and Helios-B converge with 255 clients. Message Futures takes a slower pace than Helios. This is mainly due to its higher commit latency values. 2PC/Paxos is not able to achieve a throughput that is larger than 1700 operations per second. The demand placed on the coordinator causes thrashing as soon as the number of clients reaches 195 clients. Replicated Commit experiences a similar throughput to 2PC/Paxos but does not thrash.

Converging to the peak throughput signals an I/O bottleneck. Increasing the demand (number of clients) past the convergence point stresses the bottleneck, causing a gradual degradation of the performance of individual transactions. Figure 4(b) shows the effect of increasing the number of clients on the commit latency. Helios variants maintain their commit latency up to the convergence point between 195 and 255 clients. Message Futures and Replicated Commit maintain their commit latency because they need more clients than what is shown for them to converge. 2PC/Paxos starts a gradual increase in commit latency from the beginning signaling a stress on the system even with a small number of clients. Placing a large demand on a single coordinator causes resources to be exhausted rapidly, leading to this observation on 2PC/Paxos.

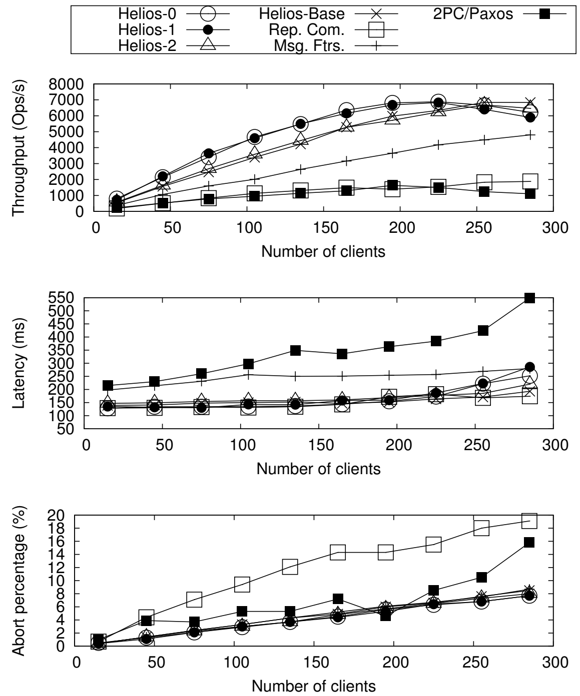

Figure 4: The throughput, average latency, and abort rate as the number of clients is increased.

Increasing the number of clients increases the contention and leads to more aborts. Figure 4(c) shows this effect. All protocols except Replicated Commit and 2PC/Paxos have similar abort rates that are increasing around 0.7% for every 30 clients added. 2PC/Paxos abort rate increases significantly for 258 clients to have 15.8% aborts, while Replicated Commit experiences close to 20% aborts for 258 clients.

5.4 Synchronization and estimation errors

Synchronization. The performance of the Helios protocols relies heavily on the level of synchronization. Using the readily available NTP clients provided by Ubuntu for the previous experiments shows that the available synchronization tools enable Helios to achieve good performance. However, we would like to get an insight on Helios performance with more hostile environments with clocks that are not synchronized. Now we will show results from a set of experiments while changing the clock readings of machines to emulate a lack of synchronization. In these experiments Helios-0 is used. The results are shown in the leftmost four groups of results in Figure 5(a). In the figure we display

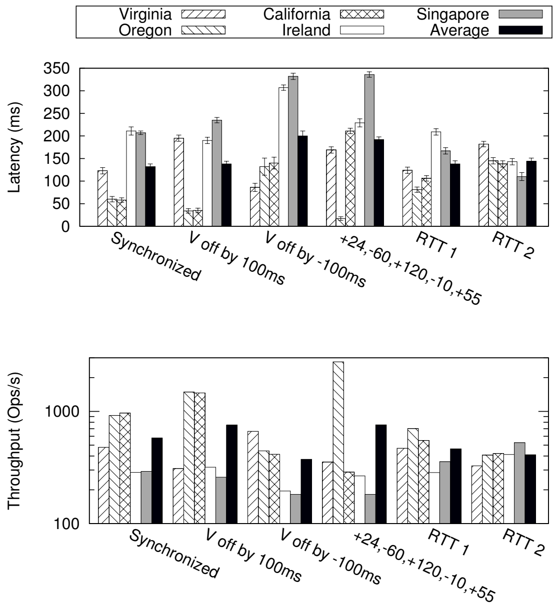

Figure 5: The effect of the lack of synchronization on the performance of Helios-0 in the leftmost four groups of results and the effect of erroneous RTT estimation in the rightmost two groups of results.

three experiments and compare them with the results from a synchronized run using NTP. The first non-synchronized scenario is to set the clock at Virginia ahead of the others by 100ms. Note how this resulted in its commit latency to increase by 62ms. Other datacenters commit latencies improved except for Singapore. This improvement is caused by setting Virginia's clock ahead in time, it puts it in a disadvantage compared to other datacenters which leads to a better commit latency for others. The second scenario has Virginia's clock 100ms behind other clocks. This change puts Virginia in an advantage that is shown by a decrease in commit latency of 37ms. However, other datacenters consequently achieve a higher commit latency and cause an increase in the average commit latency by 64ms compared to the synchronized case. In the final scenario we introduce random errors to all datacenters. The errors are presented by a sequence of numbers denoting the shift in time for the ordered set of datacenters in milliseconds: V, O, C, I, and S. Consider the case {+24,-60,+120,-10,+55}. We find that for this case the average commit latency is 60ms higher than the synchronized case. However, for some individual datacenters, this caused a significant improvement of the commit latency; California, for example, achieves a commit latency 23ms lower than California in the synchronized case.

The results demonstrate that small to medium clock skews introduce tolerable increases in the average commit latency of up to 64ms for the scenarios we considered. These increases are correlated with the level of synchronization.

RTT estimation errors. A vital part of Helios is the assignment of optimal commit latencies to datacenters. However, deriving these values requires estimating the RTT between datacenters. RTTs can be variable and change occasionally. Thus, the calculated assigned commit latencies suffer from the potential of being inaccurate. Here we perform tests to quantify the effect of non-accurate estimations of the RTT on the commit latency of Helios. Helios-0 will also be used for these experiments. We will experiment with two cases where high margins of error are introduced to our RTT estimations. The results are shown in Figure 5(a) labeled RTT estimation 1 and 2. The first RTT estimation is calculated by introducing an error to the correct estimates by increasing one fifth of the RTTs by 25ms, another fifth by 75ms, decrease a fifth of RTTs by 25ms, and decrease yet another fifth by 75ms, and the remaining RTTs are not changed. This scenario leads to the average latency being 4.5% higher that the one achieved by the original estimate. The second RTT estimate is of an erroneous estimation of no latency between datacenters. This will lead to assigning all datacenters a commit latency of 0. With this input, Helios-0 observes the following commit latencies for V, O, C, I, and S: 182ms, 145ms, 138ms, 143ms, 110ms, averaging to 144ms. This average is 9% higher than the average commit latency achieved by the original estimate.

These results illustrate that even with highly erroneous RTT estimations, Helios-0 observes a slight increase in the achieved commit latencies by 4.5% and 9% for the introduced errors. Helios shows the same stability observed in previous experiments where standard deviation values are less than 10.

Throughput. When errors are introduced to clock synchronization and RTT estimation, the commit latency is affected for different datacenters. Some datacenters have their commit latency increase while for others their commit latency decrease. The commit latency of a datacenter is a major factor of the observed throughput. Thus, we expect different datacenter throughputs to be affected in correlation with the commit latency. We present the achieved throughput results in Figure 5(b). For the scenario when Virginia's clock is 100ms ahead of others, notice how the throughput of Oregon and California increase as a result of the decrease in their commit latency. Likewise, the throughput of Virginia decreases in correlation to the commit latency's increase. This effect is observed throughout the set of experiments. However, an interesting observation is that for some scenarios, this leads to the average throughput to become larger than the synchronized case. For the *+24,-60,+120,-10,+55* case, for example, throughput is 31% higher than the synchronized case. This is even though the average commit latency of *+24,-60,+120,-10,+55* is higher than the original case. The increase in throughput is due to the extremely low resulting latency of Oregon allowing it to achieve a throughput of 2764 operations/s that caused the average throughput to exceed that of the synchronized case. We provide more discussion on this trade-off in Section A.2.

6. CONCLUSION

In this paper we have formulated a lower-bound on commit latency of transactions running on replicated data stores. We also designed an optimistic commit protocol, called Helios, using insights from the lower-bound. Helios allows tunable performance for individual datacenters. In the paper we demonstrate tuning Helios to minimize the average commit latency. Helios separates consistency from liveness guarantees. This allows the flexibility of setting the number of tolerated datacenter outages. Helios is compared experimentally to three systems: Replicated Commit, Message Futures, and 2PC/Paxos.

7. ACKNOWLEDGEMENTS

This work is supported by NSF grant IIS 1018637 and is partially funded by a gift grant from NEC Labs America. Faisal Nawab is partially funded by a scholarship from King Fahd University of Petroleum and Minerals. We would also like to thank Amaozn for access to Amazon EC2.

8. REFERENCES

[1] Summary of the Amazon EC2 and Amazon RDS service disruption in the US East Region. `http://aws.amazon.com/message/65648/`, 2011. [Online; acc. 5-Oct-2011].

[2] Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region. `http://aws.amazon.com/message/680587/`, 2012. [Online; acc. 10-May-2013].

[3] 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.

[4] A. Adya and B. Liskov. Lazy consistency using loosely synchronized clocks. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 73–82. ACM, 1997.

[5] Apache Cassandra, 2011. [Online; acc. 5-Oct-2011].

[6] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3), 2014.

[7] P. Bailis and A. Ghodsi. Eventual Consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3):20:20–20:32, Mar. 2013.

[8] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD*, 2013.

[9] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conf. Innovative Data Systems Research*, pages 223–234, 2011.

[10] N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *NSDI*, volume 6, pages 5–5, 2006.

[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[12] P. Bernstein, C. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *Proc. of CIDR*, pages 9–20, 2011.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[14] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. 1st ACM Symp. Cloud Computing*, pages 143–154. ACM, 2010.

[15] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[16] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally-distributed database. *in Proceedings of OSDI*, page 1, 2012.

[17] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endowment*, 4(8):494–505, 2011.

[18] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 715–726. VLDB Endowment, 2006.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symp. Operating Systems Principles*, pages 205–220, 2007.

[20] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 11. ACM, 2013.

[21] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 173–184. IEEE, 2013.

[22] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[23] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, New York, NY, USA, 2011. ACM.

[24] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM, 2011.

[25] HBase. `http://hbase.apache.org`, 2011. [Online; acc. 18-Jul-2011].

[26] B. Kemme, R. Jimenez-Peris, and M. Patino-Martinez. Database replication. *Synthesis Lectures on Data Management*, 2(1):1–153, 2010.

[27] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[28] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

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

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

[31] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[32] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[33] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430. ACM, 2005.

[34] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Enhancing edge computing with database replication. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 45–54. IEEE, 2007.

[35] B. Liskov. Practical uses of synchronized clocks in distributed systems. *Distributed Computing*, 6(4):211–219, 1993.

[36] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[37] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Symposium on Networked Systems Design and Implementation*, 2013.

[38] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commits. *Proceedings of the VLDB Endowment*, 6(9), 2013.

[39] R. Miller. Massive Flooding Damages Several NYC Data Centers. `http://www.datacenterknowledge.com/archives/2012/10/30/major-flooding-nyc-data-centers/`, 10 2012. [Online; acc. 10-May-2013].

[40] F. Nawab, D. Agrawal, and A. El Abbadi. Message futures: Fast commitment of transactions in multi-datacenter environments. In *Conf. Innovative Data Systems Research*, 2013.

[41] F. Nawab, V. Arora, D. Agrawal, and A. E. Abbadi. Chariots : A scalable shared log for data management in multi-datacenter cloud environments. In *EDBT*, 2015.

[42] D. Obasanjo. When databases lie: Consistency vs. availability in distributed systems, 2009.

[43] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[44] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. E. Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*, 5(11):1459–1470, 2012.

[45] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. *Flexible update propagation for weakly consistent replication*, volume 31. ACM, 1997.

[46] E. Schurman and J. Brutlag. Performance related changes and their user impact. In *velocity web performance and operations conference*, 2009.

[47] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[48] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

[49] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. *Managing update conflicts in Bayou, a weakly connected replicated storage system*, volume 29. ACM, 1995.

[50] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 international conference on Management of Data*, pages 1–12. ACM, 2012.

[51] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: a scalable log-structured database system in the cloud. *Proceedings of the VLDB Endowment*, 5(10):1004–1015, 2012.

[52] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. Spanstore: cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308. ACM, 2013.

[53] G. T. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.

[54] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.

APPENDIX

A. ANALYSIS AND TRADE-OFFS

In this section, we provide an analysis of the factors affecting the commit latency and the trade-off between commit latency on one hand and liveness or throughput on the other hand.

A.1 Commit latency analysis

This analysis shows how loose synchronization, communication links and compute variability affect the observable commit latency of Helios. We assume that Helios is assigned the *correct* commit offsets, which would lead to the exact lower-bound commit latency in perfect conditions.

Synchronization. Helios does not require clock synchronization for its correctness. However, the degree of synchronization affects the achieved commit latency. In particular, a low level of synchronization or the lack of synchronization will lead to degraded performance. Better clock synchronization will lead to commit latency values that are

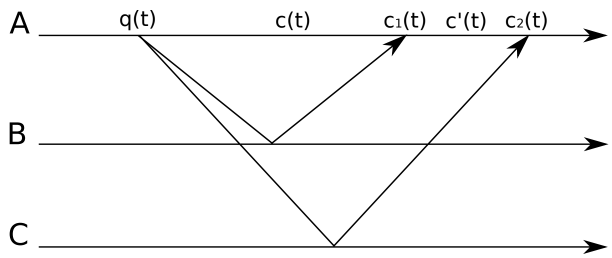

Figure 6: A transaction t executing at A demonstrating the trade-off between liveness and commit latency

closer to the optimal. Recall that the commit condition in Rule 2 states that a transaction must wait for every datacenter to send a message with a timestamp calculated by using the commit offsets. Using this protocol will result in the transaction committing with the commit latency estimated in Equation 4. However, with a clock skew between datacenters, the observed commit latency will be affected. Clock skew between two datacenters, A and B, will cause one of them to receive the expected message later than intended and the other will receive it earlier than intended. Thus, the actual time that a datacenter A waits until it receives the expected message from B for a transaction t is $L_A + \theta(A, B)$, where $\theta(A, B)$ is the time difference between A and B. $\theta(A, B)$ has a positive value if A's clock is ahead of B's clock. It is possible that A receives the message prior to requesting the commit; this is the case when $\theta < -L_A$. With the presence of clock skew the actual observed commit latency at A, $\bar{L_A}$, is given by

$$\bar{L_A} = L_A + Max_{B \in R}\{\theta(A, B)\} \tag{6}$$

RTT estimate accuracy. RTT estimation also plays a role in determining the achieved commit latency. To analyze the effect of RTT estimation errors, assume that the difference between the real RTT and the estimated RTT for A and B is given by $\rho_{A,B}$, a positive value if the real RTT is larger than the estimate. For a transaction waiting for a log of events (history) from other datacenters with a certain time, the one way latency is affected by a magnitude equal to half the error in the estimation. Thus, the average achieved latency when the effect of the propagation rate and RTT estimation error are factored becomes

$$\bar{L_A} = L_A + Max_{B \in R}\{\theta(A, B) \ + \frac{\rho_{A,B}}{2}\} \tag{7}$$

Communication link variability. The observable commit latency of individual transactions differ due to the variability of the communication latency. Thus, the RTT between two datacenters is best represented as a statistical random variable. In this case, Equation 7 will observe $\rho_{A,B}$, the error in RTT estimation, as a random variable, $\varrho_{A,B}$, rather than a fixed value, which means that $\bar{L_A}$ is also itself best represented as a random variable.

Compute overhead. The overhead of processing the log and requests in addition to various I/O and interface overheads increase the observed latency. For brevity we call the cumulative effect of all of these overheads, *compute overhead*. Measuring the compute overhead is intractable. However, it effects the observable commit latency in two ways: First, it is the overhead factors of the host of the transaction that we are observing. This can be accounted for in our expression by a random variable, call it $\boldsymbol{C}_{local}$, that affects the observable commit latency directly. Second, it determines the overhead factors that are taking place at other datacenters. These are observed in the commit latency of a transaction as a delay in receiving the needed information. This delay is represented with $\boldsymbol{C}^B_{remote}$, where B is the remote datacenter experiencing the overhead. This will make the observable commit latency be

$$\bar{L_A} = L_A + \boldsymbol{C}^A_{local} + Max_{B \in R}\{\theta(A, B) \ + \frac{\rho_{A,B}}{2} + \boldsymbol{C}^B_{remote}\} \tag{8}$$

Bounding observable commit latency. The observable commit latency depends on random variables that are unbounded. This results in the possibility of observing commit latencies that ranges from 0 to an infinitely large number, regardless of the assigned commit latency. Although Equation 8 can help us learn the distribution of observable commit latency values, it does not enable us to place an upper-bound or a lower-bound greater than zero on an individual transaction's commit latency. However, since it is a random variable, it will allow us to calculate a probability that a transaction's latency is greater than or equal to a certain value. This, indeed, remains an arduous task, since estimating the compute overhead and link variability random variables is still an area of research.

A.2 Trade-off with liveness and throughput

Liveness trade-off. There is a trade-off between the observable commit latency and liveness. As the number of tolerated datacenter failures increases, the commit latency is likely to increase as well. To illustrate this consider the scenario in Figure 6. A transaction t at A is trying to commit in a scenario with three datacenters. It requests to commit at time $q(t)$. Assume that there are no conflicts and that the knowledge required to commit t is accumulated by time $c(t)$. This means that if we did not want to tolerate failures, and thus use Helios-0, t is able to commit at time $c(t)$. However, if we want to tolerate a single datacenter failure and use Helios-1, t would need to wait until the record of the transaction is acknowledged by another datacenter, which happens to be at time $c_1(t)$. This means that the cost of increasing the level of liveness on t was increasing its commit latency by $(c_1(t) - c(t))$. Likewise, if two datacenter outages are to be tolerated, transaction t commits at time $c_2(t)$ causing an additional penalty on commit latency.

It is not always the case that increasing the number of tolerated datacenter failures will lead to an increase in commit latency. It is possible that the commit latency will remain the same. In Figure 6 assume that transaction t receives the sufficient information necessary to commit it at time $c'(t)$. Thus, Helios-0, without tolerating any failures, will commit t at time $c'(t)$. Now, Helios-1, tolerating a single datacenter failure will still commit t at time $c'(t)$. This is because t is already acknowledged by B at time $c'(t)$.

Throughput trade-off. In the paper we have focused on minimizing the average commit latency. However, there is an interesting trade-off between average commit latency and throughput. Assigning the optimal commit latency values to Helios will not necessarily result in the best overall throughput. Consider the topology and latency assignment in Table 1 of three datacenters with the following RTTs: $RTT(A, B) = 30$, $RTT(A, C) = 20$, and $RTT(B, C) = 40$. The optimal commit latency assignment to minimize the average commit latency is 5 for A, 25 for B, and 15 for C.

Assume that there are no aborts and that transactions complete in exactly the duration of the commit latency. Also, assume that the commit latency values are in ms. For N clients per datacenter, the cumulative throughput when the optimal commit latencies are assigned is: $1000 * N * (\frac{1}{5} + \frac{1}{25} + \frac{1}{15})$, which is $(N * 306.66)$ transactions per second. However, another correct assignment which is 1 for A, 29 for B, and 19 for C results in a cumulative throughput of $(N * 1087.11)$ transactions per second, which is larger than the throughput achieved by the optimal commit latency assignment.

Optimizing for higher throughput can be performed by utilizing the linear programming method in Section 3.3. The minimization condition can be changed to be a maximization condition to optimize for the sum of the rate of execution (inverse of commit latency) to be: $\sum_{A \in R} \frac{1}{L_A + c}$, where c is a constant value estimating the execution overhead of a transaction. Introducing c is necessary because otherwise the linear program would assign one of the datacenters a commit latency of 0, which erroneously yields an infinite throughput in the optimization problem. However, in a realistic environment execution takes time and therefore must be accounted for in this case.

More details can be introduced for this optimization to make it more accurate, hence higher throughput. One important factor is contention and the possibility of aborts. The maximization above assumes that all running transactions will commit successfully. However, it is likely that some transactions can abort, and a commit latency assignment that maximizes the expression above for throughput might lead to more contention than other assignments, one of which might have a higher throughput of *committed* transactions.

B. READ-ONLY TRANSACTIONS

Read-only transactions in Helios do not contend with other read-write transactions and are performed at the local datacenter. When a Helios instance receives a read-only transaction it chooses a log position to serve as the *snapshot point*. Thus, the read-only transaction will read the state of the data store as of the snapshot point. Every read request, r, will read the data object version that is written by a transaction t, where t is the most recent transaction that writes the data object, r, prior to the snapshot point. This approach is proposed by various log-structured systems such as Hyder [12] and LogBase [51] and is similar to read-only transactions in multi-version databases.