Encylopedia of Database Systems, 2016 edition

Optimistic Replication and Resolution

Marc Shapiro

Sorbonne-Universités-UPMC-LIP6 & Inria Paris
http://lip6.fr/Marc.Shapiro/

1 Synonyms

Asynchronous Replication; Lazy replication; Optimistic replication; Reconciliation-based data replication.

The term "optimistic replication" is prevalent in the distributed systems and distributed algorithms literature. The database literature prefers "lazy replication."

2 Definition

Data replication places physical copies of a shared logical item onto different sites. *Optimistic replication* (OR) [16] allows a program at some site to read or update the local replica at any time. An update is *tentative* because it may conflict with a remote update. Such conflicts are resolved after the fact, in the background. Replicas may *diverge* occasionally, but are expected to converge eventually (see entry "Eventual Consistency").

OR avoids the need for distributed coordination prior to using an item. It allows a site to execute even when remote sites have crashed, when network connectivity is poor or expensive, or while disconnected from the network.

The defining characteristic of OR is that any communication between sites occurs in the background, after local commitment, i.e., off the critical path of the application.

OR enables parallelism, and updates occur and propagate quickly. The OR approach is well adapted to distributed databases over slow or failure-prone networks, and OR is essential to be able to access remote data with high availability. Prominent examples include *geo-replication* (see entry "Multi Datacenter Consistency") and mobile computing scenarios. Indeed, the CAP Theorem states that, in a network that is prone to disconnection, it is not possible to ensure both strong consistency and availability. When availability is paramount, for instance in e-commerce applications, this leads to the choice of the weak consistency levels (e.g., see entry "Eventual Consistency") supported by OR.

Disconnected operation, the capability to compute while disconnected from a data source, e.g., in mobile computing, requires OR. In *computer-supported cooperative work*, OR enables a user to temporarily insulate himself from other users. In *cloud computing*, OR enables the system to remain available for reads and writes even when the network is slow or partitioned away.

3 Historical Background

(The vocabulary used in this history is defined in Section 4.)

The first historical instance of OR is Johnson's and Thomas's Last-Writer-Wins replicated database (1976).

Usenet News (1979) supports a large-scale ever-growing database of (read-only) items, posted by users all over the world. A Usenet site connects infrequently (e.g., daily) with its peers. New items are flooded to other sites and are delivered in arbitrary order. Users occasionally observe ordering anomalies, but this is not considered a problem. However, system administrators must deal manually with conflicts over administrative operations.

In 1984, Wuu and Bernstein's replicated mutable key-value-pair database uses an operation log, transmitted by an *anti-entropy* protocol: site A sends to site B only the tail of A's log that B has not yet seen [22]. Concurrent operations either commute or have a natural semantic order; nonconcurrent operations execute in happens-before order.

The Lotus Notes system (1988) supports cooperative work between mobile enterprise users. It replicates a database of discrete items in a peerto-peer manner. Notes is state-based and uses a Last-Writer Wins policy. A deleted item is replaced by a *tombstone*.

Several file systems, designed in the early 1990s to support disconnected work, e.g., Coda [8], are state based and use version vectors for conflict detection. Conflicts over some specific object types (e.g., directories or mailboxes) cause automatic resolver programs to run. The others must be resolved manually.

The Computer-Supported Cooperative Work (CSCW) community invented (1989) a form of OR called Operational Transformation (OT). Conflicting operations are transformed, by modifying their arguments, in order to execute in arbitrary but causal order [19].

Golding [5] (1992) studies a replicated database of mutable key-value pairs. This system purges an operation from the log when it can prove that it was delivered to all sites. Consistency is ensured by defining a total order of operations.

Bayou (1994–1997) is a seminal general-purpose database for mobile users [12]. Bayou is operation based and uses an anti-entropy protocol. Each site executes transactions in arbitrary order; transactions remain tentative. The eventual serialisation order is the order of execution at a designated *primary* site. Other sites roll back their tentative state, and re-execute committed transactions in commit order.

In 1996, Gray et al. argued that OR databases cannot scale [7], because conflict reconciliation is expensive, conflict probability rises as the third power of the number of nodes, and the wait probability further increases quadratically with disconnection time.

In 1999, Breitbart et al. [2] describe a partially replicated database that uses a form of OR. Each item has a designated primary site and may be replicated at any number of secondary sites. A read may occur at a secondary site but a write must occur on the primary. It follows that write transactions update a single site. If transactions are serialisable at each site, and update propagation is restricted to avoid ordering anomalies, then transactions are serialisable despite lazy propagation.

Cloud computing has sparked a new interest in OR. In order to avoid synchronisation, which is bad for performance and for fault-tolerance, AP (Available under Partition) databases are designed in an OR style, supporting only weakly-consistent key-value storage, such as Last-Writer Wins (Cassandra) or Multi-Value Register (Dynamo).

Geo-replication (see entry "Multi Datacenter Consistency") places database replicas at several data centers around the globe, for improved responsiveness and fault tolerance. Although a replica may be strongly consistent internally, geo-replication typically uses OR between data centers to ensure availability. Examples include Walter [18], Eiger [10], or Riak.

Around 2010, several researchers proposed the concept of a Replicated Data Type (RDT) [3, 14, 15, 17]. An RDT is similar to an ordinary data type; for instance, read-write register, set, map, graph, etc., may constitute RDT types. Abstractly, an RDT is similar to the corresponding ordinary abstract data type; for instance, the interface to a register RDT might have *read* and *assign* methods, whereas a set RDT would have methods for testing whether an element is a member of the set and for adding and removing elements to/from the set. Internally, an RDT is replicated, to provide reliability, availability, and responsiveness. Encapsulation hides the details of replication and conflict resolution.

4 Foundations

Figure 1 depicts a logical item x, concretely replicated at three different sites. In OR, any site may submit or initiate a transaction reading or writing the local replica. If the transaction succeeds locally, the system propagates it to other sites and replays the transaction on the remote sites, in a lazy manner, in the background. Local execution is tentative and may be resolved against a conflict with a concurrent remote transaction. (The happens-before and concurrency relations are defined formally by Lamport [9]. Transaction A happens-before B, if OR replayed B was initiated on some site after A executed at that site. Two transactions are concurrent if neither happens-before the other.)

OR is opposed to pessimistic (or eager) replication, where a local transaction terminates only when it commits globally. Pessimistic replication logically establishes a total order for committed transactions, at the latest when each transaction terminates. In

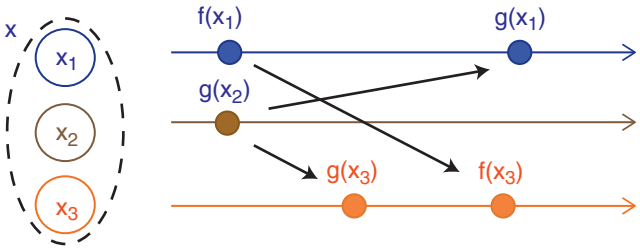

Figure 1: Three sites with replicas of logical item x. Site 1 initiates transaction f; Site 2 initiates g. The system propagates and replays on remote sites. Site 3 executes in the order $g; f$, whereas Site 1 replays f before g. Eventually, Site 2 will also execute f.

contrast, OR generally relaxes the ordering requirements and/or converges to a common order a posteriori. The effects of a tentative transaction can be observed; thus OR protocols may violate the isolation property and allow cascading aborts and retries to occur.

4.1 Transmitting and Replaying Updates

In OR, updates are propagated lazily, in the background, after the transaction has terminated locally. Transmission usually uses peer-to-peer epidemic or anti-entropy techniques (see entry on "Peer-to-Peer Content Distribution").

There are two main approaches to update transmission and delivery. In the state-based approach, a sender transmits the updated (after-values) of the object; a receiver merges the received value into its local state. In the operation-based approach, the sender transmits the program of the update transaction itself; a receiver replays the code of the transaction on its local replica.

The state-based approach is often perceived as being the simpler of the two. In the common case of last-writer-wins, state-based merge often reduces to overwriting the local replica with the received value; this is guaranteed to be deterministic. In the more general case, the merge procedure must be carefully designed to ensure convergence. The state-based approach tolerates unreliable and out-of-order delivery. However, if the replicated object is large, then state-based transmission is expensive, and replay is subject to false conflicts.

Conversely, the cost of transmitting an operation is often very small, similar to a remote procedure call.

Furthermore, logical operations are more likely to commute than writes; thus operationbased replay typically causes fewer aborts. However, the operation-based approach assumes communication layer, that ensures reliable, exactly-once delivery in happened-before order.

4.2 Conflicts

Each transaction taken individually is assumed correct (the C of the ACID properties), i.e., it maintains semantic invariants. For example, ensuring that a bank account remains positive, or that a person is not scheduled in two different meetings at the same time.

As is clear from Figure 1, concurrent transactions may be delivered to different sites in different orders (see section "Scheduling Transactions Content and Ordering"). However, requires that local schedules be equivalent. In this respect, one may classify pairs of concurrent transactions as commuting, non-commuting, and antagonistic. Transactions conflict if they are mutually noncommuting or mutually antagonistic.

The relative execution order of *commuting* transactions is immaterial; they require no remote synchronisation. Formally, two transactions $T1$ and $T2$ commute if execution order $T1; T2$ returns the same results to the user and leaves the database in the same state as the order $T2; T1$. For instance, depositing €10 in a bank account commutes with a depositing €20 into the same account and also commutes with withdrawing €100 from an independent account. If running concurrent transactions together would violate an invariant, they are said antagonistic. Safety requires aborting one or the other (or both).

For instance, if $T1$ schedules me in a meeting from 10:00 to 12:00, and $T2$ schedules a different meeting from 11:00 to 13:00, they are antagonistic since no combination of both $T1$ and $T2$ can be correct. If two transactions are non-commuting and neither is aborted, then their relative execution order must be the same at all sites. Consider for instance $T1$ = "transfer balance to savings" and $T2$ = "deposit €100." Both orders $T1; T2$ and $T2; T1$ make sense, but the result is clearly different. There must be a system-wide consensus on the order $T1; T2$ or $T2; T1$.

4.3 Conflict Resolution and Reconciliation

Conflict resolution rewrites or aborts transactions to remove conflicts. Conflict resolution can be either manual or automatic. Manual conflict resolution simply allows conflicting transactions to proceed, thereby creating conflicting versions; it is up to the user to create a new, merged version.

Reconciliation detects and repairs conflicts and combines non-conflicting updates. Thus transactions are tentative, i.e., a tentatively successful transaction may have to roll back for reconciliation purposes. OR resolves conflicts a posteriori (whereas pessimistic approaches avoid them a priori).

In many systems, data invariants are either unknown or not communicated to the system. In this case, the system designer conservatively assumes that, if concurrent transactions access the same item, and one (or both) writes the item, then they are antagonistic. Then, one of them must abort, or both.

A few systems, such as Bayou [21], IceCube [13] or CISE [6] support an application-specific check of invariants. Bailis et al. [1] shows that application-level enforcement of invariants is error prone.

4.4 Last Writer Wins

When transactions consist only of reads and assignments, a common approach is to ensure a global precedence order.

For instance, many replicated file systems follow the "Last Writer Wins" (LWW) approach. Files have timestamps that increase with successive versions. When the file system encounters two concurrent versions of the same file, it overwrites the one with the smallest timestamp with the "younger" one (highest timestamp). The write with the smallest timestamp is lost; this approach violates the Durability property of ACID.

4.5 Semantic Resolvers

A resolver is an application-specific conflict resolution program that automatically merges two conflicting versions of an item into a new one. For example, the Amazon online book store resolves problems with a user's "shopping cart" by taking the union of any concurrent instances. This maximises availability despite network outages, crashes, and the user opening multiple sessions.

A resolver should ensure that the conflicting transactions are made to commute. In a state-based approach, a resolver generally parses the item' s state into small, independent sub-items. Then it applies an LWW policy to updated and tombstoned sub-items and a union policy to newly created sub-items.

The most elaborate example exists in Bayou. A Bayou transaction has three components: the dependency check, the write, and the merge procedure. The former is a database query that checks for conflicts when replaying. The write (a SQL update) executes only if the consistency check succeeds. If it fails, the merge procedure (an arbitrary but deterministic program) provides a chance to fix the conflict. However, it is very difficult to write merge procedures in the general case.

4.6 Operational Transformation

In Operational Transformation (OT), conflicting operations are transformed [19]. Consider two users editing the shared text "abc". User 1 initiates *insert("X",2)* resulting in "aXbc" and User 2 initiates *delete(3)*, resulting in "ab". When User 2 replays the insert, the result is "aXb" as expected. However for User 1 to observe the same result, the delete must be transformed to *delete(2)*.

In essence, the operations were specified in a non-commuting way, but transformation makes them commute. OT assumes that transformation is always possible. The OT

literature focuses on a simple, linear, shared edit buffer data type, for which numerous transformation algorithms have been proposed.

OT requires two correctness conditions, often called TP1 and TP2. TP1 requires that, for any two concurrent operations A and B, running "A followed by {B transformed in the context of A}" yield the same result as "B followed by {A transformed in the context of B}." TP1 is relatively easy to satisfy and is sufficient if replay is somehow serialised. TP2 requires that transformation functions themselves commute. TP2 is necessary if replay is in arbitrary order, e.g., in a peer-to-peer system. The vast majority of published non-serialised OT algorithms have been shown to violate TP2 [11].

4.7 Conflict-Free Replicated Data Types (CRDTs)

The common memory-cell data model is not well suited to an OR system, since concurrent assignments do not commute. OR will benefit from a data model where concurrent updates can be merged, ensuring that replicas converge without requiring synchronisation or consensus. For instance, concurrent increment and decrement operations to a shared counter can be naturally merged, because they commute.

Conflict-free Replicated Data Types (CRDTs) generalise this approach [17]. A CRDT is an abstract data type that extends some sequential type, and encapsulates algorithms ensuring that concurrent updates are merged deterministically and are guaranteed to converge. Thanks to this property, replicas of a CRDT can be updated in parallel without synchronisation. CRDT types include registers, counters, sets, maps, graphs and sequences.

When used in a sequential way, a CRDT type behaves just like its sequential counterpart. Furthermore, if two updates commute in the sequential specification, then executing the same two updates concurrently will converge to the same state. For instance, the result of concurrently adding elements e and f to some CRDT set are the same as adding them in any order. This means that a CRDT type is plug-in replacement for the corresponding sequential data type.

The key challenge in CRDT design is providing a sensible concurrency semantics for updates that do not commute in the sequential specification. Thus, the concurrent specification of concurrently adding and removing the same element e to a set might be "add wins," i.e., e appears in the set; but, it could equally be "remove wins" or "highest timestamp wins" depending on application requirements.

4.8 Scheduling Transactions Content and Ordering

In order to capture any causal dependencies, transactions execute in happens-before order i.e., causal consistency. As explained in Section "Conflicts," antagonistic transactions cause aborts, and non-commuting transactions must be mutually ordered. This so-called serialisation requires a consensus, violating the availability requirement.

Whereas pessimistic approaches serialise a priori, most OR systems execute transactions tentatively in arbitrary order and serialise a posteriori. Some executions are rolled back; cascading aborts may occur.

A prime example is the Bayou system [21]. Each site executes transactions in the order received. Eventually, the transactions reach a distinguished primary site. If the dependency checks of a transaction fails at the primary, then it aborts everywhere. Transactions that succeed commit and are serialised in the execution order of the primary.

The IceCube system showed that it is possible to improve the user experience by scheduling operations intelligently [13]. IceCube is a middleware that relieves the application programmer from many of the complexities of reconciliation. Multiple applications may coexist on top of IceCube. Applications expose semantic annotations, indicating which operation pairs commute or not, are antagonistic, dependent, or have an inherent semantic order. The user may create atomic groups of operations from different applications. The IceCube scheduler performs an optimisation procedure over a batch of operations, minimising the number of aborted operations. The user commits any of the alternative schedules proposed by the system.

4.9 Freshness of Replicas

Applications may benefit from *freshness* or quality-of-service guarantees, e.g., that no replica diverges by more than a known amount from the ideal, strongly consistent state. Such guarantees come at the expense of decreased availability.

The Bayou system proposes qualitative "session guarantees" on the relative ordering of operations [20]. For instance, Read-Your-Writes (RYW) guarantees that a read observes the effect of a write by the same user, even if initiated at a different site. RYW ensures, that immediately after changing his password, a user can log in with the new password. Other similar guarantees are Monotonic-Reads, Writes-Follow-Reads, and Monotonic-Writes. The conjunction of their guarantees is equivalent to causal consistency.

Systems such as TACT control replica divergence quantitatively [23]. TACT provides a time-based guarantee, allowing an item to remain stale for only a bounded amount of time. TACT implements this by pushing an update operation to remote replicas before the time limit elapses. TACT also provides "order bounding," i.e., limiting the number of uncommitted operations: when a site reaches a user-defined bound on the number of uncommitted operations, it stops accepting new ones.

Finally, TACT can bound the difference between numeric values. For this, each replica is allocated a quota. Each site estimates the progress of other sites, using vector clock techniques. The site stops initiating operations once its cumulative modifications, or the estimated remote updates to the item, reach the quota. At that point, the site pushes its updates and pulls remote operations. For example, a bank account might be replicated at ten sites. To guarantee that the balance observed is within €50 of the truth, each site's quota is $e50/10 = e5$. Whenever the difference estimated by a site reaches €5, it synchronises with the others.

4.10 Optimistic Replication Versus Optimistic Concurrency Control

The word "optimistic" has different, but related, meanings when used in the context of replication and of concurrency control.

Optimistic replication (OR) means that updates propagate lazily. There is no a priori total order of transactions. There is no point in time where different sites are guaranteed to have the same (or equivalent) state. Cascading aborts are possible.

Optimistic concurrency control (OCC) means that conflicting transactions are allowed to proceed concurrently. However, in most OCC implementations, a transaction validates before terminating. A transaction is serialised with respect to concurrent transactions, at the latest when it terminates, and cascading aborts do not occur.

5 Key Applications

Usenet News pioneered the OR concept, allowing to share write-only information over a slow, but cheap network using dial-up modems over telephone lines.

Mobile users want to be able to work as usual, even when disconnected from the network. Thus, mobile computing is a key driver for OR applications. Systems designed for disconnected work that use OR include the Coda file system [8], the Bayou shared database [21], or the Lotus Notes collaborative suite.

Another important application area is Computer-Supported Collaborative Work. In this domain, users must be able to update shared artefacts in complex ways without interfering with one another. OR allows a user to insulate himself temporarily from other users. A key example is the Concurrent Versioning System (CVS), which enables collaborative authoring of computer programs [4]. Bayou and Lotus Notes, just cited, are also designed for collaborative work.

OR is used for high performance and high availability in large-scale web sites. A well-known example is Amazon's "shopping cart," which is designed to be highly available, even if the same user connects to several instances of the Amazon store discussed earlier. For this reason, many NoSQL databases embrace the Available under Partition (AP) option (cf. entry on "CAP Theorem"), which is OR.

6 Cross-References

- CAP Theorem
- Eventual Consistency
- Mobile Computing
- Multi Datacenter Consistency

- NoSQL Databases
- Peer-to-Peer Content Distribution
- Replicated Data Types

References

[1] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 1327–1342, Melbourne, Victoria, Australia, 2015. Assoc. for Computing Machinery. doi: 10.1145/2723372.2737784. URL http://doi.acm.org/10.1145/2723372.2737784.

[2] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, and S. Seshadril. Update propagation protocols for replicated databases. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 97–108, Philadelphia, PA, USA, 1999. URL http://doi.acm.org/10.1145/304182.304191.

[3] Sebastian Burckhardt and Daan Leijen. Semantics of concurrent revisions. In *Euro. Symp. on Programming (ESOP)*, volume 6602 of *Lecture Notes in Comp. Sc.*, pages 116–135, Saarbrücken, Germany, March-April 2011. doi: http://dx.doi.org/10.1007/978-3-642-19718-5_7.

[4] Per Cederqvist et al. Version management with CVS, date unknown. URL http://ximbiot.com/cvs/manual/stable.

[5] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California Santa Cruz, Santa Cruz, CA, USA, December 1992. URL ftp://ftp.cse.ucsc.edu/pub/tr/ucsc-crl-92-52.ps.Z. Tech. Report no. UCSC-CRL-92-52.

[6] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm Strong Enough: Reasoning about consistency choices in distributed systems. In *Symp. on Principles of Prog. Lang. (POPL)*, pages 371–384, St. Petersburg, FL, USA, 2016. doi: 10.1145/2837614.2837625. URL http://dx.doi.org/10.1145/2837614.2837625.

[7] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 173–182, Montréal, Canada, June 1996. ACM SIGMOD, ACM Press. doi: http://doi.acm.org/10.1145/233269.233330.

[8] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS)*, 10(5):3–25, February 1992. URL http://www.acm.org/pubs/contents/journals/tocs/1992-10.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. URL http://doi.acm.org/10.1145/359545.359563.

[10] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)*, pages 401–416, Cascais, Portugal, October 2011. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/2043556.2043593.

[11] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Proving correctness of transformation functions in collaborative editing systems. Rapport de recherche RR-5795, LORIA – INRIA Lorraine, December 2005. URL http://hal.inria.fr/inria-00071213/.

[12] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Symp. on Op. Sys. Principles (SOSP)*, pages 288–301, Saint Malo, October 1997. ACM SIGOPS. URL http://doi.acm.org/10.1145/268998.266711.

[13] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Int. Conf. on Coop. Info. Sys. (CoopIS)*, volume 2888 of *Lecture Notes in Comp. Sc.*, pages 38–55, Catania, Sicily, Italy, November 2003. Springer-Verlag. URL http://www.springerlink.com/content/xygj6u96h1kgew05/.

[14] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Leţia. A commutative replicated data type for cooperative editing. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 395–403, Montréal, Canada, June 2009. doi: 10.1109/ICDCS.2009.20. URL http://doi.ieeecomputersociety.org/10.1109/ICDCS.2009.20.

[15] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated Abstract Data Types: Building blocks for collaborative applications. *Journal of Parallel and Dist. Comp.*, 71(3):354–368, March 2011. doi: http://dx.doi.org/10.1016/j.jpdc.2010.12.006.

[16] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, March 2005. doi: 1057977.1057980. URL http://doi.acm.org/10.1145/1057977.1057980.

[17] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag. doi: 10.1007/978-3-642-24550-3_29. URL http://www.springerlink.com/content/3rg39l2287330370/.

[18] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385–400, Cascais, Portugal, October 2011. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/2043556.2043592.

[19] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Int. Conf. on Computer-Supported Coop. Work (CSCW)*, page 59, Seattle WA, USA, November 1998. doi: http://doi.acm.org/10.1145/289444.289469.

[20] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149, Austin, Texas, USA, September 1994.

[21] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Symp. on Op. Sys. Principles (SOSP)*, pages 172–182, Copper Mountain, CO, USA, December 1995. ACM SIGOPS, ACM Press. URL http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf.

[22] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 233–242, Vancouver, BC, Canada, August 1984. URL http://doi.acm.org/10.1145/800222.806750.

[23] Haifeng Yu and Amin Vahdat. Combining generality and practicality in a Conit-based continuous consistency model for wide-area replication. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, Phoenix Mesa, Arizona, USA, April 2001. URL http://www.cs.duke.edu/~yhf/icdcsfinal.ps.