# Improving the scalability of geo-replication with reservations*

Mahsa Najafzadeh, Marc Shapiro (INRIA & UPMC-LIP6, Paris, France),
Valter Balegas, Nuno Preguiça (U. Nova de Lisboa, Lisbon, Portugal)

## Abstract

Geo-replicated systems improve performance and fault tolerance by replicating data on sites in different physical locations. These systems often eschew guaranteeing strong consistency because of performance loss and scalability and instead choose eventually consistency. Although eventual consistency improves performance especially in large scale but it might violate system invariants. In this work, we exploit reservation techniques to strengthen eventual consistency, by adding safety guarantees. We define a consistency model called RPB that takes the advantages of eventual consistency while providing stronger guarantees, including causality and safety properties.

## 1 Introduction

In cloud computing systems, geo-replication copies data in multiple data centers, in order to bring data close to the client. This aims to improve performance, by avoiding slow long-haul communication, and to improve availability and fault tolerance thanks to redundant data. However, in the presence of failures, system designers must choose to maintain either availability (and performance) or consistency – both are not possible together [1].

The recent proposal by Li et al. [2] enables both approaches to co-exist, by classifying operations as *red* and *blue*. Blue operations commute with all others; they are fast and available even when disconnected; they ensure causal eventual consistency [3]. For instance, in a bank account application, credit operations are blue, i.e., the user can add to his account in all circumstances.

Red operations must be mutually ordered, requiring system-wide synchronisation; they ensure strong consistency. In the banking application, debits are red, because the system needs to stop a debit that would make the balance negative. If the client cannot connect with the bank server, all debits are blocked.

The system is available even when the network partitions as long as the application invokes only blue operations [4]. If the workload is dominated by blue operations, performance is improved. However, the red-blue classification is conservative. An operation that might not commute, even rarely, is classified as red. For instance, in the bank example, debits are always red, even when the account has a high balance and the amount of the debit is small.

In this work, we identify another class of operations, called *purple*, that commute in well-defined states, and propose a *reservation* mechanism to identify and leverage such states. Returning to the example, a particular bank branch could *reserve* a portion of the account's balance, say 1 000 € out of a balance of 10 000 €, for a particular amount of time. This gives the branch the *capability* to make any number of debits, up to 1 000 € until the end of the day, without communicating. Of course, a (batched) summary of the debits must be sent to the bank's main servers before the reservation expires.

Since a blue operation never conflicts, it can execute at a replica without remote synchronisation. The operation propagates asynchronously, to ensure it executes durably at every replica. In contrast, a red operation requires a strong synchronisation protocol such as two-phase commit, where the first phase checks for conflicts, and the second phase makes it durable.

De-coupling these different concerns, the purple protocol has three phases. *Acquiring* a reservation ensures that future purple operations will not conflict. *Local purple execution* with appropriate reservation occurs without any synchronisation. *Releasing* a reservation, either explicitly or by it timing out, ends the reservation guarantees. Purple execution is as fast and available as blue operations, while reservations ensure guarantees as strong as red operations.

A possibly-conflicting operation lacking proper reservation cannot be handled as purple; it must be treated as red.

*This research is supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208).

Reservation is similar to a kind of a lock or escrow [5], generalised to non-numeric data types [6]. This paper further extends the reservation technique to geo-replicated clouds.

The rest of this paper is structured as follows. In Section 2, we define our Red-Purple-Blue (RPB) model, motivating with use cases. Section 3 describes the RPB protocol. We discuss fault tolerance in Section 4. Section 5 presents some concluding results and future work.

# 2 Red-Purple-Blue Consistency

## 2.1 System model

We assume a multi-tier system model, consisting of clients, application servers, and data servers. Data servers, located in data centres, are entrusted with storing data and applying updates durably; we assume that every data centre replicates all the data. Both kinds of servers are assumed reliable, i.e., they may crash but recover with their durable state intact. A client is located at the edge of the network and can access the system only through an application server.

Application logic executes on application servers, called *scouts*, which initiate red, purple and blue operations. We assume that a scout caches recently-accessed objects and stores reservations. Physically, a scout can be located where most convenient: in a data centre, in a network point-of-presence (PoP), or even in a trusted virtual machine in the client's computer. A scout located close to a client will improve responsiveness and availability (if it has enough cached objects and reservations); one close to a data server will get better consistency and can be shared among multiple clients. However, discussion of the trade-offs is out of our scope; for illustration purposes, we will assume that a scout is located at a PoP, in the cloud infrastructure, to preferentially serve nearby clients.

## 2.2 Red, purple and blue operations

Consider operations on some shared database.[1] Recall that an object is replicated at all data servers, and at a subset of the scouts (cached). Operations are invoked at some scout.

As mentioned, RPB supports three types of update operations: red, purple and blue. All operations are causally ordered [3]; additionally, conflicting red operations are totally ordered with one another [2]. A blue operation executes at some replica without remote synchronisation, and propagates to other replicas asynchronously. A red operation is strongly consistent, i.e., conflicting red operations execute in the same mutual order at all replicas; this incurs a significant synchronisation cost, which increases with contention. Blue operations are always available; red operations are not available when the network partitions.

A purple operation requires an appropriate reservation. The reservation ensures that the purple operation will not conflict with concurrent operations, thus ensuring strong consistency guarantees, just like red operations. However, as reservation is secured in advance, a purple operation can execute without synchronisation and propagate asynchronously, just like a blue operation. A purple operation with appropriate reservation is available (assuming the data it needs is in cache) despite network partitions and data server crashes.

## 2.3 Commutativity

Two operations are said to commute unconditionally with one another if executing them in any mutual order yields the same result, whatever the state. For instance, increments to a shared counter $x$ commute with one another: concurrent operations $inc(x, 1)$ and $inc(x, 2)$ can safely be ordered $inc(x, 1); inc(x, 2)$, at one server and $inc(x, 2); inc(x, 1)$ at another, since in both cases the end result is to increment $x$ by 3.

An operation is said *unconditionally commutative* if it commutes unconditionally with *all* other operations. An unconditionally-commutative operation can be classified as blue.

For instance, assuming a counter supporting increment, decrement, and read-value operations, the $inc$ and $readval$ operations commute unconditionally. More generally, all the operations of a Conflict-free Replicated Data Type (CRDT) are unconditionally commutative [7]. CRDTs include many useful data types, such as counters, sets, graphs, and sequences [8].

An operation may be *conditionally commutative*, i.e., it commutes with all other operations, but only in certain states. For instance, consider a non-negative shared counter $y$: operations $dec(y, 4)$ and $dec(y, 5)$ commute if $y \geq 9$, but otherwise not, since at least one of the decrements fails. In the red-blue model [2], a conditionally commutative operation is always red; in our RPB model, it can be purple.

[1] Hereafter we consider only updating operations. An operation may refer to a single update, or to a transaction containing several reads and updates.

## 2.4 Reservation mechanism

A *reservation* promises, to the scout that holds it, that the system is in a state that allows the scout to run the corresponding conditionally commutative operations as purple. A reservation is valid only for a certain duration (it is a *lease* [9]).

A reservation is a contract, between the data servers and the scout, that guarantees that concurrent applications (invoked by other scouts) will not break the promise provided by the reservation. The scout may invoke purple operations locally, as long as it does not exceed the capabilities of its reservation. If transmitted and received by data servers in time, the operation(s) will become durable. However, if the scout does not hold sufficient reservation, or it times out, the same operations run as red. We describe the protocol and how it tolerates faults later in this paper.

Reservation is a kind of lock but there are differences between the traditional locks in database and the reservation. While locking a resource in database is completely transparent, the scouts have a full control over the reservation. The scouts request reservations and determine their terms. In addition, a database lock is held until the end of a transaction whereas a reservation can span multiple transactions in a scout.

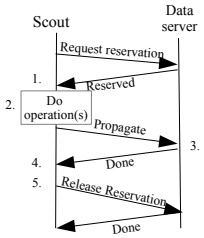


Figure 1: Reservation/purple protocol

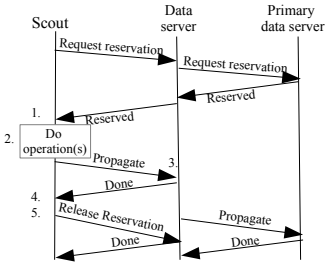


Figure 2: Server perspective of reservation-purple protocol

## 2.5 Use Cases

In this section, we take a look at some scenarios in which purple operation is natural or beneficial.

### 2.5.1 Online shopping

Online shopping offers different operations, including browsing, searching, adding products to a shopping cart, placing and canceling an order, in which different levels of consistency are possible. For example, canceling an order is blue, whereas buying an item needs strong consistency, to avoid selling an item that is not in stock. A branch store or bulk-buying agency may reserve or put an option on a number of items. Then, buying is purple.

### 2.5.2 Bank application

Bank operations such as deposit or add-interest can execute independently, because they are unconditionally commutative [2]. However, withdrawal is red, otherwise, concurrent clients could make large withdrawals, leading to negative balance. However, if a branch or a user reserves a portion of the current balance in advance, the verification can be done transparently without coordination (purple).

### 2.5.3 Collaborative tools

Conflict-free Replicated Data Types (CRDT) [10] avoid conflicts when users independently modify the same document in a collaborative application. Thanks to CRDTs, concurrent operations such as editing a document [10], or creating and removing files, can be made unconditionally commutative (blue operations).

However, some operations are not. For instance, in a file system, atomically moving a directory must be serialisable; otherwise, cycles might occur if the destination path is concurrently modified. Reserving a directory path gives a user the capability to independently perform moves within the reserved subtree (purple).

This is especially useful for mobile computing. If a user plans to travel and work disconnected, she may reserve a subtree, to be able to perform any operation, including moves within the reserved subtree.

# 3 Basic purple protocol

The basic protocol gives a scout a time-limited permission to invoke, run, and propagate corresponding purple operations, without waiting for any other server. In the next section, we will discuss fault tolerance. The full protocol is as follows, illustrated in Figure 1:

1. A scout requests and receives a reservation from the data servers.
2. The scout invokes one or more operations, and updates its local cache accordingly. (This step may occur either before or after Step 1.)
3. The scout propagates its operations to (a sufficient number of) data servers, which forward the operation to all replicas (i.e., to one another, and possibly to the scouts that cache the updated objects.) Upon receiving, the replicas also execute the operations.
4. The scout receives an acknowledgment from the data servers.
5. The scout sends any remaining reservation back to the data servers.

Note that in many cases, the protocol can be simplified. If the scout already has sufficient reservation, it may skip Step 1. If the operation exhausts the reservation, or just waits for the reservation to time out, Step 5 can be omitted. If the operations are unconditionally commutative, then both Steps 1 and 5 can be omitted: this is the blue protocol. Acquiring (Step 1), invoking, propagating (Step 3), and releasing (Step 5) can be compressed into a single two-phase commit protocol: this is the red protocol.

Although acknowledgement is required (Step 4), it is not necessary to wait for it: the scout can perform other reservations or operations immediately.

To illustrate, consider a travel agency that sells tickets for an event. For example, it might start by putting an option (a reservation in the RPB protocol) on a number of tickets. Once this is done, it can sell tickets to its clients without fear that the seats will be unavailable. The acknowledgment in Step 4 ensures the sale is durable, and visible to all replicas, but the agency can sell more tickets without waiting for the ack. At the end of the lease, the data servers automatically cancel any unconsumed reservation. This approach increases the autonomy and availability of the travel agency, and decreases the load on the data servers.

So far we focused on the scout's perspective. Consider now the protocol from the perspective of the data servers. One option is to use a primary-backup approach. A primary data server is assigned to each data object, which is responsible to handle the reservation requests on the object. A non-primary data server may forward requests from its scouts to the primary, but can also cache reservations locally to respond more quickly, as illustrated in Figure 2.

To avoid having a single point of failure, an alternative is to use a consensus protocol, such as Paxos [11], among data servers.

When a scout propagates its operations to a data server (Step 3), the latter propagates it further to other data centers and scouts.

# 4 Fault tolerance

The red operations abort under the synchronous protocols such as two-phase commit when the network partitions, whereas the purple operations proceed their execution during failures using the proper reservations and delay their confirmation. Reservation allows to separate two consistency requirements for strong operations: conflict avoidance and durability. The scouts relax conflict checking when running purple operations, as they only need to confirm their reservations with the data server at some later time before the end of lease time. Thus, reservation minimizes the overhead when retrying red operations that block or abort due to failures.

Reservation enables our Red-Purple-Blue protocol to recover from various failures. Because a communication or scout failure can lock a data object forever. We avoid such blocking situations by including a deadline per reservation. For example, if a scout crashes, the data server preserves all its reservations until they expire and after that, any state kept for those reservations at the server is garbage collected.

Each data server records the detail of granted reservations on stable logs and hence honors all reservations despite failures. When a non-primary data server fails, the replacement server in the data center resumes propagation for the confirmed reservations that have not yet been fully propagated. A primary data server failure can be handled by using a Paxos-based reconfiguration service. The reconfiguration service selects an another data server for the data objects that their primary server is down. When a failed primary data server recovers, it must synchronize with its replacement data server to take over its responsibility.

Communication failures are recovered by sending the propagation messages with the same operation ID repeatedly until an acknowledge received. If a data server has already acked an operation and received the same

operation ID again, the data server sends the acknowledgment message again, assuming the previous ack was lost. If it does not receive an acknowledgment by timeout, the purple operation reverts to red. In such case, the red operation can later fail. Thus, it is important to understand that the result of a purple operation must be considered tentative, although with a very high probability of becoming definitive. When using a fault-tolerant protocol in the servers, such as Paxos, a purple operation typically only fails if the client becomes disconnected. A client can decrease the likelihood of such event by using only reservations with validity longer than the typical failure period.

# 5 Conclusion

In this paper, we introduce a consistency model(RPB) allowing to support both weakly consistent and strongly consistency operations. Our consistency model optimizes committing operations that need stronger guarantees than eventual consistency. The optimization is based on holding reservations that allows operations run asynchronously, while taking into account their consistency requirements. Reservation provides some permissions over data that guarantee consistency.

In our ongoing studies, we plan to implement our RPB model in geo-replicated clouds and compare it with other state of the art solutions. There are two performance criteria of interest. The first criterion is the average number of round-trip messages is needed to commit a strong operation, while the second one is the percentage of operations that can commit locally.

We are also investigating to integrate service level agreements with our reservation mechanism to satisfy the quality of services. In addition, we try to reduce the overhead of reservation control messages using forecasting techniques and provide a framework to define the reservation semantics.

# References


[1] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[2] C. Li, D. Porto, A. Clement *et al.*, "Making geo-replicated systems fast as possible, consistent when necessary," in *Symp. on Op. Sys. Design and Implementation (OSDI)*, Hollywood, CA, USA, Oct. 2012, pp. 265–278.

[3] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, availability, and convergence," Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA, Tech. Rep. UTCS TR-11-22, 2011.

[4] E. Brewer, "CAP twelve years later: How the "rules" have changed," *IEEE Computer*, vol. 45, no. 2, pp. 23–29, Feb. 2012.

[5] P. E. O'Neil, "The escrow transactional method," *Trans. on Database Systems*, vol. 11, no. 4, pp. 405–430, Dec. 1986.

[6] N. Preguiça, J. L. Martins, M. Cunha *et al.*, "Reservations for conflict avoidance in a mobile database system," in *Int. Conf. on Mobile Sys., Apps. and Services (MobiSys)*, San Francisco, CA, USA, May 2003, pp. 43–56.

[7] M. Shapiro, N. Preguiça, C. Baquero *et al.*, "Conflict-free replicated data types," in *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, X. Défago, F. Petit, and V. Villain, Eds., vol. 6976. Grenoble, France: Springer-Verlag, Oct. 2011, pp. 386–400.

[8] ——, "Convergent and commutative replicated data types," *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, no. 104, pp. 67–88, Jun. 2011.

[9] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *Symp. on Op. Sys. Principles (SOSP)*. Litchfield Park AZ USA: ACM, Dec. 1989, pp. 202–210.

[10] N. Preguiça, J. M. Marquès, M. Shapiro *et al.*, "A commutative replicated data type for cooperative editing," in *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, Montréal, Canada, Jun. 2009, pp. 395–403.

[11] L. Lamport, "The part-time parliament," *Trans. on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.