Blotter: Low Latency Transactions for Geo-Replicated Storage

Henrique Moniz[1*] João Leitão[2] Ricardo J. Dias[3] Johannes Gehrke[4]
Nuno Preguiça[2] Rodrigo Rodrigues[5]

[1]Google [2]NOVA LINCS & FCT, Universidade NOVA de Lisboa [3]NOVA LINCS & SUSE Linux GmbH
[4]Microsoft [5]INESC-ID & Instituto Superior Técnico, Universidade de Lisboa

ABSTRACT

Most geo-replicated storage systems use weak consistency to avoid the performance penalty of coordinating replicas in different data centers. This departure from strong semantics poses problems to application programmers, who need to address the anomalies enabled by weak consistency. In this paper we use a recently proposed isolation level, called Non-Monotonic Snapshot Isolation, to achieve ACID transactions with low latency. To this end, we present Blotter, a geo-replicated system that leverages these semantics in the design of a new concurrency control protocol that leaves a small amount of local state during reads to make commits more efficient, which is combined with a configuration of Paxos that is tailored for good performance in wide area settings. Read operations always run on the local data center, and update transactions complete in a small number of message steps to a subset of the replicas. We implemented Blotter as an extension to Cassandra. Our experimental evaluation shows that Blotter has a small overhead at the data center scale, and performs better across data centers when compared with our implementations of the core Spanner protocol and of Snapshot Isolation on the same codebase.

Keywords

Geo-replication; non-monotonic snapshot isolation; concurrency control.

1. INTRODUCTION

Many Internet services are backed by geo-replicated storage systems, in order to keep data close to the end user. This decision is supported by studies showing the negative impact of latency on user engagement and, by extension, revenue [15]. While many of these systems rely on weak consistency for better performance and availability [10], there is also a class of applications that require support for strong consistency and transactions. For instance, many applications within Google are operating on top of Megastore [3], a system that provides ACID semantics within the same shard, instead of Bigtable [7], which provides better performance but weaker semantics. This trend also motivated the development of Spanner, which provides general serializable transactions [9], and sparked other recent efforts in the area of strongly consistent geo-replication [28, 24, 22, 30, 16, 23].

In this paper, we investigate whether it is possible to further cut the latency penalty for ACID transactions in a geo-replicated systems, by leveraging a recent isolation proposal called Non-Monotonic Snapshot Isolation (NMSI) [24]. We present the design and implementation of Blotter, a transactional geo-replicated storage system that achieves: (1) at most one round-trip across data centers (assuming a fault-free run and that clients are proxies in the same data center as one of the replicas), and (2) read operations that are always served by the local data center. Additionally, when the client is either co-located with the Paxos leader or when that leader is in the closest data center to the client, Blotter can operate in a single round-trip to the *closest* data center.

To achieve these goals, Blotter combines a novel *concurrency control* algorithm that executes at the data center level, with a carefully configured Paxos-based replicated state machine that replicates the execution of the concurrency control algorithm across data centers. Both of these components exploit several characteristics of NMSI to reduce the amount of coordination between replicas. In particular, the concurrency control algorithm leverages the fact that NMSI does not require a total order on the start and commit times of transactions. Such an ordering would require either synchronized clocks, which are difficult to implement, even using expensive hardware [9], or synchronization between replicas that do not hold the objects accessed by a transaction [25], which hinders scalability. In addition, NMSI allows us to use separate (concurrent) Paxos-based state machines for different objects, on which we geo-replicate the *commit* operation of the concurrency control protocol.

Compared to a previously proposed NMSI system (Jessy [24]), instead of assuming partial replication we target full replication, which is a common deployment scenario [3, 27, 5]. Our layering of Paxos on top of a concurrency control algorithm is akin to the Replicated Commit system, which layers Paxos on top of Two-Phase Locking [23]. However, by leveraging NMSI, we execute reads exclusively locally, and run parallel instances of Paxos for different objects, instead of having a single instance per shard.

We implemented Blotter as an extension to Cassandra [17]. Our evaluation shows that, despite adding a small overhead in a single data center, Blotter performs much better than Jessy and the protocols used by Spanner, and outperforms in many metrics a replication protocol that ensures SI [12]. This shows that Blotter can be a valid choice when several replicas are separated by high latency links, performance is critical, and the semantic differences between NMSI and SI are tolerated by the application.

[*] Work done while the author was at NOVA LINCS.

ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052603

2. SYSTEM MODEL

Blotter is designed to run on top of any distributed storage system with nodes spread across one or multiple data centers. We assume that each data object is replicated at all data centers. Within each data center, data objects are replicated and partitioned across several nodes. We make no restrictions on how this intra-data center replication and partitioning takes place. We assume that nodes may fail by crashing and recover from such faults. When a node crashes, it loses its volatile state but all data that was written to stable storage is accessible after recovery. We use an asynchronous system model, i.e., we do not assume any known bounds on computation and communication delays. We do not prescribe a fixed bound on the number of faulty nodes within each data center. As we will see, our modular design allows for plugging in different replication protocols that run within each data center. As such, the bounds on faulty nodes depend on the intra-data center replication protocol.

3. NON-MONOTONIC SI

This section specifies our target isolation level, NMSI, and discusses the advantages and drawbacks of this choice. The reason for formalizing NMSI is twofold. First, our specification is simpler than the previous definition [24], thus improving in clarity and readability. Second, some of our key design choices follow naturally from this specification.

3.1 Snapshot isolation revisited

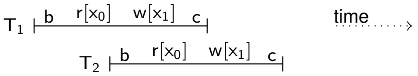
NMSI is an evolution of Snapshot Isolation (SI). Under SI, a transaction (logically) executes in a database snapshot taken at the transaction begin time, reflecting the writes of all transactions that committed before that instant. Reads and writes execute against this snapshot, and, at commit time, a transaction can commit if there are no write-write conflicts with concurrent transactions. (In this context, two transactions are concurrent if the intervals between their begin and commit times overlap.)

To define SI more precisely, we state that for any execution of a system implementing SI, we must be able to create a partial order among the transactions that were executed that (1) explains the values observed by all transactions, with reads returning the value written by the latest transaction, according to this partial order, that wrote to that object; (2) totally orders transactions that write to the same object; and (3) ensures that transactions see a snapshot that reflects all operations that committed before the transaction started. More precisely:

DEFINITION 3.1 (SNAPSHOT ISOLATION (SI)). *An implementation of a transactional system obeys SI if, for any trace of an execution of that system, there exists a partial order $\prec$ among transactions that obeys the following rules, for any pair of transactions t_i and t_j in that trace:*

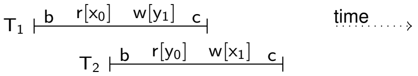
1. *if t_j reads a value for object x written by t_i then $t_i \prec t_j \wedge \nexists t_k$ writing to $x : t_i \prec t_k \prec t_j$*
2. *if t_i and t_j write to the same object x then either $t_i \prec t_j$ or $t_j \prec t_i$.*
3. *$t_i \prec t_j$ if and only if t_i commits before t_j begins.*

This definition captures the anomalies that are used in most definitions of SI. In particular, it prevents concurrent transactions from writing to the same object. For example, consider the following non-SI execution[1]:



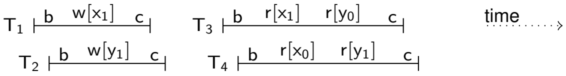
In the above example, transactions T_1 and T_2 write to the same object x and both commit. Such execution is impossible under SI, since two transactions that write to the same object must be ordered according to point number 2 of Definition 3.1, and, for any ordered pair of transactions, the first one must have committed before the start of the second transaction, according to point number 3.

The write-skew anomaly is also captured by Definition 3.1, since concurrent transactions with disjoint write-sets are not ordered. For example, the following execution meets SI, but is not serializable.



3.2 Specification of NMSI

NMSI weakens the SI specification in two ways. First, the snapshots against which transactions execute do not have to reflect the writes of a monotonically growing set of transactions. In other words, it is possible to observe what is called a "long fork" anomaly, where there can exist two concurrent transactions t_a and t_b that commit, writing to different objects, and two other transactions that start subsequently, where one sees the effects of t_a but not t_b, and the other sees the effects of t_b but not t_a. The next figure exemplifies an execution that is admissible under NMSI but not under SI, since under SI both T_3 and T_4 would see the effects of both T_1 and T_2 because they started after the commit of T_1 and T_2.



This relaxation affects Definition 3.1 by turning the equivalence in point 3 into an implication, i.e., it becomes:

3' if $t_i \prec t_j$ then t_i commits before t_j begins.

Second, instead of forcing the snapshot to reflect a subset of the transactions that committed at the transaction begin time, NMSI gives the implementation the flexibility to reflect a more convenient set of transactions in the snapshot, possibly including transactions that committed *after* the transaction began. This property, also enabled by serializability, is called *forward freshness* [24].

Going back to Definition 3.1, we can completely remove point 3, since it is now possible that the snapshot that t sees reflects writes from transactions that commit after t started, as long as the resulting snapshot is valid at some moment before the commit of t, i.e.:

DEFINITION 3.2 (NON-MON. SNAPSHOT ISOL. (NMSI)). *An implementation of a transactional system obeys NMSI if, for any trace of the system execution, there exists a partial order $\prec$ among transactions that obeys the following rules, for any pair of transactions t_i and t_j in the trace:*

1. *if t_j reads a value for object x written by t_i then $t_i \prec t_j \wedge \nexists t_k$ writing to $x : t_i \prec t_k \prec t_j$*
2. *if t_i and t_j write to the same object x then either $t_i \prec t_j$ or $t_j \prec t_i$.*

The example in Figure 1 obeys NMSI but not SI, as the depicted partial order meets Definition 3.2, but it is not possible to create a partial order obeying all three requirements of Definition 3.1.

[1] The notation b, r[$\mathsf{x_j}$], w[$\mathsf{y_l}$], c and a refers to the following operations of a transaction: begin; read version j of object x; write version l of object y; commit; and abort.

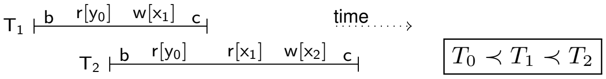


Figure 1: Example execution obeying NMSI but not SI.

3.3 What is enabled by NMSI?

NMSI weakens the specification of SI through the two properties we mentioned previously, which are individually leveraged by the design of Blotter.

The possibility of having "long forks" allows, in a replicated setting, for a single (local) replica to make a decision concerning what data the snapshot should read. This is because, in any highly available design for the commit protocol, there is necessarily the possibility of some replicas not seeing a subset of the most recent commits (since otherwise it would be impossible to provide availability when a data center is unreachable). As such, in a situation where snapshots are based on local information, and a replica in data center $DC1$ sees the writes of t_1 but not t_2, and conversely a replica in $DC2$ sees the writes of t_2 but not t_1, then the two local snapshots taken at each of these replicas can lead to the "long fork" anomaly we mentioned, where two transactions proceed independently. Avoiding this situation would require a serialization between all transaction begin and commit operations.

In the case of "forward freshness", this allows for a transaction to read (in most cases) the most recent version of a given replica, without having to worry about the instant when the transaction began. This not only avoids the bookkeeping associated with keeping track of transaction start times, but also avoids a conflict with transactions that might have committed after the transaction began.

3.4 Discussion: Limitations of NMSI

We analyze in turn the impact of "forward freshness" and "long forks" on programmability. Forward freshness allows a transaction x to observe the effects of another transaction y that committed after x began (in real-time). In this case, the programmer must decide whether this is a violation of the intended application semantics, analogously to deciding whether serializability or strict serializability is the most adequate isolation level for a given application. Long forks allow two transactions to be executed against different branches of a forked database state, provided there are no write-write conflicts. In practice, the main implication of this fact is that the updates made by users may not become instantly visible across all replicas. For example, this could cause two users of a social network to each think that they were the first to post a new promotion on their own wall, since they do not see each other's posts immediately [28]. Again, the programmer must reason whether this is admissible. In this case, a mitigating factor is that this anomaly does not cause the consistency of the database to break. (This is in contrast with the "write skew" anomaly, which is present in both SI and NMSI.) Furthermore, in the particular case of our implementation of NMSI, the occurrence of anomalies is very rare: for a "long fork" to occur, two transactions must commit in two different data centers, form a quorum with a third data center, and both complete before hearing from the other.

Finally, NMSI allows consecutive transactions from the same client to observe a state that reflects a set of transactions that does not grow monotonically (when consecutive transactions switch between two different branches of a long fork). However, in our algorithms this is an unlikely occurrence, since it requires that a client connects through different data centers in a very short time span.

4. ARCHITECTURE OF BLOTTER

The client library of Blotter exposes an API with the expected operations: `begin` a new transaction, `read` an object given its identifier, `write` an object given its identifier and new value, and `commit` a transaction, which either returns commit or abort.

The set of protocols that comprise Blotter are organized into three different components. This not only leads to a modular design, but also allows us to more clearly define the requirements and design choices of each component.

Blotter intra-data center replication. At the lowest level, we run an intra-data center replication protocol, to mask the unreliability of individual machines within each data center. This level must provide the protocols above it with the vision of a single logical copy (per data center) of each data object and associated metadata, which remains available despite individual node crashes. We do not prescribe a specific protocol for this layer, since any of the existing protocols that meet this specification can be used.

Blotter Concurrency Control. (Section 5.) These are the protocols that ensure transaction atomicity and NMSI isolation in a single data center, and at the same time are extensible to multiple data centers by serializing a single protocol step.

Paxos. (Section 6.) This completes the protocol stack by replicating a subset of the steps of the concurrency control protocol across data centers. It implements state machine replication [26, 18] using a careful parameterization of Paxos [19]. However, state machine replication must be judiciously applied to the concurrency control protocol, to avoid unnecessary coordination across data centers.

5. SINGLE DATA CENTER PROTOCOL

5.1 Overview

We start by explaining how we derive the concurrency control protocol from the NMSI requirements.

Partial order $\prec$. We use a multi-version protocol, i.e., the system maintains a list of versions for each object. This list is indexed by an integer version number, which is incremented every time a new version of the object is created (e.g., for a given object x, overwriting x_0 creates version x_1, and so on). In a multi-versioned storage, the $\prec$ relation can be defined by the version number that transactions access, namely if t_i writes x_m and t_j writes x_n, then $t_i \prec t_j \Leftrightarrow m < n$; and if t_i writes x_m and t_j reads x_n, then $t_i \prec t_j \Leftrightarrow m \leq n$.

NMSI rule number 1. Rule number 1 of the definition of NMSI says that, for object x, transaction t must read the value written by the "latest" transaction that updated x (according to $\prec$). To illustrate this, consider the example run in Figure 2. When a transaction $T1$ issues its first read operation, it can read the most recently committed version of the object, say x_i written by $T0$ (leading to $T0 \prec T$). If, subsequently, some other transaction $T2$ writes x_{i+1} ($T0 \prec T2$), then the protocol must prevent $T1$ from either reading or overwriting the values written by $T2$. Otherwise, we would have $T0 \prec T2 \prec T1$, and $T1$ should have read the value for object x written by $T2$ (i.e., x_{i+1}) instead of that written by $T0$ (i.e., x_i). Next, we detail how this is achieved first for reads, then writes, and then how to enforce the rule transitively.

Reading the latest preceding version. The key to enforcing this requirement is to maintain state associated with each object, stating the version a running transaction must read, in case such a restriction exists. In the previous example, if $T2$ writes x_{i+1}, this state records that $T1$ must read x_i.

To achieve this, our algorithm maintains a per-object dictionary data structure (x.*snapshot*), mapping the identifier of a transaction t

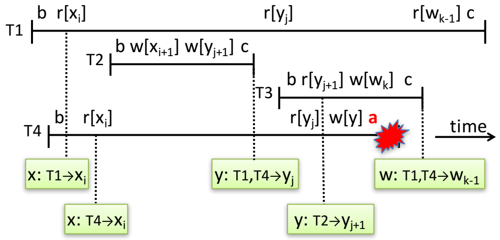


Figure 2: Example run.

to a particular version of x that t either must read or has read from. Figure 2 depicts the changes to this data structure in the shaded boxes at the bottom of the figure. When t issues a read for x, if the dictionary has no information for t, the most recent version is read and this information is stored in the dictionary. Otherwise, the specified version is returned.

In the previous example, $T1$ must record the version it read in the *x.snapshot* variable. Subsequently, when the commit of $T2$ overwrites that version of x, we are establishing that $T2 \not\prec T1$. As such, if $T2$ writes to another object y, creating y_{j+1}, then it must also force $T1$ to read the preceding version y_j. To do this, when transaction $T2$ commits, for every transaction t that read (or must read) an older version of object x (i.e., the transactions with entries in the dictionary of x), the protocol will store in the dictionary of every other object y written by $T2$ that t must read the previous version of y (unless an even older version is already prescribed). In this particular example, $y.snapshot$ would record that $T1$ and $T4$ must read version y_j, since, at commit time, $x.snapshot$ indicates that these transactions read x_i.

Preventing illegal overwrites. In the previous example, we must also guarantee that $T1$ does not overwrite any value written by $T2$. To enforce this, it suffices to verify, at the time of the commit of transaction t, for every object written by t, if T should read its most recent version. If this is the case, then the transaction can commit, since no version will be incorrectly overwritten; otherwise, it must abort. In the example, $T4$ aborts, since $y.snapshot$ records that $T4$ must read y_j and a more recent version exists (y_{j+1}). Allowing $T4$ to commit and overwrite y_{j+1} would lead to $T2 \prec T4$. This breaks rule number 1 of NMSI, since it would have required $T1$ to read x_{i+1} written by $T2$, which did not occur.

Applying the rules transitively. Finally, for enforcing rule number 1 of the definition of NMSI in a transitive manner, it is also necessary to guarantee the following: if $T2$ writes x_{i+1} and y_{j+1}, and subsequently another transaction $T3$ reads y_{j+1} and writes w_k, then the protocol must also prevent $T1$ from reading or overwriting the values written by $T3$, otherwise we would have $T0 \prec T2 \prec T3 \prec T1$, and thus $T1$ should also have read x_{i+1}.

To achieve this, when transaction $T3$ (which read y_{j+1}) commits, for every transaction t that must read version y_l with $l < j+1$ (i.e., the transactions that had entries in the dictionary of y when t_o read y), the protocol will store in the dictionary of every other object w written by $T3$ that t must read the previous version of w (if an older version is not already specified). In the example, since the state for $y.snapshot$ after $T2$ commits specifies that $T1$ must read version y_j, then, when $T3$ commits, $w.snapshot$ is updated to state that $T1$ and $T4$ must read version w_{k-1}.

NMSI rule number 2. Rule number 2 of the NMSI definition says that any pair of transactions that write the same object x must have a relative order, i.e., either $t_i \prec t_j$ or $t_j \prec t_i$. This order is defined by the version number of x created by each transaction.

Therefore, it remains to ensure that this is a partial order (i.e., no cycles). A cycle could appear if two or more transactions concurrently committed a chain of objects in a different order, e.g., if t_m wrote both x_i and y_{j+1} and t_n wrote both x_{i+1} and y_j. To prevent this, it suffices to use a two-phase commit protocol where, for each object, a single accepted prepare can be outstanding at any time.

Waiving SI rule number 3. The fact that NMSI does not have to enforce rule number 3 (which is present only in SI) already allows for some performance gains in the single data center protocol, even though other, more impactful advantages will only become clear when we extend the protocol to multiple data centers in Section 6. In particular, if we consider the example in Figure 1, transaction T_2 is bound to abort in SI after it read version 0 of y concurrently with T_1 creating version 1 of x. This is true of any concurrency control scheme that implements SI because, when T_2 subsequently reads x, SI requires it to return the version corresponding to the snapshot taken when the transaction started (i.e., x_0), and the subsequent write to x would generate a write-write conflict. In contrast, in our NMSI design, the commit of T_1 records in *x.snapshot* that T_2 should read version 1, which avoids this situation.

5.2 Protocol design

The single data center concurrency control module consists of the following three components: the client library and the transaction managers (TM), which are non-replicated components that act as a front end providing the system interface and implementing the client side of the transaction processing protocol, respectively; and the data managers (DM), which are the replicated components that manage the information associated with data objects.

Client Library. This provides the interface of Blotter, namely *begin*, *read*, *write*, and *commit*. The *begin*, and *write* operations are local to the client. *Read* operations are relayed to the TM, who returns the values and metadata for the objects that were read. The written values are buffered by the client library and only sent to the TM at *commit* time, together with the accumulated metadata for the objects that were read. This metadata is used to set the versions that running transactions must access, as explained next.

Transaction Manager (TM). The TM handles the two operations received from the clients: *read* and *commit*. For *read*s, it merely relays the request and reply to or from the Data Manager (DM) responsible for the object being read. Upon receiving a *commit* request, the TM acts as a coordinator of a two-phase commit (2PC) protocol to enforce the all-or-nothing atomicity property. The first phase sends a *dm-prewrite-request*, with the newly written values, to all DMs storing written objects. Each DM verifies if the write complies with NMSI. If none of the DMs identifies a violation, the TM sends the DMs a *dm-write* message containing the metadata with snapshot information aggregated from all replies to the first phase; otherwise it sends a *dm-abort*.

Data Manager (DM). The core of the concurrency control logic is implemented by the DM. Algorithm 1 describes its handlers for the three types of requests.

(i) Read operation. The handler for a read of object x by transaction T returns either the version of x stored in *x.snapshot* for T, if the information is present, or the most recent version and then sets *x.snapshot*$[T]$ to that version, so that a subsequent read to the same variable reads from the same snapshot (and also, as we will see, for propagating snapshot information to enforce NMSI).

Before returning, the read operation must wait in case a concurrent transaction is trying to commit a new value for x (a standard 2PC check, which is done by inspecting x.prewrite). However, blocking is not needed when the snapshot variable forces a transaction to read a prior version.

Algorithm 1: Single data center DM protocols

```
// read operation
upon $\langle$ dm-read, T, x $\rangle$ from TM do
    processRead $\langle$ T, x, TM $\rangle$;
// prewrite operation
upon $\langle$ dm-prewrite, T, x, value $\rangle$ from TM do
    if $x.prewrite \neq \bot$ then
        // another prewrite is pending
        x.pending $\leftarrow$ x.pending $\cup$ {(T, x, value, TM)};
    else
        processPrewrite $\langle$ T, x, value, TM $\rangle$;
// write operation
upon $\langle$ dm-write, T, x, agg-startd-before $\rangle$ from TM do
    for each T' in agg-startd-before do
        if T' not in x.snapshot then
            x.snapshot[T'] $\leftarrow$ x.last;
    x.last $\leftarrow$ x.last + 1;
    x.value[x.last] $\leftarrow$ x.nextvalue;
    finishWrite $\langle$ T, x, TM $\rangle$;
// abort operation
upon $\langle$ dm-abort, T, x $\rangle$ from TM do
    finishWrite $\langle$ T, x, TM $\rangle$;
// process read operation
processRead $\langle$ T, x, TM $\rangle$
    if $T \notin x.snapshot$ then
        if $x.prewrite \neq \bot$ then
            x.buffered $\leftarrow$ x.buffered $\cup$ {(T, TM)};
            return
        else
            x.snapshot[T] $\leftarrow$ x.last;
    version $\leftarrow$ x.snapshot[T];
    value $\leftarrow$ x.value[version];
    send $\langle$ read-response, T, value, $\{T' | x.snapshot[T'] < version\}$ $\rangle$ to
      TM;
// process dm-prewrite request
processPrewrite $\langle$ T, x, value, TM $\rangle$
    if $x.snapshot[T] \neq \bot \wedge x.snapshot[T] < x.last$ then
        // there is a write-write conflict
        send $\langle$ prewrite-response, reject, $\bot$ $\rangle$ to TM;
    else
        x.prewrite $\leftarrow$ T;
        x.nextvalue $\leftarrow$ value;
        send $\langle$ prewrite-response, accept, $\{T' | T' \in x.snapshot\}$ $\rangle$ to TM;
// clean prewrite information and serve buffered
   reads and pending prewrites
finishWrite $\langle$ T, x, TM $\rangle$
    if x.prewrite = T then
        x.nextvalue $\leftarrow$ $\bot$; x.prewrite $\leftarrow$ $\bot$;
    for each (T, TM) in x.buffered do
        processRead $\langle$ T, x, TM $\rangle$;
    if $x.pending \neq \bot$ then
        (T, x, value, TM) $\leftarrow$ removeFirst(x.pending);
        processPrewrite $\langle$ T, x, value, TM $\rangle$;
```

Finally, the metadata returned to the TM are the identifiers of all transactions present in $x.snapshot$ that must read from a version prior to the one returned, i.e., all T' that must obey $T \not\prec T'$ due to T reading x. This information is aggregated by the client library, and propagated to DMs in phase 2 of the commit, to ensure that those T' read a state prior to T. As explained in Section 5.1, this enforces point 1 of the NMSI definition transitively.

(ii) Prewrite operation. This first phase for the commit of T has two goals: detect write-write conflicts, and collect information about concurrent transactions, which is subsequently added to their *snapshot* variables. After checking (and if needed blocking) if there is a concurrent prewrite for an object written by T, the DM detects write-write conflicts by checking if T has to read an older version for any written object x (i.e., by checking $x.snapshot[T]$). If so, then T is writing to an object that was written by a concurrent transaction (i.e., a write-write conflict would violate Rule 1 of the NMSI Definition), and a *reject* is replied. Otherwise, *prewrite* and *nextvalue* are set, in order to block concurrent accesses to x, and an *accept* is returned, including as metadata the identifiers of all transactions in $x.snapshot$. These are the transactions that cannot be serialized after T according to $\prec$, since T is overwriting data they either read or must read. This information is aggregated by the TM and used in the next phase.

(iii) Write and abort operations. If any of the participating DMs detects a write-write conflict, then the TM sends *abort* messages to all DMs involved in T, who then remove T from *prewrite* and *nextvalue*, thus unblocking pending transactions. Otherwise, the TM sends a *write* to all DMs in T, containing the aggregated metadata, comprising the set of identifiers of all transactions present in the snapshot variable for any object read or written by T. Upon receiving a write request, the DM responsible for x will first set $x.snapshot[T']$ to the current version of x, for any transaction T' in the previous set, thus enforcing that $T \not\prec T'$. After this step, the DM will create a new version of x by incrementing its version number, and the new version is made visible to other transactions by updating x.last and x.value[x.last]. Finally, reads pending on that transaction commit are executed, followed by pending prewrites.

5.3 Garbage Collection

The per-object $x.snapshot$ data structure needs a garbage collection mechanism to prevent the number of entries from growing without bound. This is particularly important since these entries are propagated from one data object to another at the time of commit.

An entry $x.snapshot[T]$ guarantees that T reads a version of x that will not break NMSI rules and also enables the detection of write-write conflicts between T and other committed transactions when T attempts to commit. This means that an entry for T in the *snapshot* data structure only needs to be maintained while T is executing. When T terminates, the entry should be removed as soon as possible to avoid a needless propagation to the $snapshot$ structure of other objects. We take advantage of this observation to implement a simple garbage collection scheme, where each transaction T is created with a time to live (TTL), which reflects the maximum time the transaction is allowed to run. After the TTL of T expires, any entries for T are automatically garbage collected. We analyze the efficiency of this mechanism in Section 7.5.

This mechanism also enables us to garbage collect old versions of data objects: any version of an object x, other than the most recent one, with no entries in the $x.snapshot$ data structure pointing to it can be safely garbage collected.

6. GEO-REPLICATION

Blotter implements geo-replication, with each object replicated in all data centers, using Paxos-based state machine replication [19, 26]. In this model, all replicas execute a set of client-issued commands according to a total order, thus following the same sequence of states and producing the same sequence of responses. We view each data center as a state machine replica. The state is composed by the database (i.e., all data objects and associated metadata), and the state machine commands are the `tm-read` and the `tm-commit` of the TM-DM interface.

Despite being correct, this approach has three drawbacks: (1) read operations in our concurrency control protocol are state machine commands that mutate the state, thus requiring an expensive consensus round; (2) the total order of the state machine precludes the concurrent execution of two commits, even for transactions that do not conflict; and (3) each Paxos-based state machine command requires several cross-data center message delays (depending on the variant of Paxos used). We next refine our design by addressing each of these concerns, including a discussion on deadlocks.

(1) Local read operations. Read operations modify the snapshot variable state and need to be executed in the state machine, incurring in the cross-data center latency of the replication protocol.

To avoid this overhead, we propose to remove the contents of the snapshot variable from the state machine. The replicas still maintain their view of the snapshot variable, but the information is independently maintained by each replica. This is feasible under NMSI since the snapshot information is used for only two purposes.

The first one is to determine whether write-write conflicts exist. This happens if, at commit time of transaction T, for some modified object x, the snapshot information for T is not the most recent version of x. Since the snapshot information is now updated outside of the state machine, only the replica in the data center where T is initiated records the information for snapshot$[T]$. To allow all data centers to deterministically check for write-write conflicts, we include, as a parameter of the *commit* state machine command, the snapshot information present at the data center where T executed, for each object x modified by T. While this works seamlessly for objects that are both read and written by T, this does not handle the case of a blind write to x, since x.snapshot$[T]$ was not defined in this case. This can be handled by forcing blind writes to perform an artificial read to define x.snapshot$[T]$, right before T commits.

The second use of the information in the snapshot data structure is during transaction execution (i.e., before the commit) to determine which version of an object should be observed by a transaction T, so that a consistent snapshot for T is read. However, this is a local use of the information, and therefore it does not have to be maintained by the state machine. (This forces clients to restart ongoing transactions when failing over to another data center.)

By having consistent reads outside the state machine, read-only transactions can run locally in the data center where the TM runs.

Connection to NMSI. These modifications are possible because, unlike SI, NMSI allows for independence between the state reflected by transactions and the real-time instant when transactions begin and commit. Otherwise, the local snapshot variable might not be up-to-date as other transactions commit (see point (3)).

(2) Concurrent execution of database operations. Enforcing a state machine total order on database operations and executing them serially ensures consistency, but defeats the purpose of concurrency control, which is to enable transactions to execute concurrently.

To address this, we observe that the partial order required by the NMSI definition can be built by serializing the *dm-prewrite* operation on a per-object basis, instead of serializing *tm-commit*s across all objects. This is because a per-object serialization of the first phase of the two-phase commit suffices to ensure that the transaction outcome is the same at all data centers and that the state transformation of each object involved in the *tm-commit* operation is also the same at all replicas, and therefore the database evolves consistently and obeying NMSI across all data centers.

As such, instead of having one large state machine whose state is defined by the entire database, we can have one state machine per object, with the state being the object (including its metadata), and supporting only the *dm-prewrite* operation. The data center where the transaction executes is the one to issue the *dm-prewrite*s for the objects involved in the transaction, and, since the outcome is the same at all data centers, each data center can subsequently commit or abort the transaction independently, without coordination.

Connection to NMSI. This important optimization is made possible by the NMSI semantics, namely the possibility of having long forks. The intersection property between read and write quorums is only required for quorums of the same instance. As such, a commit to an object x may not be reflected in the state read by the Paxos instance for object y and vice-versa, thus leading to a long fork.

Deadlock Resolution. As presented so far, Blotter can incur in deadlocks when more than one transaction writes to the same set of objects, and the Paxos instances for these objects serialize the transactions in a different order. The possibility of deadlock is common across any two-phase commit-based system [28, 9, 23], and, since it has been addressed in the literature, we consider it orthogonal to our contribution. In particular, due to the use of Blotter Paxos, deadlocks are *replicated* across different data centers, and therefore any deterministic deadlock resolution scheme, such as edge-chasing [11], can be employed.

(3) Paxos with a single cross-data center round-trip. We adjusted the configuration of Paxos to reduce the cross data center steps to a single round-trip (from the client of the protocol, i.e., the TM) for update transactions. We leverage two techniques.

The first is to use a variant called Multi-Paxos [19], which allows, in the normal case, for command execution to proceed as follows: client (i.e., TM) to Paxos leader; Paxos leader to all replicas; all replicas to client. The second technique leverages the observation that data center outages are rare, and given that we are using a lower layer of replication protocols to make each Paxos replica fault-tolerant, it is sensible to configure Paxos to only tolerate one unplanned outage of a data center. In fact, this configuration is common in existing deployed systems [2, 9]. (Planned outages are handled by reconfiguring the Paxos membership [20].) Given this observation, we can parameterize Paxos to use read quorums of $N-1$ and write quorums of 2 processes (where N is the number of data centers). This allows the following optimization: upon receiving an operation from the leader, a replica knows that the operation is decided, since it gathered a quorum of two processes between itself and the leader. This allows a TM to commit a transaction incurring in a single cross-data center round trip for each object, irrespectively of the TM being co-located with the Paxos leaders.

Connection to NMSI. This optimization could be applied to other systems that use Paxos in the context of replicated transactions, although, as stated previously, a design that uses different replicas groups for different objects would require quorum intersection across replica groups in SI but not in NMSI. Such quorum intersection would be possible if all groups used the same quorums (e.g., majorities), whereas in NMSI we can use asymmetric quorums and optimize the location of the Paxos leader per-object.

7. EVALUATION

We evaluated Blotter on EC2, by comparing it to Cassandra, an implementation of Spanner's Two-Phase Locking (2PL), a full replication SI protocol [12], and Jessy.

Our evaluation uses various benchmarks and workloads, namely: microbenchmarks for latency and throughput (§7.2); an adaptation of the RUBiS benchmark to key-value stores (§7.3); and a social networking workload (§7.4). We also evaluate the garbage collection mechanism (§7.5); and conduct a separate comparison to Jessy, the other system that offers NMSI (§7.6).

7.1 Experimental Setup

We conducted the experiments on EC2 using data centers from three availability regions: Ireland (EU), Virginia (US-E), and California (US-W). The following table shows the roundtrip latencies between these data centers.

	Ireland	Virginia
Virginia	97	-
California	167	79

A server cluster composed of four virtual machines was setup in each data center. Four additional virtual machines per data center were used as clients. Each virtual machine is an extra large instance with a 64-bit processor architecture, 4 virtual cores, and 15 GB of RAM. Within each data center, keys are mapped to servers using consistent hashing.

We compared the following geo-replicated systems. (In all systems, centralized components, namely lock servers for 2PL and Paxos leaders, ran in Ireland.)

Blotter. We implemented Blotter on top of Cassandra. In particular, we extended Cassandra and its Thrift API with the TM and DM logic of Blotter and implemented a client library supporting the *begin*, *read*, *write*, *commit* interface. For intra-data center replication, Blotter uses Cassandra replication with $N = 2$ replicas, with write quorums of two replicas and read quorums of one replica.

Cassandra. Cassandra is a popular open source NoSQL key-value store [17]. Cassandra does not support transactions, and the consistency of individual operations is defined by the clients, who specify the number of replicas that are contacted in the foreground, before the operation returns. We used two different configurations, both with $N = 3$ replicas (one per data center): (1) the *local quorum* configuration uses read and write quorums of a single replica, thus providing weak consistency; and (2) the *'each' quorum* configuration, where a read operation completes after contacting exactly one replica (at the local data center), and a write operation completes after contacting all three replicas, thus providing strong consistency. Although Cassandra does not support transactions, we compare it against the other systems by clustering its operations into logical groups (which we simply call *transactions*), each containing zero or more read operations and optionally terminated by an aggregated set of write operations.

Spanner's 2PL. We chose Spanner [9] as one of the comparison points because of its relevance, since it is a production system deployed at Google. Update transactions in Spanner are an implementation of the two-phase locking/two-phase commit (2PL/2PC) technique for serializable transactions [4], on top of a Paxos-replicated log. By further leveraging the TrueTime API, Spanner also provides external consistency, or linearizability. (Conversely, Blotter uses Paxos to replicate the atomic commit operation across data centers instead of the log.) We extended Cassandra with transactions using the 2PL/2PC approach of Spanner, with a centralized lock server, on top of a Paxos-replicated log. We left out TrueTime, thus providing serializability, and favoring the performance of our implementation of Spanner's 2PL/2PC.

Generalized SI. Generalized SI (GSI) [12] is an extension of SI suitable for replicated databases. GSI allows transactions to read from *older* snapshots, whereas SI requires transactions to read from the most recent snapshot. The GSI algorithm assumes a full replication scenario where each replica contains the full copy of the database. Read operations can read from any replica, and commit operations must be serialized either using either a centralized transaction certifier or a state-machine approach. (We reused the Paxos implementation of Blotter.) We implemented GSI as an extension of Cassandra, with a replica of each data item per data center, and therefore local reads do not contact remote data centers.

7.2 Microbenchmarks

We measured latency and throughput under a simple workload, which parameterizes the number of read and write operations in each transaction. For this set of experiments, we loaded the database with 10 million random keys and random 256-byte values.

Latency. We first studied how the operations that comprise a transaction affect its latency. In this experiment, each transaction was composed of a single read operation and by either one or five write operations applied at commit time. For each run, the load consisted of a single client machine with a single thread executing 10,000 transactions serially. We used both a single data center in Ireland, and a configuration using all data centers.

The results in Table 1 show the latency (median and 99th percentile) for individual read and commit operations with both one and five write operations (W=1 and W=5).

The single data center configuration evaluates the protocol overheads when the latency between nodes is small. The results show that reads in Blotter incur a slight overhead with respect to the baseline Cassandra implementation (less than half a millisecond) due to the extra steps of writing snapshot information and acquiring locks. For commits, the differences are more pronounced, due the fact that Cassandra only requires two message exchanges, while Blotter, GSI, and 2PL require four because of 2PC.

For the multi-data center configuration, the previous overheads are dwarfed by the inter-data center latency. The Cassandra local quorum configuration is the only one that does not require cross-data center communication, and thus performs similarly in both configurations. Compared to the remaining systems, Blotter and GSI perform better because they only require a response from a single remote data center, in most cases the closest one. Cassandra ('each' quorum) requires replies from all data centers, so the commit latency reflects the latency between the client and the farthest data center. The latency of 2PL is higher than the other protocols because it requires more cross-data center round-trips for commits, and it also has to contact the data center responsible for the read and write locks for both types of transactions.

The results also show that GSI has a slightly lower latency than Blotter, which is likely due to the performance variance of the virtualized, wide-area environment.

Throughput. For measuring throughput, we used transactions with different combinations of read and write operations, including read-only, write-only, and mixed transactions. Each transaction executes R read operations serially followed by a commit with W write operations. We varied the number of concurrent client threads from 12 to 360, equally split across all data centers, where each client thread executes its transactions in a serial order. Figure 3 presents the maximum observed throughput for a configuration spanning all three data centers.

For read-only workloads, 2PL has a much lower throughput due to the fact that is does not necessarily read from the local data center. The overhead due to the extra processing in Blotter compared to Cassandra is also visible, particularly for R=5. For the write-only and mixed workloads, and focusing on the systems that require cross-data center coordination, Blotter has the highest throughput due to its more efficient cross-data center communication pattern and, compared to GSI, because of the increased parallelism, which, as we explained, is fundamentally tied to the use of NMSI.

7.3 Auction Site

We also evaluated Blotter using the RUBiS benchmark, which models an auction site similar to eBay. We ported the benchmark from using a relational database as the storage backend to using a key-value store. Each row of the relational database is stored with a key formed by the name of the table and the value of the primary key. We additionally store data for supporting efficient queries (namely indexes and foreign keys).

The workload consists of a mix with 85% of the interactions containing only read-only transactions, and 15% of the interactions containing read-write transactions. We initially load the key-value store with 10,000 users, 1,000 old items, and 32,667 active items.

	Single Data Center			Multi Data Center		
System	Read Latency	Commit Latency		Read Latency	Commit Latency	
		$W = 1$	$W = 5$		$W = 1$	$W = 5$
Cassandra (local quorum)	0.61 / 4.2	0.62 / 3.3	0.62 / 3.3	1.02 / 6.7	3.41 / 6.3	3.07 / 6.0
Cassandra ('each' quorum)	0.65 / 4.2	1.55 / 3.3	1.51 / 3.3	1.12 / 6.0	180.75 / 189.7	171.49 / 181.7
Blotter	0.98 / 5.0	1.46 / 4.3	1.45 / 4.3	1.60 / 7.5	85.47 / 150.0	87.09 / 128.7
GSI	0.71 / 4.3	1.75 / 5.3	1.76 / 5.3	1.21 / 79.0	79.21 / 82.7	78.89 / 82.3
2PL	0.62 / 4.0	0.61 / 4.0	0.60 / 4.0	1.72 (local), 85.19 (remote) / 150.7	247.32 / 321.7	246.17 / 310.0

Table 1: Latency of microbenchmarks in milliseconds (median / 99th percentile)

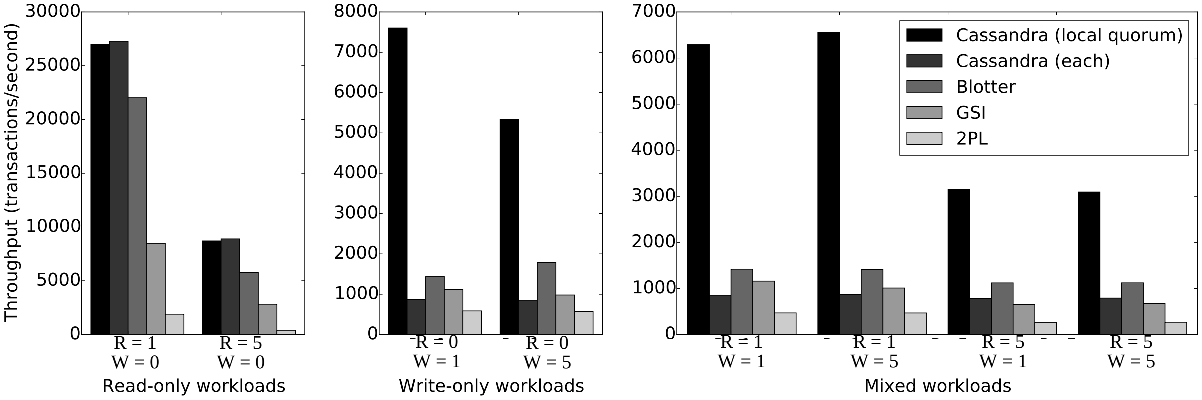


Figure 3: Throughput of multi-data center microbenchmarks

Figure 4 depicts our experimental results for the RUBiS workload. The results show that, consistently with the microbenchmark results, Blotter outperforms 2PL in terms of throughput for both read-only and read-write operations, since the design of Blotter minimizes the need for communication and coordination across data centers.

7.4 Microblogging

This experiment evaluates a mockup implementation of Twitter, supporting three different user interactions, modeled after [30]: *Post-tweet* appends a tweet to the wall of a user and its followers, which results in a transaction with many reads and writes. *Follow-user* appends new information about the set of followers to the profiles of the follower and the followee, which results in a transaction with two reads and two writes. Finally, *read-timeline* reads the wall of the user, resulting in a single read operation.

The workload consists of the following mix of interactions: 85% read-timeline, 10% post-tweet, and 5% follow-user. The database contains 100,000 users and each has an average of 6 followers. For each system, we varied the number of client threads from 120 to 720 and measured the maximum observed throughput.

The results in Figure 5 show a similar pattern to the throughput microbenchmarks. For the read-timeline (read-only) operation, Cassandra achieves the best throughput (60K tx/s), followed by Blotter (50K tx/s), and 2PL (10K tx/s). For the post-tweet and follow-user operations, which contain updates, Blotter has the highest throughput, followed by Cassandra, and then 2PL.

7.5 Garbage Collection

The garbage collection mechanism of Blotter is required to prevent the number of entries in the snapshot data structure of objects from growing without bound, which is important since this impacts the protocol message size.

To analyze the overhead of garbage collection, we deploy Blotter with one server and one client, and set the TTL to 1 second. The client executes a workload parameterized by (1) the number of objects in the database, and (2) the number of read and write operations per transaction. Both directly affect the contention and, consequently, the rate of propagation of snapshot entries in the database. As a metric of the efficiency of the garbage collection mechanism we use the number of snapshot entries returned with each read operation, as this reflects the size of the snapshot data structure at the time of the operation.

Figure 6 shows the average number of snapshot entries per read as a function of how far into the trace we are, for two database configurations: with 1000 objects (less contention) and 100 objects (more contention). For every tested combination of parameters, the GC mechanism stabilizes the size of the snapshot data structures to a very reasonable level of at most a few tens of entries. We also observed a peak in the size of the state that is returned, which happens because the high contention eventually saturates the system, thus increasing the latency of the transactions and lowering the rate at which snapshot entries are propagated.

7.6 Comparison to Jessy

We compared Blotter to the Jessy implementation of NMSI provided by its authors [25]. Jessy was the first system to explore NMSI as the isolation level for transactions in a geo-replicated setting. In contrast to Blotter, it focuses on partial replication settings (i.e., a given data center may not replicate the entire data set). For building consistent snapshots, Jessy uses a data structure called *dependence vector*, which contains either one entry per data item or per set of data items, depending on the configuration. This data structure is stored with data objects in the database, and must be read by clients and propagated within write operations. The propagation of transactions across machines uses total order multicast.

We first ran experiments in a single data center (EC2 in Ireland), where both systems were deployed across four servers, using both dependence vector configurations above. In these experiments we set a replication factor of two and a total of $50,000$

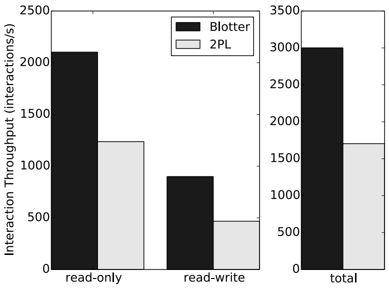


Figure 4: Average user interaction throughput for RUBiS

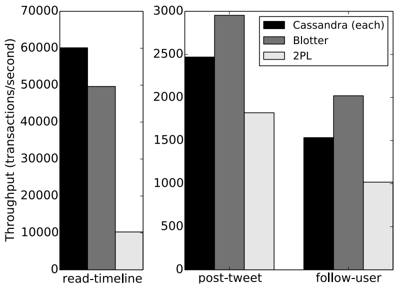


Figure 5: Microblogging throughput

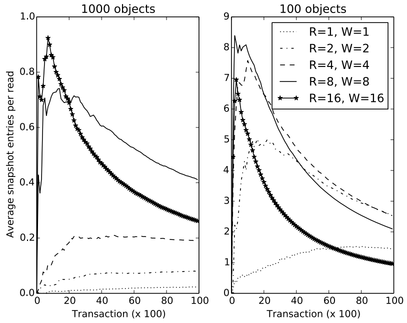


Figure 6: Garbage collection

keys. Each object in this deployment has a size of 256 Bytes. We used YCSB [8] to execute a total of $4,000$ transactions (evenly distributed across 20 client threads). Both configurations of Jessy exhibited a throughput below 100 transactions per second, while the throughput of Blotter was above 300 transactions per second. Even though the performance numbers are not directly comparable, since the code bases are very different, there are several factors that negatively affect the performance of Jessy. In particular, the size of its dependence vectors when using one entry per object grows to an average size of 200 entries only a few seconds into the experiment, which leads to significant memory and communication overheads. In turn, the other configuration of Jessy uses dependence vectors with a constant size of two entries. However, this alternative leads to a significant number of spurious conflicts.

We also deployed Jessy with geo-replication. However, both configurations of Jessy had very low throughput, which is a result using of total order multicast across data centers.

8. RELATED WORK

The closest related proposals are systems that also aim at supporting a non-monotonic version of snapshot isolation transactions in a geo-replicated setting. Jessy [25] was the first system to explore NMSI as the isolation level for transactions in a geo-replicated setting, where objects can be partitioned across different sites. For building consistent snapshots, Jessy uses a data structure called dependence vector, with one entry either per object or per set of objects. The former case has scalability issues, showed by our experiments, since the vector is $O(n)$, where n is the number of objects in the database, whereas in the latter case the algorithm restricts concurrency, leading to spurious conflicts. By focusing on the case where data is fully replicated, Blotter provides consistent snapshots with no spurious conflicts and with an overhead that is linear in the number of active transactions in the local data center.

Walter [28] also uses a non-monotonic variant of SI called parallel snapshot isolation (PSI). While both PSI and NMSI allow long forks, in PSI the snapshot is defined at transaction begin time, based on the state of the replica in which the transaction executes. Ardekani et. al. [24] have shown that this leads to a higher abort rate and results in lower performance when compared to NSMI.

There are also systems that guarantee isolation levels stronger than NMSI in geo-replicated scenarios. The replicated commit protocol [23] provides serializable transactions by layering Paxos on top of two-phase locking and two-phase commit. However, in that protocol, not only commit but also read operations require contacting all data centers and receiving replies from a majority of replicas. In Blotter, we similarly layer Paxos on top of a concurrency control protocol. But, in contrast, Blotter adopts the NMSI isolation level and uses novel concurrency control protocols that allow it to perform consistent reads locally, within the data center where the transaction was started. As a consequence, transactions in Blotter only require a single cross data center round-trip at commit time.

Spanner [9] and Scatter [13] provide strong ACID transactional guarantees with geo-replication, but, in contrast to Blotter, their architecture layers two-phase commit and two-phase locking on top of a Paxos-replicated log. Reversing the order of these two layers leads to more cross data center round-trips and a corresponding drop in performance, as shown by other authors [23].

TAPIR [29] follows the same principle as Blotter of having a lower layer of weakly consistent replication, on top of which transactions are built. However, TAPIR aims for stronger semantics, namely strict serializability. The resulting protocol requires loosely synchronized clocks at the clients (for performance, not correctness), and incurs in a single round-trip to all replicas in all shards that are part of the transaction. In contrast, we offer NMSI without any clock synchronization and with a single round-trip to the closest (or the master) data center.

Other systems also support transactions in a geo-replicated deployment, but provide weaker guarantees than Blotter. MDCC [16] provides read committed isolation without lost updates by combining different variants of Paxos [14]. In contrast to MDCC, Blotter offers stronger semantics, but the ideas of MDCC are complementary since Blotter could make use of a similar approach to further improve the geo-replicated commit.

Some systems like Megastore [3] or SQL Azure [6] provide ACID properties within a partition of data and looser consistency across partitions. In contrast, transactions in Blotter may span any set of objects. Other systems support more limited forms of transactions than Blotter. Eiger [22] supports read-only and write-only transactions. COPS [21] and ChainReaction [1] support read-only transactions that offer causality, but no isolation. In contrast, Blotter supports ACID transactions with NMSI.

9. CONCLUSION

In this paper, we studied the possibility of using NMSI to improve the performance of geo-replicated transactional systems. We proposed Blotter, a system that leverages this isolation level to introduce a set of new protocols. Our evaluation shows that Blotter outperforms other systems with stronger semantics, namely in terms of throughput. As such, our protocols may prove useful for systems that are performance critical and can run under NMSI.

Acknowledgments

Computing resources for this work were provided by an AWS in Education Research Grant. The research of R. Rodrigues is funded by the European Research Council (ERC-2012-StG-307732) and by FCT (UID/CEC/50021/2013). This work was partially supported by NOVA LINCS (UID/CEC/04516/2013) and EU H2020 LightKone project (732505).

10. REFERENCES

[1] S. Almeida, J. Leitão, and L. Rodrigues. Chain- reaction: A causal+ consistent datastore based on chain replication. In *Proc. of 8th European Conference on Computer Systems*, EuroSys'13, pages 85–98, 2013.

[2] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault- tolerant and scalable joining of continuous data streams. In *SIGMOD '13: Proc. of 2013 international conf. on Management of data*, pages 577–588, 2013.

[3] J. Baker, C. Bond, J. C. 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 *Proc. of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[4] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), January 1981.

[5] N. Bronson et al. Tao: Facebook's distributed data store for the social graph. In *Proc. of the 2013 USENIX Annual Technical Conference*, pages 49–60, 2013.

[6] D. G. Campbell, G. Kakivaya, and N. Ellis. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 1021–1024.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

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

[9] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.

[10] G. DeCandia et al. In *Proc. of the 21st ACM Symposium on Operating Systems Principles*, pages 205–220.

[11] A. K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Rec.*, 15(3):37–45, Sept. 1986.

[12] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, SRDS '05, pages 73–84, Washington, DC, USA, 2005. IEEE Computer Society.

[13] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, 2011.

[14] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, Mar. 2006.

[15] T. Hoff. Latency is everywhere and it costs you sales - how to crush it. Post at the High Scalability blog. http://tinyurl.com/5g8mp2, 2009.

[16] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, 2013.

[17] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

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

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

[20] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, Mar. 2010.

[21] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. In *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416.

[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 313–328, 2013.

[23] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013.

[24] M. Saeida Ardekani, P. Sutra, and M. Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Proc. of the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS 2013)*, pages 163–172, 2013.

[25] M. Saeida Ardekani, P. Sutra, M. Shapiro, and N. Preguiça. On the scalability of snapshot isolation. In *Euro-Par 2013 Parallel Processing*, volume 8097 of *LNCS*, pages 369–381. Springer, 2013.

[26] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[27] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, Aug. 2013.

[28] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, 2011.

[29] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 263–278, 2015.

[30] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, SOSP, pages 276–291, 2013.