2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

SSS: Scalable Key-Value Store with External Consistent and Abort-free Read-only Transactions

Masoomeh Javidi Kishi
CSE, Lehigh University
Bethlehem, PA, USA
maj717@lehigh.edu

Sebastiano Peluso
ECE, Virginia Tech
Blacksburg, VA, USA
peluso@vt.edu

Henry F. Korth
CSE, Lehigh University
Bethlehem, PA, USA
hfk2@lehigh.edu

Roberto Palmieri
CSE, Lehigh University
Bethlehem, PA, USA
palmieri@lehigh.edu

***Abstract*—We present SSS, a scalable transactional key-value store deploying a novel distributed concurrency control that provides external consistency for all transactions, never aborts read-only transactions due to concurrency, all without specialized hardware. SSS ensures the above properties without any centralized source of synchronization. SSS's concurrency control uses a combination of vector clocks and a new technique, called snapshot-queuing, to establish a single serialization order where transactions are guaranteed to read from the latest non-concurrent transaction externally visible to clients. We compare SSS against high performance key-value stores, Walter, ROCOCO, and a two-phase commit baseline. SSS outperforms 2PC-baseline by as much as 7x using 20 nodes; and ROCOCO by as much as 2.2x with long read-only transactions using 15 nodes.**

***Index Terms*—Transactions, Distributed Database, Consistency**

I. Introduction

A distributed transactional system that ensures a strong level of consistency greatly simplifies programmer responsibility while developing applications. A strong level of consistency that clients interacting with a transactional system often desire is referred to *external consistency* [8], [13], [19].

Roughly, under external consistency a distributed system behaves as if all transactions were executed sequentially, all clients observed the same unique order of transactions completion (also named *external schedule* in [19]), in which every read operation returns the value written by the latest write operation. By relying on the definition of external consistency, a transaction terminates when its execution is returned to its client; therefore the order defined by transaction client returns matches the transaction serialization order.

The latter property carries one great advantage: if clients communicate with each other outside the system before or after the transactional execution, they cannot be confused about the possible mismatch between transaction order they observe and the transaction serialization order provided by the concurrency control inside the system. Simply, if a transaction T is returned to its client, the serialization order of T will be *i)* after any other transaction T' that returned to its client before T started, and *ii)* before any transaction T'' that will start after T and will return to its client subsequently. This correctness criterion is also known as *strict serializability* if we restrict the consideration to only transaction's begin and commit events.

To picture the value of external consistency, consider an online document sharing service and two clients, C_1 connected to server N_1 and C_2 connected to another server N_2, whose goal is to synchronize the same document D. Let us assume D is replicated on two nodes for availability. C_1 modifies D and performs its synchronization. After C_1 is notified about the completion of its synchronization operation, it informs C_2 that its modifications are permanent. Now C_2 queries D to observe the changes. Since the underlying distributed system is asynchronous and clients C_1 and C_2 are on two different nodes, they cannot observe a shared timeline, therefore operations on the two replicas of D can arrive in opposite orders, which might cause clients to observe different serialization orders [9], [36]. Only if the service is external consistent, then C_2's expectation is met (i.e., C_2 observes the modification of C_1); otherwise the possible outcomes include the case where C_2 does not observe C_1, which might confuse C_2. Note that if the service provides Serializable [6] operations, C_2 will not be guaranteed to observe the outcome of C_1.

In this paper we present *SSS*, a key-value store that implements a novel distributed concurrency control providing external consistency and assuming off-the-shelf hardware. Two features enable high performance and scalability in SSS, especially in read-dominated workloads:

- SSS supports read-only transactions that never abort due to concurrency, therefore the return value of all their read operations should be consistent at the time the operation is issued. We name them as *abort-free* hereafter. This property is very appealing because many real-world applications produce significant read-only workload [4].
- SSS provides availability and fault tolerance by deploying a general partial replication scheme where each key is replicated on multiple nodes without predefined partitioning schemes (e.g., sharding [20], [38]). To favor scalability, SSS does not rely on ordering communication primitives, such as Total Order Broadcast or Multicast [16].

The core components that make the above properties possible in SSS are the following:

- SSS uses a vector clock-based technique to track dependent events originated on different nodes. This technique is similar to the one used by existing distributed transactional systems, such as Walter [32] and GMU [29], and allows SSS to track events without a global source of synchronization.

- SSS uses a new technique, which we name *snapshot-queuing*, that works as follows. Each key is associated with a *snapshot-queue*. Only transactions that will surely commit are inserted into the snapshot-queues of their accessed keys in order to leave a trace of their existence to other concurrent transactions. Read-only transactions are inserted into their read keys' snapshot-queues at read time, while update transactions into their modified keys' snapshot-queues after the commit decision is reached. Only update transactions can wait for read-only transactions if they belong to the same *snapshot-queue*. Read-only transactions leverage their membership into *snapshot-queue* to inform update transactions. This technique is similar to the one in [7] where readers leave a trace of their execution for subsequent update transactions.

  A transaction in a snapshot-queue is inserted along with a scalar value, called *insertion-snapshot*. This value represents the latest snapshot visible by the transaction on the node storing the accessed key, at the time the transaction is added to the snapshot queue. SSS concurrency control orders transactions with lesser insertion-snapshot before conflicting transactions with higher insertion-snapshot in the external schedule.

SSS uses snapshot-queues to propagate established serialization orders among concurrent transactions as follows.

If a read-only transaction T_R reads a key x subsequently modified by a concurrent committed transaction T_W, x's snapshot-queue is the medium to record the existence of an established serialization order between T_R and T_W. With that, any other concurrent transaction accessing x can see this established order and define its serialization accordingly. In addition, T_W's client response is delayed until T_R completes its execution. This delay is needed so that other update transactions can be serialized along with read-only transactions in a unique order where reads always return values written by the last update transaction that returned to its client before.

Failing in delaying T_W's response would result in a discrepancy between the external order and the transaction serialization order. In fact, the external order would show T_W returning earlier than T_R but T_R is serialized before T_W.

For non-conflicting update transactions that have dependencies with concurrent read-only transactions accessing common keys, since these transactions are aware of each other through the snapshot-queues of accessed keys, SSS prevents read-only transactions to observe those update transactions in different orders. This problem was previously discovered in [2], [29]).

On the flip side, delaying update transactions might have a domino effect on limiting the level of concurrency in the system, which might lead to poor performance. The snapshot-queue technique prevents that: it permits a transaction that is in a snapshot-queue to expose its written keys to other transactions while it is waiting for the completion of the concurrent read-only transaction(s) holding it. This feature enables progress of subsequent conflicting transactions, hence retaining the high throughput of the system.

Update transactions in SSS are serialized along with read-only transactions. They always read the latest version of a key and buffer write operations. Validation is performed at commit time to abort if some read key has been overwritten meanwhile. The Two-Phase Commit protocol (2PC) [8], [10], [29], [32], [37] is used to atomically lock and install written keys. These keys are externally visible when no concurrent read-only transactions caused the update transaction to wait due to snapshot-queuing, if any.

We implement SSS in Java and compared against two recent key-value stores, Walter [32] and ROCOCO [26], and one baseline where all transactions, including read-only, validate read keys and use 2PC to commit [6]. We name this competitor 2PC-baseline. Overall, SSS is up to 7$\times$ faster than 2PC-baseline and up to 2.2$\times$ faster than ROCOCO under read-dominated workloads and long (i.e., 16 read keys) read-only transactions. Also, when the percentage of read-only transactions is dominant and the node count is high, SSS is only 18% slower than Walter, which provides a weaker isolation level than external consistency and even serializability. When compared to the overall update transaction latency, in our experiments we assessed in less than 28% the average waiting time introduced by SSS due to the snapshot-queuing.

This paper makes the following contributions:

- SSS implements the first distributed transactional protocol for general purpose replicated systems where read-only transactions read consistently the latest committed version of objects without relying on a single synchronization service and without aborting.
- SSS's synchronization technique to serialize read-only and update transactions is the first to merge the semantics of vector clocks with visible read operations to produce the snapshot-queuing technique.
- SSS solves the problem of serializing two non-conflicting update transactions in different orders when multiple conflicting read-only transactions execute on different nodes [2], without relying on a single synchronization service and without aborting the read-only transactions.

II. System Model & Assumptions

SSS assumes a system as a set of nodes that do not share either memory or a global clock. Nodes communicate through message passing and reliable asynchronous channels, meaning messages are guaranteed to be eventually delivered unless a crash happens at the sender or receiver node. There is no assumption on the speed and on the level of synchrony among nodes. We consider the classic crash-stop failure model: sites may fail by crashing, but do not behave maliciously. A site that never crashes is correct; otherwise it is faulty. Clients are assumed to be colocated with nodes in the system; this way a client is immediately notified of a transaction's commit or abort outcome, without additional delay. Clients are allowed to interact with each other while they are not executing transactions through channels that are not provided by the system's APIs.

Data Organization. Every node N_i maintains shared objects (or keys) adhering to the key-value model [29]. Multiple versions are kept for each key. Each version stores the value and the commit vector clock of the transaction that produced the version. SSS does not make any assumption on the data clustering policy; simply every shared key can be replicated in one or more nodes, depending upon the chosen replication degree. For object reachability, SSS implements a local look-up function using consistent hashing, a commonly used technique to map keys with nodes [30].

Transaction execution. We model transactions as a sequence of read and write operations on shared keys, preceded by a begin, and followed by a commit or abort operation. A client begins a transaction on the colocated node and the transactions can read/write data belonging to any node; no a-priori knowledge on the accessed keys is assumed. SSS's concurrency control ensures the ACID properties and targets applications with a degree of data replication.

Every transaction starts with a client submitting it to the system, and finishes its execution informing the client about its final outcome: *commit* or *abort*. Transactions that do not execute any write operation are called read-only, otherwise they are update transactions. SSS requires programmer to identify whether a transaction is update or read-only.

III. SSS Concurrency Control

In this section we describe the SSS concurrency control, followed by two execution examples.

A. Metadata

Transaction vector clocks. In SSS a transaction T holds two vector clocks, whose size is equal to the number of nodes in the system. One represents its actual dependencies with transactions on other nodes, called `T.VC`; the other records the nodes where the transaction read from, called `T.hasRead`.

`T.VC` represents a version visibility bound for T. Once a transaction begins in node N_i, it assigns the vector clock of the latest committed transaction in N_i to its own `T.VC`. Every time T reads from a node N_j for the first time during its execution, `T.VC` is modified based on the latest committed vector clock visible by T on N_j. After that, `T.hasRead[j]` is set to true.

Transaction read-set and write-set. Every transaction holds two private buffers. One is rs (or *read-set*), which stores the keys read by the transaction during its execution, along with their value. The other buffer is ws (or *write-set*), which contains the keys the transaction wrote, along with their value.

Snapshot-queue. A fundamental component allowing SSS to establish a unique external schedule is the snapshot-queuing technique. With that, each key is associated with an ordered queue (`SQueue`) containing: read-only transactions that read that key; and update transactions that wrote that key while a read-only transaction was reading it.

Entries in a snapshot-queue (`SQueue`) are in the form of tuples. Each tuple contains: transaction identifier $T.id$, the *insertion-snapshot*, and transaction type (read-only or update). In general, the *insertion-snapshot* for a transaction T enqueued on some node N_i's snapshot-queue is the value of T's vector clock in position i^{th} at the time T is inserted in the snapshot-queue (see Section III-B and III-C for the actual value of the insertion snapshot, which varies depending upon the transaction type). Transactions in a snapshot-queue are ordered according to their insertion-snapshot.

A snapshot-queue contains only transactions that will commit; in fact, besides read-only transactions that are abort-free, update transactions are inserted in the snapshot-queue only after their commit decision has been reached.

Transaction transitive anti-dependencies set. An update transaction maintains a list of snapshot-queue entries, named `T.PropagatedSet`, which is populated during the transaction's read operations. This set serves the purpose of propagating anti-dependencies previously observed by conflicting update transactions.

Node's vector clock. Each node N_i is associated with a vector clock, called `NodeVC`. The i^{th} entry of `NodeVC` is incremented when N_i is involved in the commit phase of a transaction that writes some key replicated by N_i. The value of j^{th} entry of `NodeVC` in N_i is the value of the j^{th} entry of `NodeVC` in N_j at the latest time N_i and N_j cooperated in the commit phase of a transaction.

Commit repositories. `CommitQ` is an ordered queue, one per node, which is used by SSS to ensure that non-conflicting update transactions are ordered in the same way on the nodes where they commit. `CommitQ` stores tuples $<T, vc, s>$ with the following semantics. When an update transaction T, with commit vector clock vc, enters its commit phase, it is firstly added to the `CommitQ` of the nodes participating in its commit phase with its status s set as *pending*.

When the transaction commit phase concludes successfully, the status of the transaction is changed to *ready*. A ready transaction inside the `CommitQ` is assigned with a final vector clock produced during the commit phase. In each node N_i, transactions are ordered in the `CommitQ` according to the i^{th} entry of the vector clock ($vc[i]$). This allows them to be committed in N_i with the order given by $vc[i]$. Although the commitment of non-conflicting transactions in a sequential way on a node might reduce performance, it is indeed needed to guarantee a single serialization order with respect to the nodes replicating the same keys [29].

When T commits, it is deleted from `CommitQ` and its vc is added to a per node repository, named `NLog`. We identify the most recent vc in the `NLog` as `NLog.mostRecentVC`.

Overall, the presence of additional metadata to be transferred over the network might appear as a barrier to achieve high performance. To alleviate these costs we adopt metadata compression. In addition, while acknowledging that the size of vector clocks grows linearly with the system size, there are existing orthogonal solutions to increase the granularity of such a synchronization to retain efficiency [24], [35].

B. Execution of Update transactions

Update transactions in SSS implements lazy update [34], meaning their written keys are not immediately visible and accessible at the time of the write operation, but they are logged into the transactions write-set and become visible only at commit time. In addition, transactions record the information associated with each read key into their read-set.

Read operations of update transactions in SSS simply return the most recent version of their requested keys (Lines 24-27 of Algorithm 6). At commit time, validation is used to verify that all the read versions have not been overwritten.

An update transaction that completes all its operations and commits cannot inform its client if it observes anti-dependency with one or more read-only transactions. In order to capture this waiting stage, we introduce the following phases to finalize an update transaction (Figure 1 pictures them in a running example).

Internal Commit. When an update transaction successfully completes its commit phase, we say that it commits internally. In this stage, the keys written by the transactions are visible to other transactions, but its client has not been informed yet about the transaction completion. Algorithms 1 and 2 show the steps taken by SSS to commit a transaction internally.

SSS relies on the Two-Phase Commit protocol (2PC) to internally commit update transactions. The node that carries the execution of a transaction T, known as its coordinator, initiates 2PC issuing the prepare phase, in which it contacts all nodes storing keys in the read-set and write-set. When a participant node N_i receives a prepare message for T, all keys read/written by T and stored by N_i are locked. If the locking acquisition succeeds, all keys read by T and stored by N_i are validated by checking if the latest version of a key matches the read one (Lines 28-34 Algorithm 1). If successful, N_i replies to T's coordinator with a `Vote` message, along with a proposed commit vector clock. This vector clock is equal to N_i's $NodeVC$ where $NodeVC[i]$ has been incremented. Finally, T is inserted into N_i's `CommitQ` with its $T.VC$.

After receiving each successful `Vote`, T's coordinator computes the commit vector clock (*commitVC*) by calculating the maximum per entry (Line 18 of Algorithm 1). This update makes T able to include the causal dependencies of the latest committed transactions in all 2PC participants. After receiving all `Vote` messages, the coordinator determines the final commit vector clock for T as in (Lines 18-24 of Algorithm 1), and sends it along with the 2PC `Decide` message.

Lines 16-26 of Algorithm 2 shows how 2PC participants handle the `Decide` message. When N_i receives `Decide` for transaction T, N_i's $NodeVC$ is updated by computing the maximum with *commitVC*. Importantly, at this stage the order of T in the $CommitQ$ of N_i might change because the final commit vector clock of T has been just defined, and it might be different from the one used during the 2PC prepare phase when T has been added to $CommitQ$.

In Algorithm 2 Lines 27-34, when transaction T becomes the top standing of N_i's $CommitQ$, the internal commit of T is completed by inserting its commit vector clock into the

Algorithm 1 Internal Commit by Transaction T in node N_i

```
 1: upon boolean Commit(Transaction T) do
      // Check if T is a read-only transaction
 2:   if (T.ws=$\phi$) then
 3:     for ($k \in T.rs$) do
 4:       Send Remove[$T$] to all replicas(k)
 5:     end for
 6:     $T.outcome \leftarrow true$
 7:     return $T.outcome$
 8:   end if
      // Start 2PC if T is an update transaction
 9:   $commitVC \leftarrow T.VC$
10:   $T.outcome \leftarrow true$
11:   send Prepare[$T$] to all $N_j \in replicas(T.rs \cup T.ws) \cup N_i$
12:   for all ($N_j \in replicas(T.rs \cup T.ws) \cup N_i$) do
13:     wait receive Vote[$T.id, VC_j, res$] from $N_j$ or timeout
      // Check if T's 2PC commit decision was successful
14:     if ($\neg res \vee timeout$) then
15:       $T.outcome \leftarrow false$
16:       break;
17:     else
18:       $commitVC \leftarrow \max(commitVC, VC_j)$
19:     end if
20:   end for
21:   $xactVN \leftarrow \max\{commitVC[w] : N_w \in replicas(T.ws)\}$
      // Finalize T's commit vector clock
22:   for all ($N_j \in replicas(T.ws)$) do
23:     $commitVC[j] \leftarrow xactVN$
24:   end for
25:   send Decide[$T, commitVC, outcome$] to all $N_j \in replicas($
        $T.rs \cup T.ws) \cup N_i$
26:   return $T.outcome$
27: end

28: boolean validate(Set rs, VC T.VC)
      // Check if T's read keys are not overwritten
29: for all ($k \in rs$) do
30:   if ($k.last.vc[i] > T.VC[i]$) then
31:     return $false$
32:   end if
33: end for
34: return $true$
```

$NLog$ and removing T from $CommitQ$. When transaction's vector clock is inserted into the node's `NLog`, its written keys become accessible by other transactions. At this stage, T's client has not been informed yet about T's internal commit.

Pre-Commit. An internally committed transaction spontaneously enters the Pre-Commit phase after that. Algorithm 3 shows detail of Pre-commit phase. At this stage, T evaluates if it should hold the reply to its client depending upon the content of the snapshot-queues of its written keys. If so, T will be inserted into the snapshot-queue of its written keys in N_i with $commitVC[i]$ as *insertion-snapshot*.

If at least one read-only transaction (T_{ro}) with a lesser *insertion-snapshot* is found in any snapshot-queue $SQueue$ of T's written keys, it means that T_{ro} read that key before T_w internally committed, therefore a write-after-read dependency between T_{ro} and T_w is established. In this case, T is inserted into $SQueue$ until T_{ro} returns to its client. With the anti-dependency, the transaction serialization order has been established with T_{ro} preceding T_w. Informing immediately T_w's client about T_w's completion would expose an external order where T_w is before T_{ro}, which might violates external consistency if another non-conflicting update transaction T'_w is observed by a conflicting read-only transaction T'_{ro} in a different serialization order (e.g., the case in Figure 2).

Tracking only non-transitive anti-dependencies is not

Algorithm 2 Internal Commit by Transaction T in node N_i

```
1: upon receive Prepare[$Transaction\ T$] from $N_j$ do
   // Check if T passes lock acquisition and validation
2:     boolean $outcome \leftarrow getExclusiveLocks(T.id, T.ws)$
       $\wedge getSharedLocks(T.id, T.rs) \wedge validate(T.rs, T.VC)$
3:     if ($\neg outcome$) then
4:         $releaseLocks(T.id, T.rs, T, ws)$
5:         send Vote[$T.id, T.VC, outcome$] to $N_j$
6:     else
7:         $prepVC \leftarrow NLog.mostRecentVC$
8:         if ($N_i \in replicas(T.ws)$) then
9:             $NodeVC[i] + +$
10:            $prepVC \leftarrow NodeVC$
11:            $CommitQ.put(< T, prepVC, pending >)$
12:        end if
13:        send Vote[$T.id, prepVC, outcome$] to $N_j$
14:    end if
15: end
16: upon receive Decide[$T, commitVC, outcome$] from $N_j$ atomically do
17:     if ($outcome$) then
    // Update NodeVC and CommitQ if T is decided to commit
18:         $NodeVC \leftarrow \max(NodeVC, commitVC)$
19:         if ($N_i \in replicas(T.ws)$) then
20:             $CommitQ.update(< T, commitVC, ready >)$
21:         end if
22:     else
23:         $CommitQ.remove(T)$
24:         $releaseLocks(T.id, T.ws, T.rs)$
25:     end if
26: end
27: upon $\exists < T, vc, s >:< T, vc, s >= commitQ.head \wedge s = ready$ do
    // Finalize internal commit of T
28:     for all ($k \in T.ws : N_i \in replicas(k)$) do
29:         apply(k,val,vc)
30:     end for
31:     $NLog.add(< vc >)$
32:     $CommitQ.remove(T)$
33:     $releaseLocks(T.id, T.ws, T.rs)$
34: end
```

Algorithm 3 Start Pre-commit by Transaction T in node N_i

```
1: for all ($k \in T.ws$) do
2:     if ($N_i \in replicas(k)$) then
3:         $k.SQueue.insert(< T.id, T.commitVC[i], \text{“W”} >)$
4:         for all ($T^{'} \in T.PropagatedSet$) do
5:             $k.SQueue.insert(< T^{'}.id, T^{'}.snapshot, \text{“R”} >)$
6:         end for
7:     end if
8: end for
```

enough to preserve correctness. If T reads the update done by $T_{w'}$ and $T_{w'}$ is still in its Pre-commit phase, then T has a transitive anti-dependency with $T_{ro'}$ (i.e., $T_{ro'} \xrightarrow{\text{rw}} T_{w'} \xrightarrow{\text{wr}} T$). SSS records the existence of transactions like $T_{ro'}$ during T's execution by looking into the snapshot-queues of T's read keys and logging them into a private buffer of T, called $T.PropagatedSet$. The propagation of anti-dependency happens during T's Pre-commit phase by inserting transactions in $T.PropagatedSet$ into the snapshot-queues of all T's written keys (Lines 4-6 of Algorithm 3).

External Commit. Transaction T remains in its Pre-commit phase until there is no read-only transaction with lesser insertion-snapshot in the snapshot-queues of T's written keys. After that, T is removed from these snapshot-queues and an `Ack` message to the transaction 2PC coordinator is sent (Lines 1-7 of Algorithm 4).

The coordinator can inform its client after receiving `Ack` from all 2PC participants. At this stage, update transaction's external schedule is established, therefore we say that SSS

Algorithm 4 End Pre-commit of Transaction T in node N_i

```
1: for all ($k \in T.ws$) do
2:     if $N_i \in replicas(k)$ then
   // T waits for all existing anti-dependent transactions to be removed from snapshot-queues of T.ws.
3:         wait until ($\exists < T^{'}.id, T^{'}.snapshot, - >$:
           $k.SQueue.contains(< T^{'}.id, T^{'}.snapshot, - >)\wedge$
           $T^{'}.snapshot < T.commitVC[i]$)
4:         $k.SQueue.remove(< T.id, T.commitVC[i], \text{“W”} >)$
5:         send Ack [$T, vc[i]$] to $T.coordinator$
6:     end if
7: end for
```

commits the update transaction *externally*.

C. Execution of Read-Only Transactions

In its first read operation (Algorithm 5 Lines 5-7), a read-only transaction T on N_i assigns `NLog.mostRecentVC` to its vector clock ($T.VC$). This way, T will be able to see the latest updated versions committed on N_i. Read operations are implemented by contacting all nodes that replicate the requested key and waiting for the fastest to answer.

When a read request of T returns from node N_j, T sets $T.hasRead[j]$ to `true`. With that, we set the visibility upper bound for T from N_j (i.e., $T.VC[j]$). Hence, subsequent read operations by T contacting a node N_k should only consider versions with a vector clock vc_k such that $vc_k[j] < T.VC[j]$.

After a read operation returns, the transaction vector clock is updated by applying an entry-wise maximum operation between the current $T.VC$ and the vector clock associated with the read version (i.e., VC^*) from N_j. Finally, the read value is added to $T.rs$ and returned.

Algorithm 6 shows SSS rules to select the version to be returned upon a read operation that contacts node N_i.

The first time N_i receives a read from T, this request should wait until the value of N_i's `NLog.mostRecentVC[i]` is equal to $T.VC[i]$ (Line 5 Algorithm 6). This means that all transactions that are already included in the current visibility bound of $T.VC[i]$ must perform their internal commit before T's read request can be handled.

After that, a correct version of the requested key should be selected for reading. This process starts by identifying the set of versions that are within the visibility bound of T, called $VisibleSet$. This means that, given a version v with commit vector clock vc, v is visible by T if, for each entry k such that $T.hasRead[k] = true$, we have that $vc[k] \leq T.VC[k]$ (Algorithm 6 Line 6).

It is possible that transactions associated with some of these vector clocks are still in their Pre-Commit phase, meaning they exist in the snapshot-queues of T's requested key. If so, they should be excluded from $VisibleSet$ in case their insertion-snapshot is higher than $T.VC[i]$. The last step is needed to serialize read-only transactions with anti-dependency relations before conflicting update transactions.

This condition is particularly important to prevent a well-known anomaly, firstly observed by Adya in [2], in which read-only transactions executing on different nodes can observe two non-conflicting update transactions in different serialization

Algorithm 5 Read Operation by Transaction T in node N_i

```
1:  upon Value Read (Transaction T, Key k) do
2:      if ($\exists < k, val > \in T.ws$) then
3:          return val
4:      end if
    // T's vector clock is initialized with the latest committed vector clock in $N_i$
5:      if ($is\ first\ read\ of\ T$) then
6:          $T.VC \leftarrow NLog.mostRecentVC$
7:      end if
8:      target ← {replicas(k)}
    // $isUpdate$ is a boolean showing whether T is read-only or update
9:      send READREQUEST$[k, T.VC, T.hasRead, T.isUpdate]$
        to all $N_j \in target$
10:     wait Receive READRETURN $[val, VC^*, PropagatedSet]$
        from $N_h \in target$
11:     $T.hasRead[h] \leftarrow true$
12:     $T.VC \leftarrow \max(T.VC, VC^*)$
13:     $T.rs \leftarrow T.rs \cup \{< k, val >\}$
14:     $T.PropagatedSet \leftarrow T.PropagatedSet \cup PropagatedSet$
15:     return val
16: end
```

Algorithm 6 Version Selection Logic in node N_i

```
1:  upon Receive READREQUEST$[T, k, T.VC, hasRead, isUpdate]$ from $N_j$
    do
2:      $PropagatedSet \leftarrow \phi$
3:      if ($\neg isUpdate$) then
4:          if (¬hasRead[i]) then
5:              wait until NLog.mostRecentVC[i] ≥ T.VC[i]
6:              $VisibleSet \leftarrow \{vc : vc \in$ NLog$\wedge$
                 $\forall w($hasRead[w] $\Rightarrow$ vc[w] $\leq$ T.VC[w]$)\}$
7:              $ExcludedSet \leftarrow \{T' :< T'.id, T'.snapshot,$ "W" $> \in$
                 $k.SQueue \Rightarrow T'.snapshot > T.VC[i])\}$
8:              $VisibleSet \leftarrow VisibleSet \backslash ExcludedSet$
9:              $maxVC \leftarrow vc : \forall w, vc[w] = \max\{v[w] : v \in VisibleSet\}$
10:             $k.SQueue.insert(< T.id, maxVC[i],$ "R" $>)$
11:             $ver \leftarrow k.last$
12:             while ($\exists w : hasRead[w] \wedge ver.vc[w] >$
                   $maxVC[w] \vee \exists vc \in ExcludedSet : ver.vc = vc$
                   $\wedge vc[i] > maxVC[i]$) do
13:                 $ver \leftarrow ver.prev$
14:             end while
15:         else
16:             $maxVC \leftarrow T.VC$
17:             $k.SQueue.insert(< T.id, maxVC[i],$ "R" $>)$
18:             $ver \leftarrow k.last$
19:             while ($\exists w$ : (hasRead[w] $\wedge$ $ver.vc$[w] $> maxVC[w]$)) do
20:                 $ver \leftarrow ver.prev$
21:             end while
22:         end if
23:     else
24:         $maxVC \leftarrow NLog.mostRecentVC$
25:         $PropagatedSet$=$\{T'$:<$T'.id$,$T'.snapshot$,"R">$\in k.SQueue\}$
26:         $ver \leftarrow k.last$
27:     end if
28:     Send READRETURN[ver.val, maxVC,PropagatedSet ] to $N_j$
29: end
```

order [29]. Consider a distributed system where nodes do not have access to a single point of synchronization (or an ordering component), concurrent non-conflicting transactions executing on different nodes cannot be aware of each other's execution. Because of that, different read-only transactions might order these non-conflicting transactions in a different way, therefore breaking the client's perceived order. SSS prevents that by serializing both these read-only transactions before those update transactions.

At this stage, if multiple versions are still included in $VisibleSet$, the version with the maximum $VC[i]$ should be selected to ensure external consistency.

Once the version to be returned is selected, T is inserted in the snapshot-queue of the read key using $maxVC[i]$ as insertion-snapshot (Line 10 of Algorithm 6). Finally, when the read response is received, the maximum per entry between $maxVC$ (i.e., VC^* in Algorithm 5) and the $T.VC$ is computed along with the result of the read operation.

When a read-only transaction T commits, it immediately replies to its client. After that, it sends a message to the nodes storing only the read keys in order to notify its completion. We name this message `Remove`. Upon receiving `Remove`, the read-only transaction is deleted from all the snapshot-queues associated with the read keys. Deleting a read-only transaction from a snapshot-queue enables conflicting update transactions to be externally committed and their responses to be released to their clients.

Because of transitive anti-dependency relations, a node might need to forward the `Remove` message to other nodes as follows. Let us assume T has an anti-dependency with a transaction T_w, and another transaction $T_{w'}$ reads from T_w. Because anti-dependency relations are propagated along the chain of conflicting transactions, T exists in the snapshot-queues of T_w's and $T_{w'}$'s written keys. Therefore, upon `Remove` of T, the node executing T_w is responsible to forward the `Remove` message to the node where $T_{w'}$ executes for updating the affected snapshot-queues.

When a read operation is handled by a node that already responded to a previous read operation from the same transaction, the latest version according to $maxVC$ is returned, and T can be inserted into the snapshot-queue with its corresponding identifier and $maxVC[i]$ as insertion-snapshot.

D. Examples

External Consistency & Anti-dependency. Figure 1 shows an example of how SSS serializes an update transaction T_1 in the presence of a concurrent read-only transaction T_2. Two nodes are deployed, N_1 and N_2, and no replication is used for simplicity. T_1 executes on N_1 and T_2 on N_2. Key y is stored in N_2's repository. The `NLog.mostRecentVC` for Node 1 is {5,4} and for Node 2 is {3,7}.

T_1 performs a read operation on key y by sending a remote read request to N_2. At this point, T_1 is inserted in the snapshot-queue of y ($Q(y)$) with 7 as insertion-snapshot. This value is the second entry of N_2's `NLog.mostRecentVC`. Then the update transaction T_2 begins with vector clock {3,7}, buffers its write on key y in its write-set, and performs its internal commit by making the new version of y available, and by inserting the produced commit vector clock (i.e., $T2$.*commitVC*={3,8}) in N_2's $NLog$. As a consequence of that, $NLog.mostRecentVC$ is equal to $T2.commitVC$.

Now T_2 is evaluated to decide whether it should be inserted into $Q(y)$. The insertion-snapshot of T_2 is equal to 8, which is higher than T_1's insertion-snapshot in $Q(y)$. For this reason, T_2 is inserted in $Q(y)$ and its Pre-commit phase starts.

At this stage, T_2 is still not externally visible. Hence T_2 remains in its Pre-Commit phase until T_1 is removed from $Q(y)$, which happens when T_1 commits and sends the `Remove` message to N_2. After that, T_2's client is informed about T_2's completion. Delaying the external commit of T_2

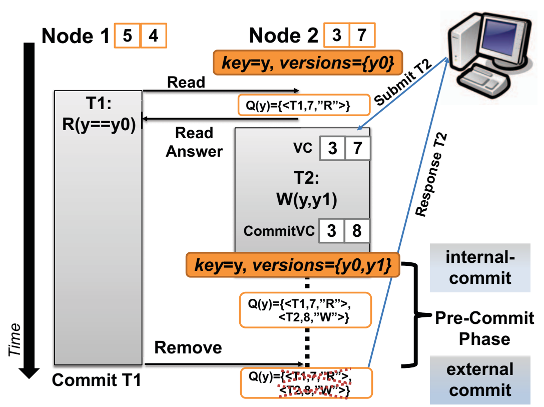


Fig. 1. SSS execution in the presence of an anti-dependency. Orange boxes show the content of the data store. Gray boxes show transaction execution. Dashed line represents the waiting time for $T2$. The red crossed entries of Q(y) represent their elimination upon Remove.

prevents clients from observing the internal completion of T_2, until T_1 returns to its client.

External Consistency & Non-conflicting transactions. Figure 2 shows how SSS builds the external schedule in the presence of read-only transactions and non-conflicting update transactions. There are four nodes, N_1, N_2, N_3, N_4, and four concurrent transactions, T_1, T_2, T_3, T_4, each executes on the respective node. By assumption, T_2 and T_3 are non-conflicting update transactions, while T_1 and T_4 are read-only.

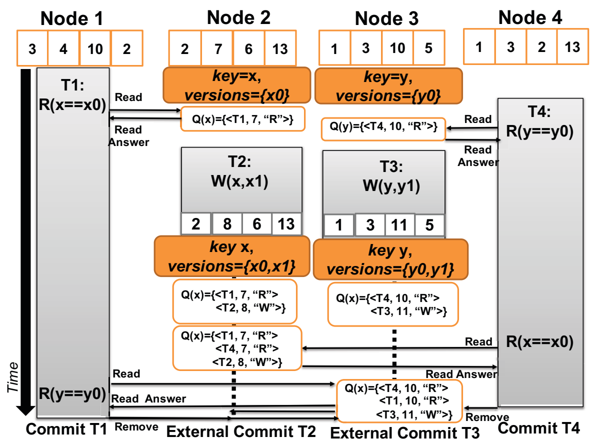


Fig. 2. Handling read-only transactions along with non-conflicting update transactions. We omitted snapshot-queue entries elimination upon Remove to improve readability.

SSS ensures that T_1 and T_4 do not serialize T_2 and T_3 in different orders and they return to their clients in the same way they are serialized by relying on snapshot-queuing. T_1 is inserted into $Q(x)$ with insertion-snapshot equals to 7. Concurrently, T_4 is added to the snapshot-queue of y with insertion-snapshot equals to 10. The next read operation by T_1 on y has two versions evaluated to be returned: $y0$ and $y1$. Although $y1$ is the most recent, since T_4 returned $y0$ previously (in fact T_4 is in $Q(y)$), $y1$ is excluded and $y0$ is returned. Similar arguments apply to T_4's read operation on x. The established external schedule serializes T_1 and T_4 before both T_2 and T_3.

E. Additional Considerations of SSS

Garbage Collection. A positive side effect of the `Remove` message is the implicit garbage collection of entries in the snapshot-queues. In fact, SSS removes any entry representing transactions waiting for a read-only transaction to finish upon receiving `Remove`, which cleans up the snapshot-queues.

Starvation. Another important aspect of SSS is the chance to slow down update transactions, possibly forever, due to an infinite chain of conflicting read-only transactions issued concurrently. We handle this corner case by applying admission control to read operations of read-only transactions in case they access a key written by a transaction that is in a snapshot-queue for a pre-determined time. In practice, if such a case happens, we apply an artificial delay to the read operation (exponential back-off) to give additional time to update transaction to be removed from the snapshot-queue. In the experiments we never experienced starvation scenarios, even with long read-only transactions.

Deadlock-Freedom. SSS uses timeout to prevent deadlock during the commit phase's lock acquisition. Also, the waiting condition applied to update transactions cannot generate deadlock. This is because read-only transactions never wait for each other, and there is no condition in the protocol where an update transaction blocks a read-only transaction. The only wait condition occurs when read-only transactions force update transactions to hold their client response due to snapshot-queuing. As a result, no circular dependency can be formed, thus SSS cannot encounter deadlock.

Fault-Tolerance. SSS deploys a protocol that tolerates failures in the system using replication. In the presented version of the SSS protocol, we did not include either logging of messages to recover update transactions' 2PC upon faults, or a consensus-based approach (e.g., Paxos-Commit [21]) to distribute and order 2PC messages. Solutions to make 2PC recoverable are well-studied. To focus on the performance implications of the distributed concurrency control of SSS and all its competitors, operations to recover upon a crash of a node involved in a 2PC have been disabled. This decision has no correctness implication.

IV. Correctness

Our target is proving that every history H executed by SSS, which includes committed update transactions and read-only transactions (committed or not), is external consistent.

We adopt the classical definition of history [2]. For understanding correctness, it is sufficient to know that a history is external consistent if the transactions in the history return the same values and leave the data store in the same state as they were executed in a sequential order (one after the other), and that order does not contradict the order perceived by clients, namely the precedence relations between non-concurrent transactions as observed by clients (similar to the real-time order relations [27] in strict serializability).

We decompose SSS's correctness in three statements, each highlighting a property guaranteed by SSS. Each statement claims that a specific history H', which is derived from H, is external consistent. In order to prove that, we rely on the characteristics of the Direct Serialization Graph (DSG) [2] which is derived from H'. Note that DSG also includes order relations between transactions' external commit. Every transaction in H' is a node of the DSG graph, and every dependency of a transaction T_j on a transaction T_i in H' is an edge from T_i to T_j in the graph. The concept of dependency is the one that is widely adopted in the literature: *i)* T_j read-depends on T_i if a read of T_j returns a value written by T_i, *ii)* T_j write-depends on T_i if a write of T_j overwrites a value written by T_i; *iii)* T_j anti-depends on T_i if a write of T_j overwrites a value previously read by T_i. We also map transactions relations as observed by clients to edges in the graph: if T_i commits externally before T_j starts, then the graph has an edge from T_i to T_j. A history H' is external consistent iff the DSG does not have any cycle [2], [6].

In our proofs we use the binary relation $\leq$ to define an ordering on pair of vector clocks v_1 and v_2 as follows: $v_1 \leq v_2$ if $\forall i,\ v_1[i] \leq v_2[i]$. Furthermore, if there also exists at least one index j such that $v_1[j] < v_2[j]$, then $v_1 < v_2$ holds.

Statement 1. *For each history H executed by SSS, the history H', which is derived from H by only including committed update transactions in H, is external consistent.*

In the proof we show that if there is an edge from transaction T_i to transaction T_j in DSG, then $T_i.commitVC < T_j.commitVC$. This statement implies that transactions modify the state of the data store as they were executed in a specific sequential order (provided by *CommitQ*), which does not contradict the transaction external commit order. Because no read-only transaction is included in H', the internal commit is equivalent to the external commit (i.e., no transaction is delayed). The formal proof is included in the technical report [23].

Statement 2. *For each history H executed by SSS, the history H', which is derived from H by only including committed update transactions and one read-only transaction in H, is external consistent.*

The proof shows that a read-only transaction always observes a consistent state by showing that in both the case of a direct dependency or anti-dependency, the vector clock of the read-only transaction is comparable with the vector clocks of conflicting update transactions. This statement implies that read operations of a read-only transaction always return values from a state of the data store as the transaction was executed atomically in a point in time that is not concurrent with any conflicting update transaction. The formal proof is included in the technical report [23].

Statement 3. *For each history H executed by SSS, the history H', which is derived from H by including committed update transactions and two or more read-only transactions in H, is external consistent.*

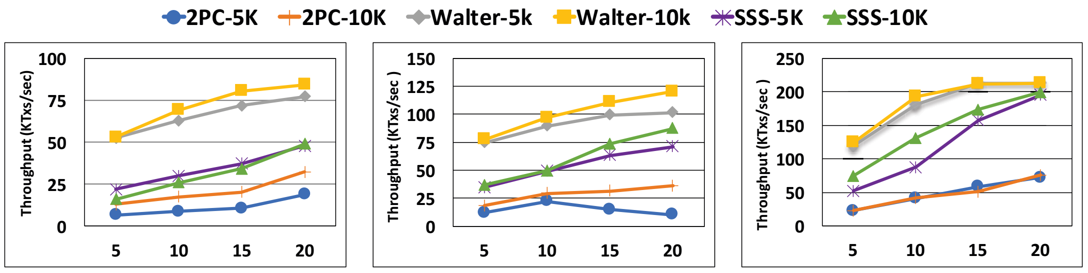
Since Statement 2 holds, SSS guarantees that each read-only transaction appears as it were executed atomically in a point in time that is not concurrent with any conflicting update. Furthermore, since Statement 1 holds, the read operations of that transaction return values of a state that is the result of a sequence of committed update transactions. Therefore, Statement 3 implies that, given such a sequence $S1$ for a read-only transaction T_{r1}, and $S2$ for a read-only transaction T_{r2}, either $S1$ is a prefix of $S2$, or $S2$ is a prefix of $S1$. In practice, this means that all read-only transactions have a coherent view of all transactions executed on the system. The formal proof is included in the technical report [23].

V. Evaluation

We implemented SSS in Java from the ground up and performed a comprehensive evaluation study. In the software architecture of SSS there is an optimized network component where multiple network queues, each for a different message type, are deployed. This way, we can assign priorities to different messages and avoid protocol slow down in some critical steps due to network congestion caused by lower priority messages (e.g., the `Remove` message has a very high priority because it enables external commits). Another important implementation aspect is related to snapshot-queues. Each snapshot-queue is divided into two: one for read-only transactions and one for update transactions. This way, when the percentage of read-only transactions is higher than update transactions, a read operation should traverse few entries in order to establish its visible-set.

We compare SSS against the following competitors: 2PC-baseline (2PC in the plots), ROCOCO [26], and Walter [32]. All these competitors offer transactional semantics over key-value APIs. With 2PC-baseline we mean the following implementation: all transactions execute as SSS's update transactions; read-only transactions validate their execution, therefore they can abort; and no multi-version data repository is deployed. As SSS, 2PC-baseline guarantees external consistency.

ROCOCO is an external consistent two-round protocol where transactions are divided into pieces and dependencies are collected to establish the execution order. ROCOCO classifies pieces of update transactions into immediate and deferrable. The latter are more efficient because they can be reordered. Read-only transactions can be aborted, and they are implemented by waiting for conflicting transactions to complete. Our benchmark is configured in a way all pieces are deferrable. ROCOCO uses preferred nodes to process transactions and consensus to implement replication. Such a scheme is different from SSS where multiple nodes are involved in the transaction commit process. To address this discrepancy, in the experiments where we compare SSS and ROCOCO, we disable replication for a fair comparison. The third competitor is Walter, which provides PSI a weaker isolation level than SSS. Walter has been included because it synchronizes nodes using vector clocks, as done by SSS.



(a) 20% (b) 50% (c) 80%

Fig. 3. Throughput varying % of read-only transactions. Number of nodes in X-axes.

All competitors have been re-implemented using the same software infrastructure of SSS because we want to provide all competitors with the same underlying code structure and optimization (e.g., optimized network). For fairness, we made sure that the performance obtained by our re-implementation of competitors matches the trends reported in [32] and [26], when similar configurations were used.

In our evaluation we use YCSB [12] benchmark ported to key-value store. We configure the benchmark to explore multiple scenarios. We have two transaction profiles: update, where two keys are read and written, and read-only transactions, where two or more keys are accessed. In all the experiments we co-locate application clients with processing nodes, therefore increasing the number of nodes in the system also increases the amount of issued requests. There are 10 application threads (i.e., clients) per node injecting transactions in the system in a closed-loop (i.e., a client issues a new request only when the previous one has returned). All the showed results are the average of 5 trials.

We selected two configurations for the total number of shared keys: 5k and 10k. We selected these ranges since they give us the appropriate level of contention on snapshot-queues in the case of 20% read-only transactions (write-dominated work load) and 80% read-only transactions (read-dominated work load). With the former, the observed average transaction abort rate is in the range of 6% to 28% moving from 5 nodes to 20 nodes when 20% read-only transactions are deployed. In the latter, the abort rate was from 4% to 14%. Unless otherwise stated, transactions select accessed objects randomly with uniform distribution.

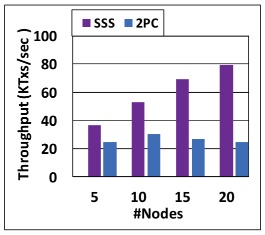
As test-bed, we used CloudLab [31], a cloud infrastructure available to researchers. We selected 20 nodes of type c6320 available in the Clemson cluster [1]. This type is a physical machine with 28 Intel Haswell CPU-cores and 256GB of RAM. Nodes are interconnected using 40Gb/s Infiniband HPC cards. In such a cluster, a network message is delivered in around 20 microseconds (without network saturation), therefore we set timeout on lock acquisition to 1ms.

In Figure 3 we compare the throughput of SSS against 2PC-baseline and Walter in the case where each object is replicated in two nodes of the system. We also varied the percentage of read-only transactions in the range of 20%, 50%, and 80%. As expected, Walter is the leading competitor in all the scenarios because its consistency guarantee is much weaker than external consistency; however, the gap between SSS and Walter reduces from 2× to 1.1× when read-only transactions become predominant (moving from Figure 3(a) to 3(c)). This is reasonable because in Walter, update transactions do not have the same impact in read-only transactions' performance as in SSS due to the presence of the snapshot-queues. Therefore, when the percentage of update transactions reduces, SSS reduces the gap. Considering the significant correctness level between PSI (in Walter) and external consistency, we consider the results of the comparison between SSS and Walter remarkable.

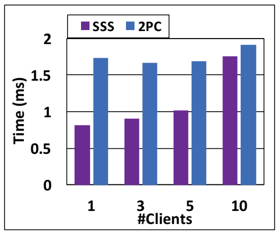
Performance of 2PC-baseline is competitive when compared with SSS only at the case of 20% read-only. In the other cases, although SSS requires a more complex logic to execute its read operations, the capability of being abort-free allows SSS to outperform 2PC-baseline by as much as 7× with 50% read-only and 20 nodes. 2PC-baseline's performance in both the tested contention levels become similar at the 80% read-only case because, although lock-based, read-only transaction's validation will likely succeed since few update transactions execute in the system.

Figure 3 also shows the scalability of all competitors. 2PC-baseline suffers from higher abort rate than others, which hampers its scalability. This is because its read-only transactions are not abort-free. The scalability trend of SSS and Walter is similar, although Walter stops scaling at 15 nodes using 80% of read-only transactions while SSS proceeds. This is mostly related with network congestion, which is reached by Walter earlier than SSS since Walter's transaction processing time is lower than SSS, thus messages are sent with a higher rate.

In Figure 4 we compare 2PC-baseline and SSS in terms of maximum attainable throughput and transaction latency. Figure 4(a) shows 2PC-baseline and SSS configured in a way they can reach their maximum throughput with 50% read-only workload and 5k objects, meaning the number of clients per nodes differs per reported datapoint. Performance trends are similar to those in Figure 3(b), but 2PC-baseline here is faster than before. This is related with the CPU utilization of the nodes' test-bed. In fact, 2PC-baseline requires less threads to execute, meaning it leaves more unused CPU-cores than SSS, and those CPU-cores can be leveraged to host more clients.



(a) Max attainable throughput.



(b) External Commit latency.

Fig. 4. Performance of SSS against 2PC-baseline using 5k objects and 50% read-only transactions.

The second plot (Figure 4(b)) shows transaction latency from its begin to its external commit when 20 nodes, 50% read-only transactions, and 5k objects are deployed. In the experiments we varied the number of clients per node from 1 to 10. When the system is far from reaching saturation (i.e., from 1 to 5 clients), SSS's latency does not vary, and it is on average 2× lower than 2PC-baseline's latency. At 10 clients, SSS's latency is still lower than 2PC-baseline but by a lesser percentage. This confirms one of our claim about SSS capability of retaining high-throughput even when update transactions are held in snapshot-queues. In fact, Figure 3(b) shows the throughput measurement in the same configuration: SSS is almost 7× faster than 2PC-baseline.

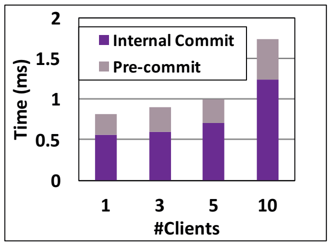
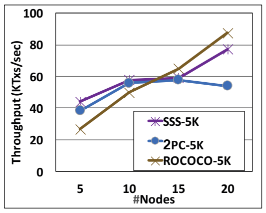


Fig. 5. Breakdown of SSS transaction latency.

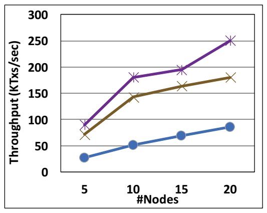
Figure 5 shows the relation between the internal commit latency and the external commit latency of SSS update transactions. The configuration is the one in Figure 4(b). Each bar represents the latency between a transaction begin and its external commit. The internal gray bar shows the time interval between the transaction's insertion in a snapshot-queue and its removal (i.e., from internal to external commit). This latter time is on average 30% of the total transaction latency.

In Figure 6 we compare SSS against ROCOCO and 2PC-baseline. To be compliant with ROCOCO, we disable replication for all competitors and we select 5k as total number of shared keys because ROCOCO finds its sweet spot in the presence of contention. Accesses are not local.

Figures 6(a) and 6(b) show the results with 20% and 80% read-only transactions respectively. In write intensive workload, ROCOCO slightly outperforms SSS due to its lock-free executions and its capability of re-ordering deferrable transaction pieces. However, even in this configuration, which matches a favorable scenario for ROCOCO, SSS is only 13%



(a) 20%.



(b) 80%.

Fig. 6. SSS, 2PC-baseline, ROCOCO varying % of read-only transactions. Legend in (a) applies to (b).

slower than ROCOCO and 70% faster than 2PC-baseline. In read-intensive workload, SSS outperforms ROCOCO by 40% and by almost 3× 2PC-baseline at 20 nodes. This gain is because ROCOCO is not optimized for read-only transactions; in fact, its read-only are not abort-free and they need to wait for all conflicting update transactions in order to execute. Also, since in YCBS transaction size is small, the overhead of ROCOCO's two-round commitment protocol is dominant.

We also configured the benchmark to produce 50% of keys access locality, meaning the probability that a key is stored by the node where the transaction is executing (local node), and 50% of uniform access. Increasing local accesses has a direct impact on the application contention level. In fact, since each key is replicated on two nodes, remote communication is still needed by update transactions, while the number of objects accessible by a client reduces when the number of nodes increases (e.g., with 20 nodes and 5k keys, a client on a node can select its accessed keys among 250 keys rather than 5k). Read-only transactions are the ones that benefit the most from local accesses because they can leverage the local copy of each accessed key.

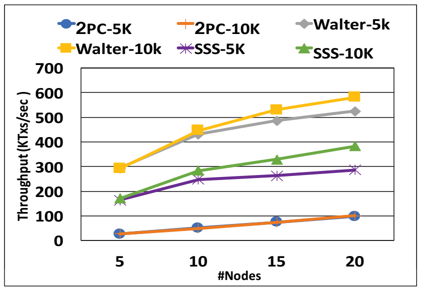


Fig. 7. Throughput of all competitors with 80% read-only transactions and 50% locality.

We report the results (in Figure 7) using the same configuration in Figure 3(c) because that is the most relevant to SSS and Walter. Results confirm similar trend. SSS is more than 3.5× faster than 2PC-baseline but, as opposed to the non-local case, here it cannot close the gap with Walter due to the high contention around snapshot-queues.

In Figure 8 we show the impact of increasing the number of read operations inside read-only transactions from 2 to 16. For this experiment we used 15 nodes and 80% of read-only

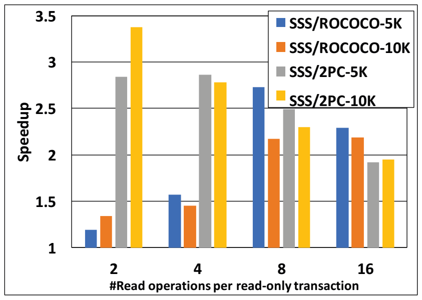


Fig. 8. Speedup of SSS over ROCOCO and 2PC-baseline increasing the size of read-only transactions.

workload. Results report the ratio between the throughput of SSS and both ROCOCO and 2PC-baseline. When compared to ROCOCO, SSS shows a growing speedup, moving from 1.2× with 2 read operations to 2.2× with 16 read operations. This is because, as stated previously, ROCOCO encounters a growing number of aborts for read-only transactions while increasing accessed objects. 2PC-baseline degrades less than ROCOCO when operations increases because it needs less network communications for read-only transactions.

VI. Related Work

Many distributed transactional repositories have been proposed in literature, examples include [3], [5], [10], [11], [13]–[15], [18], [22]. Among them, Spanner [13], Scatter [20], and ROCOCO [26] guarantee the same level of consistency as SSS.

Google Spanner [13] is a high performance solution that leverages a global source of synchronization to timestamp transactions so that a total order among them can always be determined, including when nodes are in different geographic locations. This form of synchronization is materialized by the *TrueTime* API. This API uses a combination of a very fast dedicated network, GPS, and atomic clocks to provide accuracy of the assigned timestamps. Although outstanding, Spanner's architecture needs special-purpose hardware and therefore it cannot be easily adopted and extended.

Scatter provides external consistency on top of a Paxos-replicated log. The major difference with SSS is that Scatter only supports single key transactions while SSS provides a more general semantics. ROCOCO uses a two-round protocol to establish an external schedule in the system, but it does not support abort-free read-only transactions.

Replicated Commit [25] provides serializability by replicating the commit operation using 2PC in every data center and Paxos to establish consensus among data centers. As opposed to SSS, in Replicated Commit read operations require contacting all data centers and collect replies from a majority of them in order to proceed. SSS's read operations are handled by the fastest replying server.

Granola [14] ensures serializability using a timestamp-based approach with a loosely synchronized clock per node. Granola provides its best performance when transactions can be defined as independent, meaning they can entirely execute on a single server. SSS has no restriction on transaction accesses.

CockroachDB [10] uses a serializable optimistic concurrency control, which processes transactions by relying on multi-versioning and timestamp-ordering. The main difference with SSS is the way consistent reads are implemented. CockroachDB relies on consensus while SSS needs only to contact the fastest replica of an object.

Calvin [33] uses a deterministic locking protocol supported by a sequencer layer that orders transactions. In order to do that, Calvin requires a priori knowledge on accessed read and written objects. Although the sequencer can potentially be able to assign transaction timestamp to meet external consistency requirements, SSS does that without assuming knowledge of read-set and write-set prior transaction execution and without the need of such a global source of synchronization.

SCORe [28], guarantees similar properties as SSS, but it fails to ensure external consistency since it relies on a single non-synchronized scalar timestamp per node to order transactions, and therefore its abort-free read-only transactions might be forced to read old version of shared objects.

Other protocols, such as GMU [29], Walter [32], Clock-SI [17] and Dynamo [15], provide scalability by supporting weaker levels of consistency. GMU [29] provides transactions with the possibility to read the latest version of an object by using vector clocks; however it cannot guarantee serializable transactions. Walter use a non-monotonic version of Snapshot Isolation (SI) that allows long state fork. Clock-SI provides SI using a loosely synchronized clock scheme.

Acknowledgment

Authors would like to thank anonymous reviewers for the insightful comments. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367 and by the National Science Foundation under Grant No. CNS-1814974.

VII. Conclusions

In this paper we presented SSS, a transactional repository that implements a novel distributed concurrency control providing external consistency without a global synchronization service and abort-free read-only transactions. The combination of snapshot-queuing and vector clock is the key technique that makes SSS possible. Results confirmed significant speedup over state-of-the-art competitors in read-dominated workloads.

Since the definition of external consistency does not enforce an order among concurrent transactions, it is enough for SSS's read operations to return the value written by the latest write operation. Snapshot-queuing has the potential to trace dependency between read-only and externally committed concurrent update transactions. With that, the external serialization order will leave the data store in the same state as the transactions were executed in a sequential order (one after the other), and that order does not contradict the order in which these transactions return to their clients. We leave such an extension as future work.

References

[1] CloudLab Clemson, 2017. http://docs.cloudlab.us/hardware.html.

[2] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999. AAI0800775.

[3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 405–414. IEEE, 2016.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In P. G. Harrison, M. F. Arlitt, and G. Casale, editors, *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64. ACM, 2012.

[5] P. A. Bernstein and S. Das. Rethinking eventual consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 923–928, New York, NY, USA, 2013. ACM.

[6] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[7] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[9] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, (3):203–216, 1979.

[10] Cockroach Labs. CockroachDB , 2017. https://github.com/cockroachdb/cockroach.

[11] 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. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, Aug. 2013.

[14] J. A. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference*, volume 12, 2012.

[15] 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 *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[16] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[17] 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.

[18] 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.

[19] D. K. Gifford. *Information storage in a decentralized computer system*. PhD thesis, Stanford University, 1981.

[20] 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.

[21] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[22] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[23] M. J. Kishi, S. Peluso, H. Korth, and R. Palmieri. SSS: Scalable Key-Value Store with External Consistent and Abort-free Read-only Transactions. Technical report, Lehigh University, 2019. URL: http://sss.cse.lehigh.edu/files/pubs/TR-sss.pdf.

[24] T. Landes. Dynamic vector clocks for consistent ordering of events in dynamic distributed applications. In *PDPTA*, pages 31–37, 2006.

[25] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, 6(9):661–672, 2013.

[26] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, volume 14, pages 479–494, 2014.

[27] S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In C. Georgiou and P. G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 217–226. ACM, 2015.

[28] S. Peluso, P. Romano, and F. Quaglia. SCORe: A scalable one-copy serializable partial replication protocol. In *Middleware 2012*, pages 456–475, 2012.

[29] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. Gmu: Genuine multiversion update-serializable partial data replication. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2911–2925, 2016.

[30] Red Hat. Infinispan Data Grid. http://infinispan.org/docs/stable/glossary/glossary.html#consistent_hash.

[31] R. Ricci, E. Eide, and C. Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[32] 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.

[33] 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 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[34] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 18–32. ACM, 2013.

[35] X. Wang, J. Mayo, W. Gao, and J. Slusser. An efficient implementation of vector clocks in dynamic systems. In *PDPTA*, pages 593–599, 2006.

[36] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[37] X. Yan, L. Yang, H. Zhang, X. C. Lin, B. Wong, K. Salem, and T. Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*, pages 231–243. ACM, 2018.

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