Be General and Don't Give Up Consistency in Geo-Replicated Transactional Systems

Alexandru Turcu, Sebastiano Peluso, Roberto Palmieri and Binoy Ravindran

Virginia Tech
{talex;peluso;robertop;binoy}@vt.edu

Abstract. We present Alvin, a system for managing concurrent transactions running on a set of geographically distributed sites. Alvin supports general-purpose transactions, and guarantees strong consistency criteria. Through a novel partial order broadcast protocol, Alvin maximizes the parallelism of ordering and local transaction processing. Alvin processes read-only transactions either locally or globally, according to the selected consistency criterion, and orders only conflicting transactions across all sites. We built Alvin in the Go language and conducted an evaluation study relying on the Amazon EC2 infrastructure and Paxos- and EPaxos-based state machine replication protocols as competitors. Our experimental results reveal that Alvin provides significant speed up for read-dominated TPC-C workloads and on 7 datacenters by as much as 4.8x when compared to EPaxos, and up to 26% in write-intensive workloads.

Keywords: Geo-Replication, Transaction, Distributed System

1 Introduction

In the recent years, transaction processing on geographically distributed computer systems (or "GDS") received significant research interest [22, 12, 23, 5, 17]. Geo-replicated concurrency control protocols can be classified in two approaches. The first approach ensures high consistency, but restricts the type of transactions that are allowed [23, 17]. This enables exploiting specific protocol optimizations to achieve high performance. The second approach allows general-purpose transactions, but weakens the consistency criterion for better performance [2, 22]. This has the negative effect of reduced programmability, as programmers must cope with potential inconsistent states in application behaviors.

Motivated by this gap between strong consistency/poor performance and weak consistency/good performance, we propose a geo-replicated transactional system called Alvin, which finds an effective tradeoff between performance and strong consistency. At the core of Alvin is a novel Partial Order Broadcast protocol (*POB*) that globally orders only conflicting transactions and minimizes the number of communication steps for non-conflicting transactions. While the idea of defining the agreement of consensus on the basis of message semantics is not new and has been previously introduced in Generalized Consensus [13] or

Generic Broadcast [19], POB encompasses a novel approach for ordering transactions' commits that overcomes the limitations of existing single leader-based solutions (i.e., Generalized Paxos [13]) when deployed in GDS. POB does not rely on a designated leader to either order transactions or support conflict resolution in case of conflicting concurrent transactions.

POB has been designed to inherit the benefits of state-of-the-art, multi-leader, state machine replication protocols specifically proposed for GDS such as *Mencius* [16] and *EPaxos* [17], and, at the same time, to overcome their drawbacks. In particular, POB, like Mencius [16], has the advantages of defining the final order of messages on the sender nodes. Typically, this technique avoids expensive distributed decisions by determining an a priori assignment of delivered positions to messages. This approach suffers from potentially expensive waiting conditions that are needed to ensure that the delivery of a message in position p does not precede the delivery of a message in position $p' < p$. However, POB, unlike Mencius, relies on a quorum of replies, instead of waiting for the information about delivered positions from all nodes. This makes POB's performance robust even in scenarios where nodes are far apart (as is often the case in GDS), or when the message sending rate is unbalanced among nodes.

On the other hand, POB, like EPaxos [17], may adjust the order of a message that has been already proposed, according to its dependencies, to reduce communication steps in scenarios of no conflicting proposals of dependent messages. However, unlike EPaxos, POB does not need to build a dependency graph of received messages and avoids the execution of complex tasks on that graph. Such housekeeping operations can be significantly expensive in transaction processing: the number of dependencies in the dependency graph can rapidly grow when a transaction's size and data contention increases.

Roughly, in POB, each node is the leader of transactions originating on it and is responsible for assigning a final position to those transactions. A node has a predefined and exclusive subset of positions that can be used for the assignment. As in Mencius, transactions can be delivered in the order defined by their position numbers. However, unlike Mencius, the delivery of a transaction at a certain position does not need to wait for the notification of all previous positions. This is because, besides a position, a transaction T is associated with a set of dependencies, namely, the set of transactions conflicting with T that must precede T in the order defined by POB. T's leader computes the position and the dependencies of T on the basis of a partial view of the system built by means of quorums. POB ensures that for any pair of transactions T_1 and T_2, if T_1 is in T_2's dependencies, then the position of T_1 is less than the position of T_2. Therefore, a transaction T is delivered on a node after all transactions in T's dependencies have been delivered on that node.

POB's advantages are fully exploited by P-CC, a local parallel concurrency control layer that we propose. P-CC commits non-conflicting transactions in parallel with conflicting transactions, thereby increasing the parallelism.

Alvin's processing model allows clients to execute transactions locally on the spawning site, whose execution is globally certified against concurrent trans-

actions at other sites. To this goal, POB disseminates transactions and P-CC locally validates and commits them according to the delivery order provided by POB using a timestamp-based multi-versioning scheme. This combination allows all transactions, including those aborted, to always observe a consistent state. This property is mandatory for in-memory deployment in order to avoid unexpected failures due to inconsistent memory accesses [8].

In addition to these features, ALVIN exports design choices to programmers to customize the POB and P-CC according to the needs of the application and system at hand. As an example, ALVIN offers two strong consistency criteria that programmers can select, namely, Serializability (SR) [3] and Extended Update Serializability (EUS) [1, 20] (i.e., PL-3U [1]). With the former, transactions that never write (i.e., read-only) must be broadcast through POB. In contrast, with the latter, such transactions execute locally at the cost of generating some non-serializable schedules, which, however, are usually silent to the application. Another example is the potential for computing a fast decision on the transaction delivery order, at the cost of quorum bigger than that for a classic decision.

We built ALVIN in the *Go* programming language and evaluated on the Amazon EC2 infrastructure using up to 7 sites, and benchmarks including Bank [11] and TPC-C [6]. As competitors, we implemented two certification-based transactional systems [18] that rely on MultiPaxos [14] and EPaxos [17] for their ordering layer. Our experiments reveal that ALVIN provides significant speed up for TPC-C workloads and 7 datacenters by as much as 4.8× when compared to EPaxos and configured for exploiting EUS. This significant gain is due to a more efficient execution of read-only workload, which is enabled by EUS's semantics. Rather, if ALVIN runs under SR, it gains up to 26% over EPaxos because it does not pay the cost of graph analysis needed by EPaxos for delivering transactions. On Bank, due to its small transactions and trivial dependency graphs, that cost is not significant, thus EPaxos behaves similarly to ALVIN. MultiPaxos highlights the drawbacks of having a single leader in GDS, thus its performance is lower than other (multi-leader) competitors.

The paper makes the following contributions: *(1)* ALVIN, the first geo-replicated transactional system that guarantees a strong consistency level and supports the execution of general-purpose transactions in classic asynchronous environments; *(2)* a novel multi-leader protocol for partially ordering transactions, enabling high scalability in geo-replicated environments. In addition, the protocol does not need complex local processing for determining the final delivery order, yielding reduced client-perceived latency; *(3)* a publicly available prototype[1], which can be customized for coping with different execution environments.

2 Related Work

Many modern transactional systems employ geo-replication as a means to reduce data access latency and to provide fault-tolerance and disaster recovery.

[1] http://www.hyflow.org/software.html

Spanner [5] is Google's globally-replicated database. It provides externally-consistent transactions, but its architecture is complex: it relies on the TrueTime API, which exposes the absolute time and the uncertainty of the time measurement. Alvin's architecture is more general and suited for easier deployment.

Walter [22] and MDCC [12] are two solutions designed for geo-replicated transactional systems. Walter ensures Parallel Snapshot Isolation, which allows non-conflicting write transactions that span multiple sites to commit even if they observed incompatible histories. Alvin ensures that all update transactions are serializable. On the other hand, MDCC commits transactions by using one instance of Multi-Paxos [14] (or Generalized Paxos [13] to exploit commutative operations) per replication group containing the accessed data items and, if a transaction touches multiple replication groups, an additional phase is required to reach a consensus among the leaders of the various groups.

Lynx [23] is a geo-distributed transactional storage that works by chopping transactions into sequences of pieces. Each piece executes at a different datacenter, and the system usually replies to clients after the first hop. Lynx's drawback is that it does not tolerate aborts after a chain's first segment.

Finally, we consider EPaxos [17] and Mencius [16] as the closest approaches to Alvin. EPaxos [17] proposes a partial order protocol for ordering conflicting commands and it uses a per-command leader to avoid the designated leader of (Generalized) Paxos. It considers two types of quorums for executing a command: one is used for implementing a fast-path of one round-trip of communication in case the command does not conflict with other concurrent commands; the other is used in case two phases of communication are required to agree on the order.

EPaxos yields high performance but it has several drawbacks when plugged in transactional processing or in the presence of read operations. In fact, after having agreed on the dependency set for a command, each node adds that command to a dependency graph and its execution is in accordance with an order computed over the strongly connected components of that graph. In case a command represents a transaction or even a read operation, the client has to wait until the command's outcome is available, thus putting the graph analysis into the execution's critical path. Alvin is not based on graph analysis because dependencies are already available when the transaction attempts to commit, thus resulting in better performance.

At the core of Mencius' [16] ordering protocol there is the fixed assignment of sending slots to nodes. A sender can decide the order of a message only after hearing from all nodes. This approach results in poor performance in case there is a slow or faraway node, as in geo-replication.

3 Assumptions and System Model

We assume a set of geographically distributed sites $\Pi = \{P_1, P_2, \ldots, P_N\}$ that cooperate to synchronize their activities on common shared data. They rely on a wide area network as the communication infrastructure, therefore we assume an asynchronous distributed system. We do not assume any specific distribution

of network delays and we do not upper-bound them either. Every message may experience an arbitrarily large, although finite, delay.

Each site (or node) can be seen as a logical representation of a datacenter. Managing the synchronization within each datacenter is an orthogonal problem which we scope out in this paper. Each site is equipped with the entire shared data set, thus transactions running on that site can access data locally.

We assume that the total number of sites is equal to N, where at most $f < \lceil \frac{N}{2} \rceil$ of them can be faulty at any time, thus at least a majority of nodes is always correct. In this paper we assume sites fail according to the crash-stop failure model [3] and we scope out any malicious behavior. In any ordering communication step, a node contacts all the sites and waits for a *quorum* Q of replies. We define two types of quorum size: a *classic quorum* (CQ) size and a *fast quorum* (FQ) size. We assume that both CQ and FQ are at least equal to $\lfloor \frac{N}{2} \rfloor + 1$. This way any two quorums always intersect, thus ensuring that, even though f failures happen, there is always at least one site with the last updated information that we can use for recovering the system. The values assumed by CQ and FQ are configuration-dependent, and they will be specified throughout the presentation of the communication layer.

In order to eventually reach an agreement on the order of transactions when sites are faulty (e.g., a datacenter is unreachable), we assume that the system can be enhanced with the weakest type of unreliable failure detector [10] that is necessary to implement a leader election service [9].

4 Alvin: Geo-Replicated Transactional System

We propose simple object-oriented interfaces, where all accesses (Read, Write) to shared objects are enclosed between Begin and Commit operations.

Alvin bases its benefits on the exploitation of a partial order of transactions rather than a total order. In fact, ordering all the transactions' commits on all nodes is sufficient to guarantee that all nodes execute the same state transitions, but it is too strong as a condition, especially in GDS, because it enforces that the finalization of a transaction is delayed by the completion of even non-conflicting transactions, thus hampering the system's scalability. On the contrary, enforcing that only conflicting transactions are ordered on all nodes (as in Alvin) has a twofold benefit: it still guarantees that all nodes eventually converge on a common state, and it allows a degree of parallelism needed for scaling in low inter-datacenter conflict scenarios (which are the expected workloads in GDS).

The software architecture of Alvin includes two fundamental layers: the Partial Order Broadcast layer (POB) and the Parallel Concurrency Control layer (P-CC). POB is in charge of broadcasting transactions to certify and commit them according to the certification-based approach [18] and in a way such that conflicting transactions are always delivered in the same order on all nodes. P-CC is responsible for optimistically executing transactions by always providing a consistent view of the transactional state, and applying the updates of write transactions that can commit. This makes Alvin a geo-replication solution also

suitable for in-memory transactional systems, which require that all transactions (even those aborted) do not observe incorrect states. This requirement has been defined to be desirable for non-sandboxed environments [8] because reading from an inconsistent snapshot could generate an application's unrecoverable failure.

The transactional application executing on top of the platform is composed of multiple threads balanced on all nodes. According to the certification-based replication scheme [18], each thread activates and executes a transaction T at the same node where it is running, recording objects read from and written to in private spaces called the read-set ($T.RS$) and the write-set ($T.WS$) respectively.

T is optimistically executed under the control of P-CC and, when it reaches the stage where all of its operations have been executed, the executing thread broadcasts T via the POB layer and waits until T is globally validated and either aborted or committed. In the former case the application thread has to re-issue T from its very beginning; in the latter case T's updates are applied to the transactional shared state after the commit of any other transaction preceding T in the order defined by POB. During the optimistic execution of a transaction, in fact, the updates of write operations are only buffered in the transaction's write-set and they cannot be directly applied to the shared state because the transaction could abort later on.

The POB layer provides two interfaces to send and receive a transaction T: POBroadcast(T), used for broadcasting a transaction T along with its read-set and write-set; PODeliver($T, \{T_1, \cdots, T_m\}$), used for delivering a transaction T to nodes, along with the set of transactions $\{T_1, \cdots, T_m\}$, defined as $deps_T$, which conflict with T and must be processed (i.e., certified and possibly committed) before T. Formally, two transactions T and T' are conflicting if at least one of the following three conditions are verified: (i) $T.WS \cap T'.WS \neq \emptyset$, (ii) $T.WS \cap T'.RS \neq \emptyset$, (iii) $T.RS \cap T'.WS \neq \emptyset$.

4.1 Partial Order Broadcast Layer

The core idea behind the design of POB is guaranteeing that all nodes agree on the same delivery order for conflicting transactions. This is because, if two transactions do not conflict, then they can be validated and committed (or aborted) in any order (i.e., all the orders are equivalent due to the absence of conflicts). Formally, POB guarantees that any pair of conflicting transactions – i.e., two transactions that access at least one common object, where at least one of the accesses is a write operation – are not delivered in different orders on two nodes.

Therefore POB guarantees the following properties:

- *P1: Strong Uniform Conflicting Order.* If some node delivers message $m = [T, deps_T]$ before message $m' = [T', deps_{T'}]$ and transactions T and T' conflict, then every node delivers m' only after m.
- *P2: Local Dependency.* For any node that delivers message $m = [T, deps_T]$ before message $m' = [T', deps_{T'}]$ and T and T' conflict, then $T \in deps_{T'}$ and $T' \notin deps_T$ (i.e., no circular dependency between conflicting transactions).

Property *P1* is defined as strong because it does not allow omission of messages. It is in contrast with the weak order property that, instead, allows the

omission of messages despite the fact that the order of delivery on all nodes is still preserved. In particular, POB does not allow a scenario in which a node P_i delivers m before m' while a node P_j delivers m' without delivering m, where m and m' contain two conflicting transactions. We need the strong version of this property because in transaction processing, even if the partial order is not violated, the aforementioned scenario can generate two different outcomes for the same transaction T', enclosed in m', on the nodes P_i and P_j. As an example, the P-CC on P_i could abort T' because its execution has been invalidated by transaction T contained in m, while P_j commits T'.

The property *P2* regards the semantics of the interfaces exposed to P-CC. In particular, when POB delivers a message $m' = [T', deps_{T'}]$ to P-CC, transaction T' has to wait for the completion of all the transactions in $deps_{T'}$ before determining its outcome. This condition is sufficient for ensuring that all transactions are processed in accordance with the partial order defined by POB. In addition, POB also guarantees the typical properties of a reliable broadcast service (Validity, Integrity, Uniform Agreement) [7].

Due to space constraints we report the detailed correctness proofs of POB in the technical report.

Overview. The idea of enforcing an order only among conflicting commands has already been specified by the Generalized Consensus [13] and Generic Broadcast [19] problems and followed by a set of implementations, e.g., Generalized Paxos [13], EPaxos [17]. POB improves the above proposals by relying on a fully decentralized design without leveraging on a stable leader to establish the order of transactions and without expensive housekeeping computations before issuing the delivery of a transaction.

The main idea behind POB is to define a deterministic scheme for the assignment of *delivery slots* (i.e., positions in the final order that are associated with positive integers) to submitted transactions, by following the general design of *communication history*-based total order broadcast protocols [7, 16] in which the delivery order of messages is determined by the senders. In POB, for each transaction T we define a unique transaction leader tl_T that establishes the final delivery position of T by applying the following rules:

- *Rule 1.* If a node P_i is T's leader (i.e., tl_T), then T can only be delivered in unused positions numbered with pos_T, such that $pos_T \bmod N = i$.
- *Rule 2.* Transaction T' is delivered in position $pos_{T'}$ if and only if, for each conflicting transaction T delivered in position $pos_T > pos_{T'}$, $T' \in deps_T$ and $T \notin deps_{T'}$, where $deps_T$ (respectively $deps_{T'}$) is the set of transactions which T (respectively T') depends on.

Rule 1 guarantees that two transactions from different leaders cannot occupy the same position. However, Alvin is also able to concurrently broadcast multiple requests from the same node and, as it will be clear later, this could cause two transactions from the same leader to be assigned the same position number. Such transactions are deterministically ordered using the transaction identifier. On the other hand, *Rule 2* is specifically defined for satisfying property *P2*.

The transaction leader tl_T for a transaction T is either the sender of T, or any other elected node if T's sender is suspected as crashed by the failure detector.

Protocol. A transaction T, that is submitted to the POB service via the POBroadcast(T) interface, goes through four phases: *Proposal phase*, *Decision phase*, *Accept phase* and *Delivery phase*.

Proposal phase. The node P_i, acting as the leader of T (i.e., tl_T), selects the next available position number for T to be proposed to all the other nodes. This position, named pos_T, is the smallest number among the ones allowed by *Rule 1* and greater than any other position that P_i has observed as already used. P_i also selects the set $deps_T$ of dependencies, namely all transactions T' conflicting with T and having a (even temporary) position less than pos_T.

Subsequently, P_i broadcasts a Propose message with the tuple $\langle T, pos_T, deps_T, e\rangle$ to all nodes. By broadcasting a transaction T we mean broadcasting T's identifier ($T.tid$), read-set ($T.RS$) and write-set ($T.WS$).

The e value is an epoch number associated with transaction T and the messages containing T. It identifies the epoch in which messages for T can be exchanged. A transition to a new epoch is forced by T's new elected leader when T's old leader is suspected as crashed. Messages associated with an epoch e_1 cannot be processed by nodes that have already executed a transition to an epoch e_2, with $e_2 > e_1$. In the Propose message, the epoch number is 0 since it identifies the initial epoch of T in which T's sender is recognized by default as the initial leader tl_T of T.

A node P_j receiving a Propose message for T, replies with an AckPropose message in order to update P_i with the set of transactions conflicting with T and observed by P_j so far, i.e., $newDeps_T^j$, and a possibly new position to be chosen for T, i.e., $newPos_T^j$. In particular, let us define $temp_T^j$ as the smallest number among the ones allowed by *Rule 1* for P_i and greater than any other position used by transactions conflicting with T and already received by P_j. Then $newPos_T^j$ is equal to $temp_T^j$ in case $temp_T^j$ is greater than pos_T (the position proposed by P_i); otherwise it is equal to pos_T. On the other hand, $newDeps_T^j$ is the set of all transactions T' conflicting with T and having a (even temporary) position less than $newPos_T^j$.

A transaction T received during this phase is marked as Pending and it is inserted in a data structure named *delivery queue* (DQueue). On each node P_j, DQueue is a queue storing the transactions received by P_j as tuples $\langle T, pos_T, deps_T, status\rangle$, where *status* has values in {Pending, Accepted, Stable}. The tuples in the DQueue are totally ordered according to their pos_T's values.

Decision phase. Transaction T's leader P_i waits for a quorum of FQ replies from the previous phase. It then computes the final position pos_T and final dependencies $deps_T$ that are used for the delivery of T in the next phases as follows: pos_T is the maximum position among the proposals ($newPos_T^j$) in the quorum, while $deps_T$ is the union among the dependency sets ($newDeps_T^j$) proposed in the quorum. Afterwards, P_i broadcasts an Accept message for T with the final position and dependencies in order to request to other nodes to accept the delivery of T. The value of FQ in the base configuration of POB is equal to

$f + 1$. Section 4.1 shows how to enable a so called *fast transaction decision* by changing the value of FQ.

Accept phase. A node P_j receiving T updates its DQueue accordingly. This means changing the status of T to ACCEPTED and replacing the old values of pos_T and $deps_T$ with the ones received in this phase. Then P_j replies with an ACKACCEPT message by including pos_T and a possibly new set of dependencies $newDeps_T^j$. In fact, in this phase P_j can also attach an additional set $deltadeps_T$ to $deps_T$, if it detects that it received transactions T^δ conflicting with T and having a position in between the old and the new values of pos_T in DQueue. This is because, P_j could have been received T^δ after that $deps_T$ was computed in the *Proposal phase*. More formally, $newDeps_T^j$ is equal to $deps_T \cup deltadeps_T$, where $deltadeps_T$ is the set of all transactions $T' \notin deps_T$ conflicting with T and having a (even temporary) position less than pos_T.

Delivery phase. T's leader P_i waits for a quorum of CQ replies from the previous phase, where CQ is equal to $f + 1$, to be sure that its decision will be stable even if f failures (including itself) occur. After that, it broadcasts its decision via a STABLE message including pos_T, which was already decided in the *Decision phase*, and $deps_T$, which is computed as the union of the $newDeps_T^j$ collected during the previous phase.

A node receiving the STABLE message for T marks T as STABLE in its DQueue by also replacing the old values of pos_T and $deps_T$ with the ones received in this phase. Then, the node can deliver the message $[T, deps_T]$ to the concurrency control when all transactions in $deps_T$ have been already delivered by triggering PODELIVER$(T, deps_T)$.

Since the position of a transaction T' can change throughout the execution of the POB protocol, there might be scenarios in which a transaction $T'' \in deps_{T'}$ becomes STABLE with a position $pos_{T''}$ greater than the final position of T', which would lead T' to wait infinitely for a conflicting transaction that is actually ordered after it. To address this problem, in such a case T'' is removed from the $deps_{T'}$ set. Note that, when this condition is true, T' is guaranteed to be already present in $deps_{T''}$.

Failure Recovery. When a node P_k detects that T's current leader P_i crashed, and P_k has not yet marked T as STABLE, it attempts to become T's new leader by executing a classic Paxos *Prepare phase* [14]. Therefore, P_k broadcasts an epoch number e for T greater than the last one observed for T. Then it waits for a PROMISE from a quorum Q of $f+1$ nodes, meaning that they will not participate in any new *Prepare phase* or *Proposal/Accept phases* for T associated with an epoch number less than e. The nodes in Q also send back the latest status known for T and identified by the most recent tuple $\langle T, pos_T, deps_T, status \rangle$ they have in their DQueue. This allows P_k to take a final decision that cannot differ from the one P_i took (if any).

Therefore, we distinguish three cases depending on the value of *status*:

- At least one $\langle T, pos_T, deps_T, \text{STABLE} \rangle$ is received from Q. In this case, P_k starts a *Delivery phase* by broadcasting a STABLE message for T with pos_T and $deps_T$.

- At least one $\langle T, pos_T, deps_T, \text{ACCEPTED}\rangle$ is received from Q and no STABLE status is present. In this case, P_k starts an *Accept phase* by broadcasting an ACCEPT message for T with pos_T and $deps_T$.
- Neither ACCEPTED nor STABLE value is received from Q. In this case, P_k selects a new position available for T by restarting a new instance of the protocol starting from the *Proposal phase* for T.

Fast Transaction Decision. POB can be configured to allow a so called *fast transaction decision* about the order of a transaction if there are no concurrent conflicting transactions. The idea is the same as adopted in [15, 17] and entails that a transaction leader can determine the final position of a transaction early, i.e., after only two communication steps, because it has received all equal ACK-PROPOSE messages from a quorum of nodes in the *Decision phase*. Enabling the fast decision introduces a trade-off. On the one hand, the leader can define the order of a transaction in fewer communication delays, but on the other hand, quorum sizes become bigger and the recovery phase more complex.

When fast decisions are enabled, POB must use a size FQ greater than CQ, i.e., fast quorums bigger than classic quorums, otherwise a fast ordering decision by a transaction leader P_i could be irrecoverable after the fault of P_i. Specifically, the new leader of a transaction T, e.g., P_k, has to decide in the same way the old leader of T, e.g., P_i, decided.

First of all, we have to notice that in case P_i had a fast decision for T by including (respectively not including) a concurrent and conflicting transaction T' in its dependencies, it would be impossible that the leader of T' also had a fast decision by including (respectively not including) T in its dependencies, due to the definition of quorums. Therefore, a trivial recovery of P_k would be contacting the leader of T' to know the final decision for T' with respect to T. On the contrary, in case the new leader of T, i.e., P_k, is not able to contact the current leaders of the transactions conflicting with and concurrent to T, it must take a decision by analyzing collected replies.

If P_i had a fast decision for T, i.e., it collected all equal proposals for T hence it decided in two communication steps, we have to enforce a deterministic behavior on the quorum of replies collected by P_k during recovery. Specifically, in that case, we want P_k to have a majority (i.e., $\left\lfloor \frac{CQ}{2} \right\rfloor + 1$) of values equal to the fast decision in the quorum of replies collected during recovery. In other words, when the new leader of T collects a classic quorum in the recovery phase, then the number of replies different from a possible fast decision of the old leader (and that do not include the reply from the leader of a generic conflicting transaction), i.e., $N - FQ - 1$, has to be less than the majority in the quorum, i.e., $\left\lfloor \frac{CQ}{2} \right\rfloor + 1$. Equation 1 follows.

In addition to the above, another constraint is needed to avoid two new leaders of two conflicting and concurrent transactions T and T', here called *opponents*, both believing that the associated old leaders of T and T' respectively had fast decisions. So after f failures and ignoring the reply from the other

opponent, i.e., -1, two opponents cannot both collect a sufficient number of replies, i.e., $\frac{N-f}{2}$, that summed up f form a fast quorum. Equation 2 follows.

$$N - FQ - 1 < \left\lfloor \frac{CQ}{2} \right\rfloor + 1 \quad (1) \qquad \frac{N-f}{2} + f - 1 < FQ \quad (2)$$

If we minimize the ratio $\frac{N}{f}$ by still considering $f < \lceil \frac{N}{2} \rceil$, e.g., $N = 2f + 1$, we obtain the following sizes for the classic and fast quorums, respectively:

$$CQ = f + 1 \quad (3) \qquad FQ = f + \left\lfloor \frac{f+1}{2} \right\rfloor \quad (4)$$

Note that CQ and FQ in Equations 3 and 4 have the same values adopted by EPaxos. The new recovery phase that applies under this optimization is a trivial extension of the recovery procedure as presented in EPaxos.

By using these new values for CQ and FQ, the *fast transaction decision* works as follows. After having collected FQ AckPropose in the *Proposal phase* for transaction T, T's leader can directly send the final decision via the Stable message if all the collected proposals are the same. Otherwise it proceeds in the classic way by entering the *Accept phase*.

4.2 Parallel Concurrency Control Layer

Each node in the system is equipped with a *Parallel Concurrency Control* layer (P-CC) that is responsible for executing transactions submitted by clients as well as processing the commit of transactions delivered by POB.

We can split P-CC's operations into two parts. The first part, the *execution phase*, is responsible for executing transactions optimistically. Following the classic multi-version concurrency control scheme implemented in state-of-the-art in-memory transactional systems [4], a transaction executes its read operations on the snapshot of memory present at the time of its beginning (i.e., which includes the set of commits applied before the transaction began), while its writes are buffered and can be applied atomically on all nodes only if the transaction can commit. At this stage, the transaction's read-set and write-set are also built.

The second part, the *commit phase*, is responsible for validating and committing the optimistic execution of transactions on all nodes. This is done by sending the commit message of a transaction T with T's read-set and write-set via the POB layer, and triggering the validation of T as soon as T is delivered by POB. A sufficient condition to guarantee that T appears as executed atomically on all nodes is to validate it by checking that no value read by T has been updated in between its beginning and its finalization.

The P-CC layer guarantees that *i)* every transaction, including aborted ones, observes a consistent state, and *ii)* the set of committed transactions satisfies Serializability (SR). Even if SR is one of the reference consistency criteria for transactional systems, it might be considered not necessary for several types of applications [20]. Such applications stand to benefit from requiring: *i)* that the

transactional state never performs a transition to an incorrect state, and *ii)* that all operations always observe consistent states. While the former requires that only update transactions appear as executed sequentially (as demanded by SR), the latter allows to implement read-only transactions with lower guarantees.

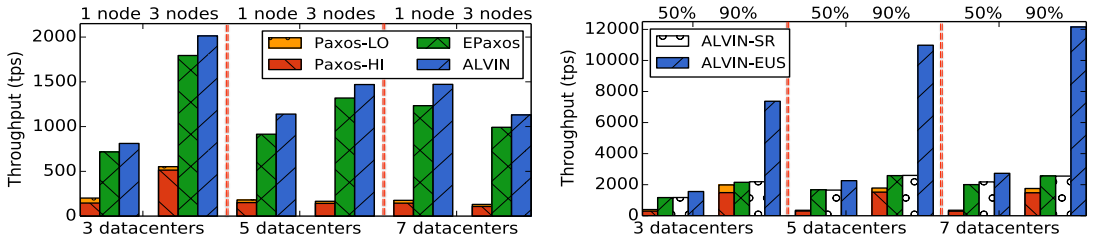
In order to take advantage of the above considerations, Alvin supports another strongly consistent criterion, besides SR, named Extended Update Serializability (EUS) [1], which can be considered as strong as Serializability for many common workloads [20, 6]. Roughly speaking, EUS preserves Serializability of committed update transactions and disallows any transaction to observe incorrect states. However, with EUS, two read-only transactions might observe two different non-compatible histories of commits, caused by a different perceived commit order of non-conflicting update transactions. EUS gives the necessary flexibility to Alvin for committing two update non-conflicting transactions T_h and T_k in an arbitrary order thanks to the POB layer, such that T_h completes before T_k on a node P_i, and vice-versa on another node P_j. At that point, transactions T_q and T_w that are executing on nodes P_i and P_j respectively, are allowed to observe two different serializations of T_h and T_k without providing any inconsistent view to the application. Then, in case T_q and T_w are read-only, they are also allowed to commit under EUS.

Therefore, in order to allow behaviors like the one described above, and to support EUS, P-CC can be configured to avoid the global certification of read-only transactions through POB at commit time, so that a read-only transaction can safely commit as soon as it has been processed locally. In fact, as described in [21], certification-less read-only transactions disallow Serializabiliy in case a total order on the commit of update transactions is not enforced.

Summarizing, since POB ensures a total order among commits of conflicting (both read-only and update) transactions and P-CC ensures that the read-set of committed transactions is not invalidated by concurrent transactions, then Alvin enforces SR [21]. Moreover, P-CC guarantees that all read operations return the last value committed before the beginning of the transaction execution. This way, any transaction can never observe inconsistent states. Therefore, if read-only transactions are only processed locally without being submitted for a global certification, Alvin guarantees SR restricted to committed update transactions, as demanded by EUS.

5 Evaluation

We evaluate Alvin by comparing it against two certification-based transaction execution protocols [18] that rely on MultiPaxos [14] and EPaxos [17] for their ordering layer. MultiPaxos ensures serializability by total-ordering the commit requests for all write transactions, while serving read-only transactions locally leveraging multi-versioning. However, MultiPaxos is sequencer-based, thus the location of the node designated as the leader significantly affects its performance. In order to conduct a fair comparison, we used two versions of MultiPaxos: one with the leader located at a node with a point-to-point latency to other nodes



(a) Write-intensive workload for {3,5,7} sites and {1,3} nodes per site.

(b) 50% and 90% read-only transactions. One node per site.

Fig. 1. Throughput of TPC-C benchmark.

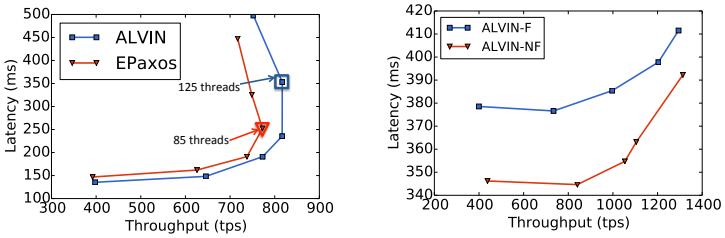
that is higher than the average (*Paxos-HI*), and another where the connection latency is lower (*Paxos-LO*). We implemented Alvin and competitors in the same transaction processing framework, using *Go* as the programming language.

We used two benchmarks in the evaluation: TPC-C [6] and Bank [11]. The former is a well known benchmark representative of on-line transaction processing workloads; the latter mimics operations of a monetary application where each transaction transfer amount of money among bank accounts. We ran our experiments on the Amazon EC2 infrastructure, using *r3.2xlarge* nodes in up to 7 geographically distributed sites (three in Asia, two in North America, one in South America and Europe). Each node has 8 CPU cores and 61GB RAM. Results are the average of 7 samples.

Figure 1 reports Alvin's throughput of TPC-C benchmarks by varying the number of geographically distributed sites {3,5,7}. In Figure 1(a) we also changed the number of nodes per site as {1,3}, using a write intensive workload (<3% read-only). Results on read-dominated workloads are showed in Figure 1(b). Here we change the percentage of read-only transactions from 50% to 90% while using one node per datacenter. In this read dominated scenario we explore both versions of Alvin, ensuring SR (Alvin-SR) and EUS (Alvin-EUS), with the purpose of assessing the effectiveness of EUS. In all depicted scenarios, we configured Alvin to run with fast decisions enabled. We batch messages for all competitors, using a window of 20 to 50 msec, according to the nodes deployed.

TPC-C's transactions access several shared objects and have a non-negligible computation. This results in long transaction execution time and a complex dependency graph to be analyzed during the processing of commit requests in EPaxos. Rather, Alvin is able to improve the parallelism thanks to the different delivery rules of POB, gaining up to 26% in throughput against EPaxos. Both EPaxos and Alvin sustain their throughput while increasing the system's load until 9 nodes (3 datacenters with 3 nodes each), then the system becomes overloaded and performance degrades due to increasing contention. MultiPaxos in both its configurations performs worse than others due to the presence of single remote leader that slows down the entire system's progress. In addition, here transactions are long thus the sequential certification limits its performance.

Figure 1(b) shows the effectiveness of exploiting EUS in read-dominated workloads by avoiding to broadcast read-only transactions via the ordering layer. Therefore Alvin-EUS provides a speed up of up to 4.8× in throughput when compared to Alvin-SR and EPaxos. It is important to notice that in these scenarios, MultiPaxos is also able to take advantage of local computation of read-only transactions. In fact, its Paxos-LO configuration performs similar to EPaxos and Alvin-SR for the case of 90% of read-only transactions and 3 datacenters. In other scenarios, Paxos-LO saturates its leader's resources, slowing down the ordering process. As before, Paxos-HI exposes poor performance due to the high communication latency with the faraway designated leader. Regarding the comparison between EPaxos and Alvin-SR, they follow about the same trend observed in Figure 1(a) because they both process read-only transactions in the same way.



(a) Alvin Vs EPaxos on 5 sites. (b) Impact of fast decision.

Fig. 2. Throughput Vs Latency using TPC-C benchmark varying application threads.

In Figure 2(a) we plot the latency increasing the system's load by adding application threads per node from 15 to 125. Here, we used 5 sites and TPC-C as the benchmark, adopting the same workload as in Figure 1(a). For increasing the readability of the plot we excluded MultiPaxos because its results were 3× slower than the other competitors. From the analysis of EPaxos's and Alvin's trends we observe that Alvin has a lower transaction latency and it sustains its throughput better than EPaxos. Specifically, with 85 threads per site EPaxos stops scaling while Alvin is still able to serve more requests. Alvin reaches its saturation point running 125 threads per site.

With the plot in Figure 2(b) we highlight the importance of configuring Alvin without the fast decision in high contention scenarios. In these situations, the probability of taking a fast decision after having collected a fast quorum of replies is low. Therefore the POB layer always pays the maximum number of communication steps to reach a decision by contacting a fast quorum of nodes in the *Proposal phase* and then falling back to the *Accept phase*. Disabling the fast decision forces the leader to always collect replies from a classic quorum. We configured TPC-C as in Figure 1(a) with 7 sites and one node each, and we increased the load as before. Alvin-NF (fast decisions disabled) improves the latency of Alvin-F (fast decisions enabled) up to 30 msec, confirming that, in some scenarios, waiting for an unlikely fast decision does not pay off.

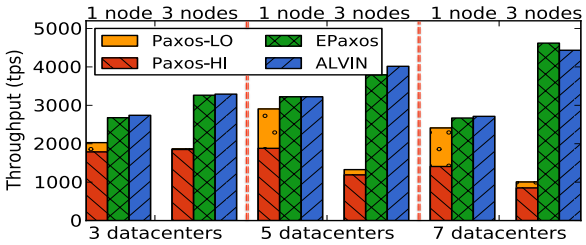


Fig. 3. Throughput under write-intensive workload for {3,5,7} sites and {1,3} nodes per site using Bank benchmark.

The Bank benchmark has very small transactions (only few operations) and the amount of transactional work can be considered as negligible when compared to the coordination steps required for establishing the agreement on the global ordering. This makes the results of both ALVIN and EPaxos comparable in almost all configurations tested as we showed in Figure 3. Bank's accesses are uniformly distributed across all objects and we managed the total number of shared objects for having an average transaction's abort rate in the range of 10-20%.

EPaxos's dependency graph analysis does not slow down the transaction's critical path significantly because the strongly connected components with more than one node are only 1.7% of all, thus the main impacting factor on the performance is the number of communication delays used for delivering transactions and, with fast decisions enabled, both ALVIN and EPaxos use the same communication delays for delivering. However, it is worth noticing that all competitors relying on partial order instead of total order sustain their throughput when we increase the number of nodes until 7 datacenters, where they start degrading. MultiPaxos in both its configurations performs worse than others due to the presence of single remote leader that slows down the entire system's progress. The exception is Paxos-LO, which is the closest to others because it benefits from having a low latency leader when site count is limited.

6 Conclusion

At its core, the design of ALVIN shows that it is possible to achieve an effective tradeoff between performance and programmability in geo-replicated environments. An important insight of our work is that partial ordering of transactions can be significantly exploited to speed up local concurrency control through parallelism and that it can be determined without a unique leader, which increases scalability in a geo-replicated setting.

Acknowledgments

This work is supported in part by US National Science Foundation under grant CNS-1217385.

References

1. Adya, A.: Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. PhD thesis AAI0800775. MIT (1999)
2. Almeida, S., Leitão, J., Rodrigues, L.: ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In: 8th ACM EuroSys, pp. 85–98. ACM (2013)
3. Bernstein, P. A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
4. Cachopo, J., Rito-Silva, A.: Versioned Boxes As the Basis for Memory Transactions. Sci. Comput. Program. 63(2), 172–185 (2006)
5. Corbett J. C. et al.: Spanner: Google's Globally Distributed Database. ACM Trans. Comput. Syst. 31(3), 8:1–8:22 (2013)
6. TPC-C Benchmark, http://www.tpc.org/tpcc/
7. Défago, X., Schiper, A., Urbán, P.: Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. ACM Comput. Surv. 36(4), 372–421 (2004)
8. Guerraoui, R., Kapalka, M.: On the Correctness of Transactional Memory. In: 13th ACM SIGPLAN PPoPP, pp. 175–184. ACM (2008)
9. Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming. Springer (2006)
10. Guerraoui, R., Schiper, A.: Genuine Atomic Multicast in Asynchronous Distributed Systems. Theor. Comput. Sci. 254, 297–316 (2001)
11. Hirve, S., Palmieri, R., Ravindran, B.: Archie: A Speculative Replicated Transactional System. In: 15th ACM/IFIP/USENIX Middleware. ACM (2014)
12. Kraska, T., Pang, G., Franklin, M. J., Madden, S., Fekete, A.: MDCC: Multi-data Center Consistency. In: 8th ACM EuroSys, pp. 113–126. ACM (2013)
13. Lamport, L.: Generalized Consensus and Paxos. Technical report MSR-TR-2005-33, Microsoft Research (2005)
14. Lamport, L.: The Part-time Parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)
15. Lamport, L.: Fast Paxos. Distributed Computing 19(2), 79–103 (2006)
16. Mao, Y., Junqueira, F. P., Marzullo, K.: Mencius: Building Efficient Replicated State Machines for WANs. In: 8th USENIX OSDI, pp. 369–384. USENIX (2008)
17. Moraru, I., Andersen, D. G., Kaminsky, M.: There is More Consensus in Egalitarian Parliaments. In: 24th ACM SOSP, pp. 358–372. ACM (2013)
18. Pedone, F., Guerraoui, R., Schiper, A.: The Database State Machine Approach. Distrib. Parallel Databases 14(1), 71–98 (2003)
19. Pedone, F., Schiper, A.: Generic Broadcast. In: 13th DISC, pp. 94–108. Springer-Verlag (1999)
20. Peluso, S., Ruivo, P., Romano, P., Quaglia, F., Rodrigues, L.: When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In: 32nd ICDCS, pp. 455–465. IEEE Computer Society (2012)
21. Schmidt, R., Pedone, F.: A Formal Analysis of the Deferred Update Technique. In: 11th OPODIS, pp. 16–30. Springer-Verlag (2007)
22. Sovran, Y., Power, R., Aguilera, M. K., Li, J.: Transactional Storage for Geo-replicated Systems. In: 23rd ACM SOSP, pp. 385–400. ACM (2011)
23. Zhang, Y., Power, R., Zhou, S., Sovran, Y., Aguilera, M. K., Li, J.: Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In: 24th ACM SOSP, pp. 276–291. ACM (2013)