Non-Interleaving Operational Semantics for Geographically Replicated Databases

Gabriel Ciobanu and Ross Horne
Romanian Academy, Institute of Computer Science, Blvd. Carol I, no. 8, 700505 Iaşi, Romania

***Abstract*—For scalable distributed database systems, weak consistency models are essential. Distributed databases, such as Google Spanner, scale to millions of nodes that replicate data across datacentres possibly located on different continents. At this scale, it is infeasible to maintain serialisability, which assumes that a global total order over committed transactions can be established. Instead, weaker consistency models, such as eventual consistency, causal consistency, sequential consistency and external consistency, are assumed. The problem is that operational models, such as labelled transition systems, tend to assume an interleaving semantics which serialises transactions. To address this limitation, we provide a novel operational model that allows a weaker notion of consistency for a geographically distributed database inspired by Spanner. We reduce the timing guarantees provided by Spanner's TrueTime protocol to causal dependencies that are specified in a formal calculus.**

I. Introduction

Geographic replication, the replication of data across data centres in geographically separated locations, is now a requirement for many Cloud providers. The primary reasons are high availability and durability (safeguarding data). In the event of some data centres becoming unavailable, due to any reason (network partition, administrative oversights, routine maintenance, or natural disaster), the services reliant on the data can continue without outage. Furthermore, in the event of catastrophic failure (e.g. fire), no data is lost. Failure to provide high availability leads to loss of customers. Failure to safeguard data leads to serious legal cases. Competitive Cloud providers are expected to manage such risks [3].

A secondary benefit of geographic replication is that the data is closer to clients. For Japanese clients, reading from a nearer data centre in Tokyo can significantly reduce the latency of read-only transactions, compared to reading from a data centre in London. However, the reduction in read-only latency comes at the cost of slower read-write transactions that guarantee reasonable consistency. Thus, systems experts behind geographically replicated databases must compromise between performance and consistency.

In this work, we analyse the consistency model devised for Google Spanner [16], which guarantees external consistency [22] while at the same time providing acceptable write performance. To maintain external consistency in the face of expressive atomic transactions, Spanner combines Paxos [27] (for vertical replication between geographic zones) with a Two-Phase Commit [5] (for atomic transactions that span shards). The consistency guarantees are achieved by ensuring that commit timestamps respect the bounded error in local clocks, guaranteed by using Spanner's TrueTime API. To analyse these protocols, we specify an operational semantics that is no stronger than the consistency model offered by Spanner.

Providing an operational semantics for scalable distributed systems is technically challenging. Due to the impossibility of perfectly synchronising clocks [25], it is expensive to guarantee a global linear order over events; hence events are partially ordered. The cost of guaranteeing a global total order over events severely constrains the scalability of a system [2]. In Google, BigTable scales to thousands of nodes [14], while Spanner is designed to scale to millions of nodes across hundreds of data centres [16]. In both cases, we must accept weaker notions of consistency, where events are not globally totally ordered.

Further to the above challenge, models that partially order events [32], [31] are typically denotational rather than operational. There exists early work on reconciling the denotational and operational approaches to weaker concurrency models [7], [8], [9], [12], [18]. We approach the problem by expressing causal dependencies between events syntactically in order to structure properly the interactions of a geographically replicated database with its clients.

In Section II, we survey geographically replicated systems used by leading Cloud providers, and compare their consistency guarantees. In Section III, we introduce our operational semantics for a geographically replicated database. In Section IV, we analyse characteristic concurrency scenarios that illustrate the relationship between timestamps and causality.

II. Consistency Models for Geographic Replication

We briefly survey consistency models for scalable distributed databases, then put the consistency models in the context of geographically replicated systems. We emphasise that no model of consistency guaranteed by geographically replicated databases enforces a global total order over committed transactions.

A. Weak Consistency for Scalable Distributed Systems

Consistency models for transactions form a spectrum, where the weakest guarantee is eventual consistency and the strongest is serialisability. In this work, we consider models at least as strong as causal consistency, as relied on by many clients, but no stronger than external consistency as guaranteed by Spanner. Eventual consistency, sequential consistency and serialisability are included for comparison.

Eventual consistency: Eventual consistency allows transactions to proceed despite network partitions. During a network partition, two replicas may apply different updates. If the network returns and no further updates occur, then replicas will eventually reconcile their differences and agree

on an order in which updates are applied at all replicas. Before reconciliation replicas may serve inconsistent values.

Causal Consistency: Lamport [25] emphasises that, due to the impossibility of a global clock in any distributed system, distributed systems partially order events (in this setting events are committed transactions). Lamport's partial ordering of events defines *causal consistency* or *potential causality*, which respects sequentiality demanded by the client and causal relationships due to writes of a value preceding reads of the same value. For example, if you can see the response to a message, then you can see the original message. Thus causal consistency avoids read-write conflicts. Causal consistency can be refined to specify how to handle write-write conflicts, which may occur when two co-located write transactions are unrelated by causal consistency.

Sequential Consistency: Sequential consistency [26] guarantees that all committed transactions over co-located data "appear" to be atomic and totally ordered. Due to co-location of data, most causally consistent systems have some degree of localised sequential consistency, where actions over localities are sequentially ordered. We found that distributed databases that guarantee sequential consistency indeed treat sequential consistency as a local property. Hence committed transactions are globally partially ordered; but when transactions are restricted to a particular location commits are totally ordered.

External Consistency: External consistency, or *linearisability* [22], strengthens sequential consistency by guaranteeing that *non-overlapping* transactions are ordered with respect to the real global time that transactions commit, i.e. if, with respect to real time, one transaction commits before another prepares, then the timestamps assigned to transactions respect the real order of transactions. However, even with external consistency, concurrent transactions spanning different locations need not be totally ordered. Hence, like sequential consistency, external consistency is a local property of a system.

Serialisability: Historically, distributed database systems conformed to strong consistency, or *serialisability* [6]. An execution is serial if it appears that no transactions in the execution execute concurrently. However, early distributed databases were only required to scale to tens of nodes, not thousands or millions of nodes. The overhead incurred by ensuring than no transactions appear to execute concurrently prevents high scalability. Serialisability is the only notion of consistency mentioned in this section that guarantees a global total order over events; hence, all scalable notions of consistency partially order events.

B. Geographic Replication for Cloud Computing

Many Cloud providers have developed systems that geographically replicate data. This section summarises the consistency guarantees offered by geographically replicated databases widely deployed by Yahoo!, Google, Microsoft and Amazon. We also highlight related systems research projects.

Dynamo [17] is Amazon's key-value store based on distributed hash tables. Dynamo offers replication across multiple data centres, but within one geographical region (e.g., Western Europe and Japan and South Korea), where data centres are connected by high speed network links; hence latency is bounded. Dynamo is a pioneer of *eventual consistency*, where replicas may apply updates in different orders but eventually a global order for updates is reconciled. Thus Dynamo maximises partition tolerance and availability, but cannot guarantee causal consistency.

MegaStore [4] builds geographical replication on top of BigTable [14]. Megastore uses Paxos (without a distinguished leader) to replicate tablets across data centres, providing *sequentially consistency* at the level of locality groups (called tablets). However, for transactions that span tablets, *eventual consistency* is assumed. Megastore is used by many of Google's services, e.g. GMail and AppEngine, since these applications require geo-replication. Megastore offers poor write throughput.

PNUTS [15] is a geographically replicated storage service from Yahoo!. PNUTS guarantees *sequential consistency* only at the level of records, where all replicas of a record apply all updates to the record in the same order. Windows Azure Storage [11] is a geographically replicated service. Data is partitioned into locality groups called extents, which contain records. Windows Azure Storage provides *sequential consistency* at the level of extents, where once a commit is acknowledged to the client, later reads from any replica will see the same data.

Further to the above commercial systems, several recent research projects support scalable geographic replication. COPS (Clusters of Order-Preserving Servers) [29], [30] provides a form of weak consistency referred to as *causal consistency with convergent conflict handling*. Causal consistency preserves the causal ordering both provided by the programmer and resulting from reads dependent on writes. Convergent conflict handling ensures that replicas never diverge and conflicting transactions are resolved in the same way at all replicas.

The Walter [34] key-value store geographically replicates data using *parallel snapshot isolation* to provide a form of *causal consistency*. MDCC (Multi Data Centre Consistency) [24] guarantees *sequential consistency* using Generalised Paxos [28]. Finally, Spanner [16], discussed in the next section, is the first system to guarantee *external consistency* at scale.

III. Non-Interleaving Operational Semantics for Spanner

In this section, we introduce a non-interleaving operational semantics for a geographically replicated database inspired by Spanner [16]. In this model, a system consists of a finite set of replicas that cooperate to serve transactions to clients. We explain our syntax for replicas, justify our operational rules, and explain how the rules are combined to specify an appropriate non-interleaving operational semantics.

A. Definition of a Replica in Spanner

All replicas can be uniquely distinguished by their zone and group. Each replica maintains both persistent data in a log and role metadata that is not persisted between incarnations of a replica.

Zones: Each zone represents a distinguished geographical location where replicas are located. Zones are typically located in geographically distributed data centres, however data centres may be divided into more than one zone.

g group z zone Q quorum $\mathtt{p}(log)$ prepared writes $\mathtt{c}(log)$ committed writes

$$log ::= (\mathtt{p}(log), \mathtt{c}(log))$$

$$role ::= \mathtt{idle} \mid \mathtt{l}(Q) \mid \mathtt{s}(z)$$

failed or waiting; group leader; slave

$$Span ::= [log]^{g,z}_{role} \mid Span \parallel Span \mid \mathrm{I}$$

a replica; parallel composition; nothing

$$\coprod_{i \in \emptyset} \left[log^i\right]^{g_i,z_i}_{role_i} \triangleq \mathrm{I} \qquad \coprod_{i \in I+\{j\}} \left[log^i\right]^{g_i,z_i}_{role_i} \triangleq \left[log^j\right]^{g_j,z_j}_{role_j} \parallel \coprod_{i \in I} \left[log^i\right]^{g_i,z_i}_{role_i}$$

Figure 1. The syntax for configurations of replicas (*Span*).

Groups: A Paxos group or simply a *group* consists of a fixed number of replicas located in separate zones. All servers in a group serve a consistent view of the same locality group, called a tablet. Note that some replicas in a group may not have logged the most recent write transactions.

Logs: Committed writes logged through Paxos persist between incarnations of replicas. Each replica persists its log using a distributed file system similar to the Google File System [19]. The log records a history of writes to keys located in the tablet of the replica. From the log the timestamp of the most recently committed write can be determined, which is the *safe* time for reading the tablet. The log also manages prepare records, which contain a lower bound for the timestamp of a write. Prepare records are used for transactions that span several groups.

We use the notational convention that a subscript on a log indicates the safe time, and, if there is further ambiguity, a superscript drawn from a finite set may also disambiguate logs. Note that the real log would also, for each write, keep track of the last vote in Paxos ballots. This is an essential part of the Paxos algorithm that allows voting to continue across failures. However, we do not model this level of granularity.

Roles: Each replica can take on any role in the system. Some replicas are idle, each group has at most one leader at any time, and other active replicas in a group are slaves. A small amount of metadata records the roles of a replica. This metadata is not persisted between incarnations of a replica, hence it is lost upon failure.

Syntax of Replicas: The syntax of replicas indicates the log, group, zone and role, as defined in Fig. 1. A configuration of replicas is the parallel composition of replicas. We introduce the short-cut notation $\coprod$ to represent the parallel composition of several replicas indexed by a finite set.

In Figure 1, g is drawn from a finite set of groups, and z is drawn from a finite set of zones. A *log* has two parts: a prepare log and a commit log. A *prepare log* is a set of *prepare quadruples* (U, g, P, t) consisting of a set of key-value pairs U, a coordinator group g, a set of participant groups P and a prepare timestamp t. A *commit log* records committed transactions as a partial map from keys and timestamps to values. In this way, the history of all transactions is logged, allowing snapshot reads in the past.

We introduce auxiliary operations over logs, to allow the operational semantics to be expressed concisely. We introduce operations:

- $\mathtt{p}(log) + (U, g, P, t)$ and $\mathtt{p}(log) \setminus (U, g, P, t)$ to add and remove transactions from the prepare log;
- $\mathrm{noConflicts}(U, \mathtt{p}(log))$ to check that there is no conflict between a transaction and a prepare log, i.e. no keys in U appear in the prepare log $\mathtt{p}(log)$;
- $\mathtt{c}(log) + (U, t)$ to add transactions to the commit log;
- $\mathrm{canRead}(R, \mathtt{c}(log))$ to perform a snapshot read of key-value-timestamp triples R from a commit log $\mathtt{c}(log)$;
- $\mathrm{merge}_{i \in I}\,(log_i)$ to merge several logs.

Assumptions: For any configuration of replicas, we make the assumptions that no two replicas have both the same group and zone, and no two groups manage the same keys. We assume that each group g has a fixed replication factor $\mathrm{size}(g)$, which is the number of zones in the group.

As in process calculi, we quotient the syntax by a structural congruence that assumes that $(Span, \parallel, \mathrm{I})$ forms a commutative monoid.

B. Metadata for Roles

The main roles that a replica can take are leader, slave or idle. However, the leader may have a distinguished role in read-write transactions spanning more than one group. Each role records some metadata to keep track of the system configuration and state of transactions.

Idle: Replicas may be idle, due to voluntary abdication, failures or network partitions. Idle replicas have lost their role information, however they know what group and zone they represent. It is sufficient to identify idle replicas using the distinguished state $\mathtt{idle}$. An idle replica is written $[log]^{g,z}_{\mathtt{idle}}$.

Slave: A slave is a non-leader member of the current quorum. The slave records the zone of the leader (z_0) that it has granted a lease vote to, using notation $[log]^{g,z}_{\mathtt{s}(z_0)}$.

Leader: A leader is a single distinguished member of a quorum. The leader drives all read-write transactions and sets timestamps for read requests. A leader records the other members of the quorum Q, which is a finite set of zones. For each read-write transaction that spans multiple groups, a leader can be *coordinator leader* or a *participant leader*. For each transaction there is one coordinator leader. All other leaders that are part of the transaction are participant leaders. Note that one leader can simultaneously be the coordinator for one transaction and a participant in another transaction.

$$\frac{}{P \xrightarrow{\mathrm{I}} P} \qquad \frac{P \xrightarrow{U} Q \quad Q \xrightarrow{V} R}{P \xrightarrow{U \triangleleft V} R} \qquad \frac{P \xrightarrow{U} P' \quad Q \xrightarrow{V} Q'}{P \parallel Q \xrightarrow{U \otimes V} P' \parallel Q'}$$

Figure 2. The unit rule and the rules for series and parallel composition.

All leaders have a *transaction manager* that keeps track of fine grained information about their role for each transaction. The transaction manager records fine grained read locks; hence read locks are lost on failure of the leader. To simplify this presentation of our model, we do not include the transaction manager; hence preclude the read phase of a read-write transaction.

C. How to Read the Rules of our Non-Interleaving Semantics

In order to avoid an interleaving semantics, care must be taken when rules are combined, To provide a non-interleaving parallel composition, we allow transitions to be composed both in series and in parallel. The rules in Fig. 2 build up a label consisting of read and write events sequentially ordered using the operator $\triangleleft$, pronounced "before", or composed in parallel using the operator $\otimes$, pronounced "par". The labels represent the interaction of the system with clients. When a rule does not interact with a client, the unit label appears. We quotient labels, such that $(U, \otimes, \mathrm{I})$ forms a commutative monoid and $(U, \triangleleft, \mathrm{I})$ forms a monoid.

When more than one read or write event is involved in a transaction, typically due to a transaction involving more than one key, then the events are joined using the $\otimes$ operator. The $\otimes$ operator indicates that events act together atomically, as in [21] and [23]. For events on labels, we use the notation $(\{(k_0, v_0), (k_1, v_1), \ldots, (k_n, v_n)\}, t)$ to represent the event $\mathtt{W}(k_0, v_0, t) \otimes \mathtt{W}(k_1, v_1, t) \otimes \ldots \mathtt{W}(k_n, v_n, t)$, where t is a commit timestamp.

Consider an example where the following transitions are derivable using the operational semantics.

$$P \xrightarrow{(\{(k_0,v_0),(k_1,u_0)\},5)} Q \parallel R \qquad Q \xrightarrow{\mathtt{W}(k_0,v_1,16)} Q' \quad Q' \xrightarrow{\mathtt{W}(k_0,v_2,21)} Q'' \qquad R \xrightarrow{\mathtt{W}(k_1,u_1,18)} R'$$

The rules in Fig. 2 can be used to combine these transition to obtain the following transition.

$$P \xrightarrow{\begin{pmatrix} \mathtt{W}(k_0,v_0,5) \\ \otimes \\ \mathtt{W}(k_1,u_0,5) \end{pmatrix} \triangleleft \begin{pmatrix} (\mathtt{W}(k_0,v_1,16) \triangleleft \mathtt{W}(k_0,v_2,21)) \\ \otimes \\ \mathtt{W}(k_1,u_1,18) \end{pmatrix}} Q'' \parallel R'$$

In the above transition, two write operations are applied to different keys k_0 and k_1, in one atomic action with timestamp 5. Following this transition, the state is split into two parts, Q and R, responsible for k_0 and k_1, respectively. Two writes are applied to k_0, and one write is applied to k_1. The writes on k_0 are ordered; however, the write on k_1 is not interleaved with the writes on k_0. Although, the timestamp 18 is between 16 and 21, because there is no coordination between the transactions, there is no guarantee that the transaction on k_1 happened before, during or after the transactions on k_0. Thus, the $\otimes$ operator explicitly indicates that the commit events are independent.

The synchronisation followed by divergence into two independent branches above is characteristic of non-interleaving semantics. For example, the heating rule in the CHAM [10] allows state to be split into pieces that are independently evaluated. Building a structure of events on the label is related to proved transition systems where pomsets are constructed on the labels [8]. The independence of concurrent events is also respected by distributed bisimulations [12].

D. Fault Tolerance through Paxos

Data is consistently replicated using the Multi-Paxos [13] variant of the Paxos algorithm. In Multi-Paxos, one replica acts as a long term leader across several transactions. A replica becomes a leader when it has a *quorum* of lease votes, i.e. a majority of replicas acknowledge the replica as the leader. The leader retains its role until it looses its quorum of lease vote, abdicates or fails. All transactions are driven by the leader, by voting in rounds until the quorum of replicas acknowledge the transaction. Upon receiving a quorum of acknowledgements, the leader commits the transaction and notifies its quorum of participant replicas.

A replica may, for any reason and at any point, stop performing its role. This models voluntary abdication, e.g. for maintenance, and involuntary failures e.g. software failures, hardware failures, and network partitions. There are performance differences between the failures, however we model all failures using the same rule (*fail* in Fig. 3).

When a replica receives a quorum of lease votes, that replica becomes the leader for the group. The leader knows it has an exclusive lock on the tablet for as long as it can maintain a quorum of lease votes. In Spanner, all replicas follow the *single vote* rule [16], which prevents more than one replica believing that they are the group leader simultaneously. The adequacy of the single vote rule is guaranteed by using the TrueTime API. The election of a leader with respect to a quorum is modelled by the *elect* rule in Fig. 3. Note that a leader is automatically part of its own quorum.

E. Read-Write Transactions Spanning Groups

Read-write transactions use a hybrid Paxos and Two-Phase Commit protocol, similar to [20]. The Paxos algorithm handles replication between zones, while the Two-Phase Commit algorithm ensures the atomicity of transactions that span groups. We model the protocol at a high level of abstraction, where transitions correspond to the points at which consensus is inevitable.

The read phase: Read-write transactions are driven by the leaders of Paxos groups, since leaders know the most recent committed transactions. First, data is read from the leader replica of the Paxos group that manages the relevant data. The leader acquires read locks from its transaction manager[1]. A read lock prevents the data from being used in another read-write transaction, but does allows the data to be used in a read-only transaction.

[1]For simplicity, we omit the read phase of read-write transactions in this presentation. This design decision helps communicate the combined Paxos and Two-Phase Commit protocol, which is only used in the write phase.

$$\frac{}{[log]^{g,z}_{\texttt{anyrole}} \xrightarrow{\mathrm{I}} [log]^{g,z}_{idle}}\ fail \qquad \frac{|Q|+1 > \mathrm{size}(g)/2 \qquad log' = \mathrm{merge}_{z\in Q\cup\{z_0\}}(log^z)}{[log^{z_0}]^{g,z_0}_{\texttt{idle}} \parallel \coprod_{z\in Q} [log^z]^{g,z}_{\texttt{idle}} \xrightarrow{\mathrm{I}} [log']^{g,z_0}_{1(Q)} \parallel \coprod_{z\in Q} [log']^{g,z}_{\mathsf{s}(z_0)}}\ elect$$

$$\frac{\mathrm{noConflicts}(\mathsf{p}(log_t), U) \qquad log_u = (\mathsf{p}(log_t) + (U, g_0, P, u), \mathsf{c}(log_t)) \qquad t < u}{[log_t]^{g,z_0}_{1(Q)} \parallel \coprod_{z\in Q} [log_t]^{g,z}_{\mathsf{s}(z_0)} \xrightarrow{\mathrm{I}} [log_u]^{g,z_0}_{1(Q)} \parallel \coprod_{z\in Q} [log_u]^{g,z}_{\mathsf{s}(z_0)}}\ prepare$$

$$\frac{\begin{array}{c}\mathrm{noConflicts}\left(\mathsf{p}\left(log^{g_0}_{t_0}\right), U_0\right) \qquad log^{g_0}_u = \left(\mathsf{p}\left(log^{g_0}_{t_0}\right), \mathsf{c}\left(log^{g_0}_{t_0}\right) + (U_0, u)\right) \qquad t_0 < u \\ \forall g \in P \qquad log^{g'} = \left(\mathsf{p}(log^g) \setminus \left(U_g, g_0, P, t_g\right), \mathsf{c}(log^g) + (U_g, u)\right) \qquad t_g < u \\ \left[log^{g_0}_{t_0}\right]^{g_0,z_0}_{1(Q_0,)} \parallel \coprod_{z\in Q_0} \left[log^{g_0}_{t_0}\right]^{g_0,z}_{\mathsf{s}(z_0)} \parallel \coprod_{g\in P}\left([log^g]^{g,z_g}_{1(Q_g,)} \parallel \coprod_{z\in Q_g} [log^g]^{g,z}_{\mathsf{s}(z_g)}\right) \xrightarrow{(U_0,u)\otimes\coprod_{g\in P}(U_g,u)} \left[log^{g_0}_u\right]^{g_0,z_0}_{1(Q_0,)} \parallel \coprod_{z\in Q_0} \left[log^{g_0}_u\right]^{g_0,z}_{\mathsf{s}(z_0)} \parallel \coprod_{g\in P}\left(\left[log^{g'}\right]^{g,z_g}_{1(Q_g,)} \parallel \coprod_{z\in Q_g} \left[log^{g'}\right]^{g,z}_{\mathsf{s}(z_g)}\right)\end{array}}{}\ commit$$

$$\frac{log' = (\mathsf{p}(log) \setminus T, \mathsf{c}(log))}{[log]^{g,z_0}_{1(Q,)} \parallel \coprod_{z\in Q} [log]^{g,z}_{\mathsf{s}(z_0)} \xrightarrow{\mathrm{I}} [log']^{g,z_0}_{1(Q,)} \parallel \coprod_{z\in Q} [log']^{g,z}_{\mathsf{s}(z_0)}}\ abort \qquad \frac{\mathrm{noConflicts}(\mathsf{p}(log), R) \qquad \mathrm{canRead}(\mathsf{c}(log), R)}{[log]^{g,z}_{role} \xrightarrow{R} [log]^{g,z}_{role}}\ read$$

Figure 3. Operational rules for transactions.

The prepare phase: The prepare timestamp is a lower bound on the global timestamp for the transaction. To guarantee this lower bound, the prepare timestamp is chosen by the leader of a group to be strictly greater than the timestamp of any previously committed writes in the group. Write locks are applied at the leader and a prepare quadruple is logged through Paxos. The acceptance of the prepare phase for a participant Paxos group is a synchronisation point for that Paxos group. Finally, the the coordination leader is notified about the prepare timestamp. The prepare phase is modelled by the *prepare* rule in Fig. 3. The condition noConflicts checks that there are no prepared but uncommitted transactions sharing keys with the new transaction.

The commit phase: The coordinator leader applies write locks and determines a timestamp, then logs the updates and timestamp for the coordinator group. The timestamp is chosen such that it is greater than the prepare timestamp for any participant group and definitely greater than the real time the coordinator leader received the update request from the client (the later condition is guaranteed by using the TrueTime API). The coordinator observes *commit wait*, which ensures that the real time of the timestamp has definitely passed (again guaranteed by using the TrueTime API). After the *commit wait*, the coordinator leader notifies clients and participant leaders about the timestamp. Finally, the participant leaders log the timestamp through Paxos.

The commit phase is modelled by the *commit* rule in Fig. 3. The rule may involve several groups: a coordinator group g_0 and participant groups P. The check for conflicts is only performed at the coordinator group, since conflicts at participant groups are identified at the prepare phase. The label uses $\otimes$ to represent the synchronisation of several actions to form a single atomic action, where $\coprod$ represents the repeated application of $\otimes$ indexed by a set, as with parallel composition.

In the above protocol, the coordinator incurs two delays before committing. Firstly, the TrueTime API is used to choose the timestamp such that is is certainly greater than the real time the write request was received by the coordinator. Secondly, commit wait is observed to ensure that the real time when the commit is confirmed is certainly later than the assigned timestamp. If the average error on real time given by TrueTime is $\bar{\epsilon}$, then the expected wait is $2\bar{\epsilon}$. If $\bar{\epsilon}$ is 4ms, then the average wait is 8ms. For comparison, Verzion guarantees[2] that average round trips between New York and London are bounded by 90ms, and between San Francisco and Tokyo by 160ms. Since the wait overlaps with communication between data centres, commit wait has a small impact on performance and guarantees external consistency.

Notice that a read-write transaction involving a single group is a special case of read-write transactions spanning multiple groups, as follows.

$$\frac{\begin{array}{c}\mathrm{noConflicts}(\mathsf{p}(log_t), U) \\ log_u = (\mathsf{p}(log_t), \mathsf{c}(log_t) + (U, u)) \qquad t < u\end{array}}{[log_t]^{g,z_0}_{1(Q,)} \parallel \coprod_{z\in Q} [log_t]^{g,z}_{\mathsf{s}(z_0)} \xrightarrow{(U,u)} [log_u]^{g,z_0}_{1(Q,)} \parallel \coprod_{z\in Q} [log_u]^{g,z}_{\mathsf{s}(z_0)}}$$

In the above special case of the commit rule, there is only one group; hence there is one coordinator leader for the transaction,

[2] http://www.verizonenterprise.com/about/network/latency/, accessed June 2013.

and no participant groups. Thus the transaction can skip the prepare phase. Instead it logs the transaction through Paxos, and picks a timestamp using commit wait as before. Thus transactions over one group use only Paxos, without any Two-Phase Commit.

The above observation shows that the protocol generalises Paxos. Similarly, the protocol also generalises Two-Phase Commit, by taking every group to have only one zone. In this special case, all replicas are leaders hence the Two-Phase Commit proceeds without replication due to Paxos.

Tolerating failures during the Two-Phase Commit: In a Two-Phase Commit protocol, what happens if a participant leader fails after logging a prepare record, but before the second commit is logged? The prepare record is logged by Paxos so prepared writes are known to the new leader, hence the leader can continue the transaction. However, the read locks have been lost. Fortunately, the timestamp has already been picked and commit wait and separation of lease intervals guarantee that the next timestamp of a read-write transaction to locked locations will be greater than the current timestamp, thus will wait to read the result of the current writes. Furthermore, the client sends keep alive messages to the participant leaders until after the commit wait. The timestamp has certainly been logged at the coordinating group hence a new leader can obtain the timestamp and complete the transaction. From this analysis we know that commit is inevitable once the timestamp has been picked; thereby we justify that the commit rule is a synchronisation point.

If the client or coordination leader finds that a participant group does not respond at the prepare phase, then the transaction should abort. This is done by revoking prepare logs and releasing locks. This is modelled by the *abort* rule in Fig. 3. An abort is logged through Paxos, hence a quorum must accept the abort.

Only the leader holds read locks, so the read locks are lost when the leader fails. Writes can be continued using the prepare logs, but it is possible that a concurrent quick read-write transaction can overwrite the data that was read by the first transaction and apply a lower timestamp. This would result in an inconsistency; e.g., if both transactions deduct a sum from a customers balance then the effect is that the quicker balance deduction is completely lost.

F. Read-only transactions

A read-only transaction is defined by its extent and the timestamp at which it should execute. Read-only transactions can be served by any replica that is sufficiently up to date. The *read* rule in Fig. 3 models a read operation.

For read-only transactions, there is a conflict if any keys to be read are in the prepare phase of a read-write transaction and the timestamp for the snapshot read is not earlier than the prepare timestamp. In this case, the read-only transaction would have to wait to execute. To read the most recent version of data, the client must contact the leader to find the latest timestamp that preserves external consistency. For further details about negotiating read timestamps we refer to [16].

IV. Scenarios for Concurrent Transactions

We illustrate our operational semantics, by providing examples of characteristic concurrency scenarios. The main point of interest is the causal dependencies, as seen by clients, and how they relate to the order of timestamps.

A. Example: Concurrent Transactions with Shared Keys

Suppose that two spanning read-write transactions share resources. The transaction that first begins must commit before the second transaction can start. This is because the second transaction cannot obtain write locks and begin a prepare log at the shared groups. The operations are therefore sequentially ordered at the shared group.

In the following example, assume that key k_i is managed by group g_i.

$$\begin{array}{l}\left[log_5^0\right]_{1(\{z_0\})}^{g_0,z_0} \parallel \\ \left[log_9^1\right]_{1(\{z_1\})}^{g_1,z_1} \parallel \\ \left[log_8^2\right]_{1(\{z_2\})}^{g_2,z_2}\end{array} \quad \begin{array}{l}\left(\mathtt{W}(k_0,v_0,12) \between \mathtt{W}\left(k_1,v_1^0,12\right)\right) \between \\ \left(\mathtt{W}\left(k_1,v_1^1,23\right) \between \mathtt{W}(k_2,v_2,23)\right)\end{array} \longrightarrow \quad \begin{array}{l}\left[log_{12}^0\right]_{1(\{z_0\})}^{g_0,z_0} \parallel \\ \left[log_{23}^1\right]_{1(\{z_1\})}^{g_1,z_1} \parallel \\ \left[log_{23}^2\right]_{1(\{z_2\})}^{g_2,z_2}\end{array}$$

By the operational semantics we can expect the following logs in the final state:

$$\begin{aligned} log_{12}^0 &= \left(\mathsf{p}\left(log_5^0\right), \mathsf{c}\left(log_5^0\right) + (k_0, v_0, 12)\right) \\ log_{23}^1 &= \left(\mathsf{p}\left(log_9^1\right), (\mathsf{c}\left(log_9^1\right) + (k_1, v_1^0, 12)) + (k_1, v_1^1, 23)\right) \\ log_{23}^2 &= \left(\mathsf{p}\left(log_{12}^2\right), \mathsf{c}\left(log_{12}^2\right) + (k_1, v_2, 23)\right) \end{aligned}$$

The order of timestamps on the transactions are guaranteed by the rules. Also, notice that the two logs are safe up to timestamp 23, but the first log is safe up to timestamp 12.

B. Example: Conflicts between Causal Ordering and Timestamps

Suppose that two spanning read-write transactions both prepare before either commits. Furthermore, suppose that neither share resources, but they do share a participant group. In this case, there is no locking conflict, nor any requirement that one transaction commits before the other. Both transactions are prepared by the shared group, and distinct prepare timestamps are assigned to each transaction. The coordinators for each transaction choose timestamps independently of each other, taking only the prepare timestamps and the most recent transaction at each coordinator into account. Thus, there is no correlation between the two timestamps. Furthermore, there is no guarantee about the causal order of the transaction, but a causal order will be enforced by the shared group.

As a consequence of the above, it is possible to have two writes on distinct keys (managed by the same group), such that one operation appears to be causally dependent on the other, but the earlier committed write has an equal or higher timestamp. Stated otherwise, a group can participate in multiple simultaneous non-conflicting two phase commits without any guarantees on the timestamp. However, operations are (randomly) sequentially ordered by the shared groups.

The above scenario is illustrated by the following example. Assume that key k_0 is managed by group g_0, keys k_1^0 and k_1^1 are distinct keys managed by group g_1, and key k_2 is managed by group g_2.

$$\begin{array}{l}\left[log^0_{t^0_0}\right]^{g_0,z_0}_{1(\{z_0\})} \parallel \\ \left[log^1_{t^1_0}\right]^{g_1,z_1}_{1(\{z_1\})} \parallel \\ \left[log^2_{t^2_0}\right]^{g_2,z_2}_{1(\{z_2\})}\end{array} \xrightarrow{\begin{array}{l}\left(\mathtt{W}(k_0,v_0,t_1) \between \mathtt{W}\left(k^0_1,v^0_1,t_1\right)\right) \lhd \\ \left(\mathtt{W}\left(k^1_1,v^1_1,t_2\right) \between \mathtt{W}(k_2,v_2,t_2)\right)\end{array}} \begin{array}{l}\left[log^0_{t_1}\right]^{g_0,z_0}_{1(\{z_0\})} \parallel \\ \left[log^1_{t_1}\right]^{g_1,z_1}_{1(\{z_1\})} \parallel \\ \left[log^2_{t_2}\right]^{g_2,z_2}_{1(\{z_2\})}\end{array}$$

where the logs are updated as follows:

$$\begin{aligned} log^0_{t_1} &= \left(\mathrm{p}\left(log^0_{t^0_0}\right), \mathrm{c}\left(log^0_{t^0_0}\right) + (k_0, v_0, t_1)\right) \\ log^1_{t_1} &= \left(\mathrm{p}\left(log^0_{t^1_0}\right), (\mathrm{c}\left(log^0_{t^1_0}\right) + (k^0_1, v^0_1, t_1)) + (k^1_1, v^1_1, t_2)\right) \\ log^2_{t_2} &= \left(\mathrm{p}\left(log^0_{t^1_0}\right), \mathrm{c}\left(log^0_{t^2_0}\right) + (k_1, v_2, t_2)\right) \end{aligned}$$

Due to the choice of prepare timestamps, we know that $t^0_0 < t_1$, $t^1_0 < t_1$, $t^1_0 < t_2$ and $t^2_0 < t_2$. However, we do not have any guarantees about the relationship between t_1 and t_2. Hence it is possible that $t_2 < t_1$, despite the causal ordering indicated by $\lhd$, which intuitively should mean "happens before". This possibility is reflected in the safe time of group g_1.

This conflict does not violate the guarantees offered by Spanner. In particular, Spanner guarantees that if one transaction begins after another one has committed, then the timestamp of the committed transaction is lower than the prepared transaction. In the above scenario, neither transaction commits before the other is prepared. Indeed, related work on the relationship between time and causality [1], argues that there is no need to ban such ill-timed sequences of events when there are no causal dependencies from the perspective of the programmer (the potential causality). The scenario, above only occurs if clients have issued independent requests without causal dependencies. The above scenario could be explicitly resolved using a conflict resolution scheme, such as commutative replicated data types [33], [28]. Such an approach, would allow the above transactions to be treated causally independently. An operational semantics that respects such conflict resolution schemes is future work.

Note that, if both transactions share the same coordination leader, then the timestamps and causal ordering of the transactions match, since the coordination leader can ensure that the timestamp of the second transaction committed is later than the first. In this case, the observable effect is similar to Example IV-A, with the only difference being that two separate keys are updated at g_1.

C. Example: Transactions over Disjoint Groups

Consider when two concurrent writes occur using disjoint groups. There is no dependency between the two transactions.

Assume that k_0 and k_1 are managed by groups g_0 and g_1 respectively.

$$\begin{array}{l}\left[log^{t^0_0}_{t^0_0}\right]^{g_0,z_0}_{1(\{z_0\})} \parallel \\ \left[log^{t^1_0}_{t^1_0}\right]^{g_1,z_1}_{1(\{z_1\})}\end{array} \xrightarrow{\mathtt{W}\left(k_0,v_0,t^0_1\right) \between \mathtt{W}\left(k^1_1,v_1,t^1_1\right)} \begin{array}{l}\left[log_{t^0_1}\right]^{g_0,z_0}_{1(\{z_0\})} \parallel \\ \left[log_{t^1_1}\right]^{g_1,z_1}_{1(\{z_1\})}\end{array}$$

where $log_{t^i_1} = \left(\mathrm{p}\left(log_{t^i_0}\right), \mathrm{c}\left(log_{t^i_0}\right) + (k_i, v_i, t^i_1)\right)$ for $i \in \{0, 1\}$.

In the above scenario, $t^0_0 < t^0_1$ and $t^1_0 < t^1_1$. No other relationships between timestamps are guaranteed. Furthermore, there is no causal relationship between the two writes. They are causally independent as indicated by $\between$.

D. Example: Independent Read-only and Read-write Transactions

Suppose that a read-only transaction is requested during a transaction, where resources are not write locked, but are located in the same group. The read-only transaction executes at any replica that is sufficiently up-to-date.

In the following example, assume that k_0 and k_1 distinct keys managed by group g with replication factor 3. Also assume that u is such that $u \leq t^2_0$, i.e. a timestamp in the past that the idle replica can serve.

$$\begin{array}{l}\left[log_{t^0_0}\right]^{g,z_0}_{1(\{z^0_0,z^1_0\})} \parallel \\ \left[log_{t^1_0}\right]^{g,z_1}_{\mathtt{s}(z_0)} \parallel \left[log_{t^2_0}\right]^{g,z_2}_{\mathtt{idle}}\end{array} \xrightarrow{\substack{\mathtt{W}\left(k_0,v_0,t^0_1\right) \between \\ \mathtt{R}\left(k^1_1,v_1,t^1_1\right)}} \begin{array}{l}\left[log_{t_1}\right]^{g,z_0}_{1(\{z^0_0,z^1_0\})} \parallel \\ \left[log_{t_1}\right]^{g,z_1}_{\mathtt{s}(z_0)} \parallel \left[log_{t^2_0}\right]^{g,z_2}_{\mathtt{idle}}\end{array}$$

where $log_{t_1} = \left(\mathrm{p}\left(log_{t^0_0}\right), \mathrm{c}\left(log_{t^0_0}\right) + (k_0, v_0, t^0_1)\right)$.

In the above scenario, the read and write to the same group are causally independent, since they are served by disjoint replicas.

V. Conclusion

To date, Google Spanner offers the strongest consistency guarantees at scale. None the less, the consistency guarantees are significantly weaker than the consistency guarantees imposed by an interleaving operational semantics. For a scalable system to guarantee a global order over events, as imposed by an interleaving operational semantics, the performance overhead is prohibitive. However, we can provide a non-interleaving operational semantics for transactions that respects weaker consistency models, which are sufficient from the perspective of clients.

Both Spanner and our operational semantics guarantees external consistency, which ensures that, if a transaction is committed before another transaction starts, then the timestamps of the transactions respect the causal order of transactions, as illustrated in Example IV-A. A subtlety is illustrated by Example IV-B, where, if concurrent transactions prepare before either commits, it is possible that causal dependencies can conflict with the order of timestamps. A second subtlety is that when there is no potential causality, or co-location relating transactions, then, for concurrent transactions, the real time of a commit and the timestamps of commit are unrelated (see Examples IV-C and IV-D).

This work is the first to provide an operational semantics for transactions over a geographically replicated database inspired by Spanner. This operational semantics enables a formal analysis for such systems, which is useful for understanding the complex system design of related systems. Due to the necessity of geographic replication for the high availability and durability of data hosted by Cloud providers, we expect interest in understanding such systems to extend beyond the field of operational semantics.

Our primary contribution in this work is a concrete example of a modern system, where the operational semantics is certainly non-interleaving. Future work includes using the

calculus to verify formal properties of geographically replicated databases, including external consistency and weak progress. The study of the operational semantics of geo-replicated databases, can be used to investigate further the foundations of programming languages that interact with such systems.

Acknowledgements: The work was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-II-ID-PCE-2011-3-0919.

References

[1] Aceto, L., Murphy, D.: Timing and causality in process algebra. Acta Informatica 33(4), 317–350 (1996)

[2] Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. Distributed Computing 9(1), 37–49 (1995)

[3] Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: Above the Clouds: A Berkeley view of Cloud Computing. Communications of the ACM 53(4), 50–58 (2010)

[4] Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., Léon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: Proc. of CIDR. pp. 223–234 (2011)

[5] Berger, M., Honda, K.: The two-phase commitment protocol in an extended π-calculus. Electronic Notes in Theoretical Computer Science 39(1), 21–46 (2003)

[6] Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. 13(2), 185–221 (1981)

[7] Berry, G., Boudol, G.: The chemical abstract machine. Theoretical computer science 96(1), 217–248 (1992)

[8] Boudol, G., Castellani, I.: Permutation of transitions: An event structure semantics for CCS and SCCS. In: de Bakker, J., de Roever, W., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS, vol. 354, pp. 411–427. Springer (1989)

[9] Boudol, G., Castellani, I., Hennessy, M., Nielsen, M., Winskel, G.: Twenty years on: Reflections on the CEDISYS project. combining true concurrency with process algebra. In: Concurrency, Graphs and Models, pp. 757–777. Springer (2008)

[10] Boudol, G., Castellani, I.: Permutation of transitions: An event structure semantics for CCS and SCCS. In: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Lecture Notes in Computer Science, vol. 354, pp. 411–427. Springer (1989)

[11] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., et al.: Windows Azure Storage: a highly available cloud storage service with strong consistency. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 143–157. ACM (2011)

[12] Castellani, I., Hennessy, M.: Distributed bisimulations. J. Assoc. Comput. Mach. 36, 887–911 (1989)

[13] Chandra, T., Griesemer, R., Redstone, J.: Paxos made live — an engineering perspective. In: Proceedings of the 26th ACM Symposium on Principles of Distributed Computing-PODC. vol. 7 (2007)

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

[15] Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment 1(2), 1277–1288 (2008)

[16] Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation. pp. 251–264. OSDI'12, USENIX Association, Berkeley, CA, USA (2012)

[17] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: ACM Symposium on Operating Systems Principles: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. vol. 14, pp. 205–220 (2007)

[18] Degano, P., De Nicola, R., Montanari, U.: On the consistency of truly concurrent operational and denotational semantics. In: Logic in Computer Science, 1988. LICS'88., Proceedings of the Third Annual Symposium on. pp. 133–141. IEEE (1988)

[19] Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. In: ACM SIGOPS Operating Systems Review. vol. 37, pp. 29–43. ACM (2003)

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

[21] Guglielmi, A.: A system of interaction and structure. ACM Transactions on Compututational Logic 8(1) (2007)

[22] Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) 12(3), 463–492 (1990)

[23] Horne, R., Sassone, V.: A verified algebra for Read-Write Linked Data. Science of Computer Programming (2013), http://dx.doi.org/10.1016/j.scico.2013.07.005

[24] Kraska, T., Pang, G., Franklin, M.J., Madden, S.: MDCC: Multi-data center consistency. In: Proceedings of April 15-17, 2013, Prague, Czech Republic. pp. 113–126. ACM (2013)

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

[26] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. Computers, IEEE Transactions on 100(9), 690–691 (1979)

[27] Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems (TOCS) 16(2), 133–169 (1998)

[28] Lamport, L.: Generalized consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research (2005)

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

[30] Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation. pp. 313–328. NSDI'13, USENIX Association, Berkeley, CA, USA (2013)

[31] Mazurkiewicz, A.: Concurrent program schemes and their interpretations. DAIMI Report Series 6(78) (1977)

[32] Pratt, V.: Modeling concurrency with partial orders. International Journal of Parallel Programming 15(1), 33–71 (1986)

[33] Preguica, N., Marques, J.M., Shapiro, M., Letia, M.: A commutative replicated data type for cooperative editing. In: Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on. pp. 395–403. IEEE (2009)

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