G-DUR: A Middleware for Assembling, Analyzing, and Improving Transactional Protocols[*]

Masoud Saeida Ardekani
Inria &
Sorbonne Universités,
UPMC Univ Paris 06

Pierre Sutra
University of Neuchâtel

Marc Shapiro
Inria &
Sorbonne Universités,
UPMC Univ Paris 06

ABSTRACT

A large family of distributed transactional protocols have a common structure, called Deferred Update Replication (DUR). DUR provides dependability by replicating data, and performance by not re-executing transactions but only applying their updates. Protocols of the DUR family differ only in behaviors of few generic functions. Based on this insight, we offer a generic DUR middleware, called G-DUR, along with a library of finely-optimized plug-in implementations of the required behaviors. This paper presents the middleware, the plugins, and an extensive experimental evaluation in a geo-replicated environment. Our empirical study shows that:*(i)* G-DUR allows developers to implement various transactional protocols under 600 lines of code; *(ii)* It provides a fair, apples-to-apples comparison between transactional protocols; *(iii)* By replacing plugs-ins, developers can use G-DUR to understand bottlenecks in their protocols; *(iv)* This in turn enables the improvement of existing protocols; and *(v)* Given a protocol, G-DUR helps evaluate the cost of ensuring various degrees of dependability.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed databases*; H.2.4 [**Database Management**]: Systems—*Transaction processing*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*

General Terms

Algorithms, Performance

Keywords

Deferred Update Replication, Distributed Transaction, Distributed Data Store, Consistency Criterion

1. INTRODUCTION

Internet applications have conflicting requirements. On the one hand, they should be highly parallel and distributed in order to be fast, responsive and available; on the other, application servers must remain synchronized, in order to maintain consistency. Because of this tension, there exists a large number of distributed transaction models and protocols with different trade-offs: from very strongly synchronized ones [1], to ones with lots of parallelism [2–4].

Finding one's way in the jungle of consistency criteria and transactional protocols is not easy. Although literature is abundant, papers use different vocabulary, formalisms, and perspectives. Because they assume different environments, the implementations themselves are not comparable. It thus remains difficult to understand what are the important differences, and to make an objective, scientific comparison of their real-world behavior.

We propose a pragmatic tool for exploring the consistency design space. It is based on the insight that many protocols share a common structure, called deferred update replication (DUR), and differ only by the parametrization of a few generic functions [3–15]. For instance, they all have a read phase, differing by their choice of which specific object version to read; and a termination phase, differing only by how they detect and resolve concurrency conflicts.

We express this insight as a common algorithmic structure, with well-identified realization points. This generic structure is instantiated into a specific protocol by selecting appropriate plug-ins from a library. For instance, for a serializable protocol, the read plug-in will select the most recent committed version of an object, and the termination plug-in will abort any transaction if it is concurrent with an already-committed conflicting transaction.

We implement this structure as a generic middleware, called Generic DUR (G-DUR). G-DUR is a research tool that offers a library of highly-optimized plug-ins. In particular, it support the following customizations: *(i)* Optimistic read protocols differing by their versioning mechanisms and their freshness guarantees. *(ii)* Certification procedures differing by whether they can handle multiple transactions in parallel, and by how they manage conflicts. *(iii)* Commitment protocols based on group communication primitives (atomic broadcast or multicast), two-phase commit, or Paxos Commit.

[*]The research leading to this publication was partly funded by the European Commission's FP7 under grant agreement number 318809, LEADS project, and by the Agence Nationale de la Recherche (ANR) project ConcoRDanT (ANR-10-BLAN 0208).

Middleware '14 December 08 - 12 2014, Bordeaux, France

ACM 978-1-4503-2785-5/14/12$15.00.
http://dx.doi.org/10.1145/2663165.2663336

By mixing and matching the appropriate plug-ins, it is relatively easy to obtain a high-performance implementation of a protocol. We leverage this capability with an extensive experimental evaluation in a geo-replicated environment. The contributions of this paper include the following:

(1) We tailor G-DUR to implement six prominent transactional protocols [3, 4, 10–12, 15]. The implementation of each protocol in G-DUR requires only 200 to 600 lines of code (LOC), an order of magnitude less than the monolithic originals. We evaluate empirically these protocols. Our apples-to-apples comparison brings out differences between the protocols, and between the consistency criteria they implement. In addition, it shows that they have well-separated performance domains, and enables us to precisely identify their respective limitations.

(2) We show how a developer can use G-DUR to finely understand the limitations of a protocol. We take a recently published protocol [4], and identify its bottlenecks by methodically replacing its plugs-ins by weaker ones.

(3) The previous approach also helps a developer to enhance existing protocols. We illustrate this point by presenting a variation of P-Store [11] that leverages workload locality to perform up to 70% faster than the original protocol.

(4) We evaluate the cost of various degrees of dependability. To this end, we take a protocol ensuring serializability and we study the price of tolerating failures by varying the replication degree and the algorithm in use during commitment.

We organize the remainder of this paper as follows. Section 2 reviews related work. We give an overview of the transactional middleware at core of G-DUR in Section 3. Section 4 details the execution phase of G-DUR, and we explain its termination, and atomic commitment protocols in Section 5. In Section 6, we show how to realize various protocols in G-DUR. The implementation of G-DUR is covered in Section 7. In Section 8, we conduct our experimental evaluation, and we conclude in Section 9.

2. RELATED WORK

In general, application workloads exhibit a large portion of non-conflicting transactions. Under such an assumption, the interest of the DUR approach, i.e., an optimistic phase followed by a termination phase, was underlined by Alonso [16]. Wiesmann and Schiper [17] compare several replication protocols and confirm that DUR is better than distributed locking and primary copy under full replication. The work of Schmidt and Pedone [18] provides a formal analysis of DUR, focusing on serializability and full replication. The DUR approach is also a de facto standard for software transactional memories [19, 20].

Several works aim at understanding and classifying full replication techniques. Wiesmann et al. [21] provide a classification of different replication mechanisms, for both transactional and non-transactional systems. Subsequently, the authors propose a three parameter classification of transactional protocols [22]. They classify protocols according to the server architecture (primary copy versus update-everywhere), the server interaction (constant versus linear), and the transaction termination (voting versus non-voting). The present work continues their study, focusing on DUR protocols under *partial replication*. According to their terminology, this means that we shall be interested in passive replication with either update-everywhere or primary-copy, and both voting and linear interaction.

An abundant literature (see detailed survey elsewhere [23]) provides analytical models and simulations of distributed database systems. These works focus on evaluating throughput or latency as a function of workload, replication factor and network characteristics, and provide useful insights on how transactional protocols behave. However, by oversimplifying the management of conflicting transactions, they do not give a completely accurate figure. In particular, the impacts of *(i)* versioning mechanism, *(ii)* consistency criteria, *(iii)* convoy effects during certification, and *(iv)* genuineness [1] on a protocol are largely under-evaluated. Our experiments show that these parameters strongly influence performance.

Commercial database products [24, 25] usually allow the client to pick a consistency criterion when executing transactions. Several researchers have studied, and compared multiple criteria. Berenson et al. [26] show that the phenomena-based definition of ANSI SQL-92 does not properly characterize the differences between SI and SER. They propose new anomalies and compare most well-known criteria based on this characterization. Adya et al. [27, 28] present the first implementation-independent specification of ANSI levels. This specification is not limited to pessimistic implementations, that is based on an *a priori* ordering of the transactions [29, 30], but is also applicable to optimistic and multi-version schemes. To achieve this, the authors capture conflicts with various graphs that they use to model and compare well-known consistency criteria. This specification is yet hard to understand, especially in the context of distributed transactional systems.

A few works study and compare different family of transactional protocols. Bernabé-Gisbert et al. [31] introduce a middleware that ensures different levels of consistency. They use an update everywhere approach, and certify transactions with the help of atomic broadcast. Each transaction can declare separately a consistency criterion of its choice. It is then the job of the underlying database to certify transactions, and to execute the concurrency control. No comparison or evaluation is given in this paper. Kemme and Alonso [6] introduce a family of eager replication protocols using group communication primitives. They compare various consistency criteria (SER, SI and Cursor Stability) with simulations, but do not provide a unified protocol.

3. OVERVIEW

The core goal of Deferred Update Replication (DUR) is to provide clients with a transactional datastore abstraction. Under the hood, this store is distributed and replicated across multiple replicas. Replicas synchronize each other to offer clients a consistent and live access to the datastore.

G-DUR is designed as a generic, tailorable, implementation of DUR. Figure 1 presents the global architecture of the middleware. Clients submit their transactions to G-DUR instances. A client may execute a transaction interactively, i.e., G-DUR does not require all the transactional code to be submitted at once. Certain optimizations are nevertheless possible if such an assumption holds. A transaction starts by a *begin* operation, followed by one or more CRUD operations (i.e., Create, Read, Update, or Delete) , and ends with *commit* or *abort*. Create, update, and delete operations are

[1] A protocol is genuine when only the replicas of the objects accessed by a transaction make computational steps to execute it.

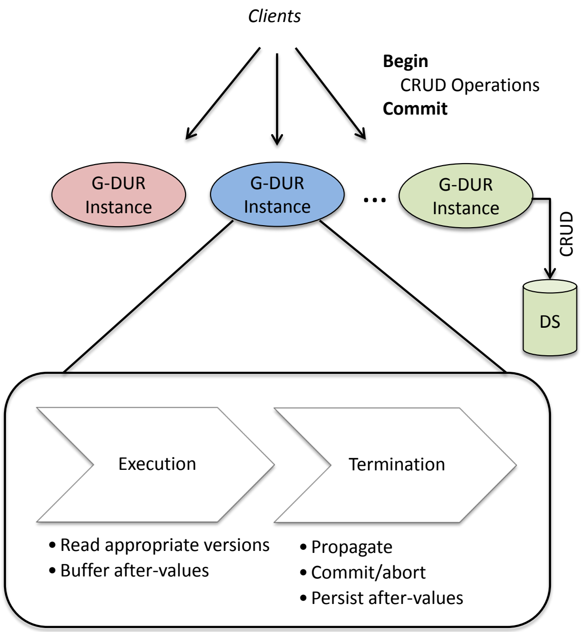


Figure 1: G-DUR Architecture

Notation	Meaning
x, y	object
T_i, $i \in \mathbb{N}$	transaction
x_i	version of x written by T_i
$o_i(x)$	T_i reads or writes object x
$\mathit{coord}(T_i)$	coordinator of T_i
$\mathit{rs}(T_i)$, $\mathit{ws}(T_i)$	read and write set of T_i
$T_i \parallel T_j$	T_i and T_j are concurrent
Π	set of all replicas

Table 1: Notations

implemented as write operations. Consequently, we shall be simply referring to read and write operations in the following.

Each G-DUR instance *coordinates* the transactional requests it receives from a client. To that end, an instance holds a local datastore containing a subset of the globally available data, and it executes two customizable *execution* and *termination* protocols (see bottom of Figure 1). The execution protocol is responsible for reading data and buffering after-values. The termination protocol handles the propagation of the transaction side effects, its commitment and the persistence of after-values.

Refer to Table 1 for a summary of the notations used in the remainder of this paper.

At some G-DUR instance, a transaction T_i, can be in four distinct states: *Executing*, *Submitted*, *Committed* or *Aborted*, explained next:

- *Executing*: Each operation $o_i(x)$ in T_i is executed speculatively at the coordinator, i.e., at the replica that receives the transaction from the client. If $o_i(x)$ is a read, the coordinator returns the corresponding value, fetched either from the local replica or a remote one. If $o_i(x)$ is a write, the coordinator stores the corresponding update value in a local buffer, enabling *(i)* subsequent reads to observe the modified value, and *(ii)* the subsequent commit to send the after-values to remote replicas.

Algorithm 1 Execution protocol - code at process p

```
Variables:
    $ds$, $committed$, $aborted$, $executing$, $submitted$

$execute(\textsc{begin}, T_i)$
    pre: $T_i \notin executing \cup aborted \cup committed \cup submitted$
    eff: $executing \leftarrow executing \cup \{T_i\}$
         $\underline{init}(T_i)$
$execute(\textsc{read}, T_i, x)$
    pre: $T_i \in executing$
    eff: if $\exists x_i \in ws(T_i)$ then return $x_i$
         else if $isLocal(x)$ then return $localRead(x)$
         else
             $send\ \langle \textsc{req}, T_i, x\rangle\ to\ \ q \in replicas(x)$
             wait until $received\ \langle \textsc{reply}, T_i, x_j\rangle\ from\ q$
             return $x_j$
$execute(\textsc{write}, T_i, x_i)$
    pre: $T_i \in executing$
    eff: $ws(T_i) \leftarrow ws(T_i) \cup \{x_i\}$
$execute(\textsc{commit}, T_i)$
    pre: $T_i \in executing$
    eff: $submit(T_i)$
$localRead(T_i, x)$
    pre: $\underline{choose}(x,\ T_i) \neq \varnothing$
    eff: let $x_j \in \underline{choose}(x,\ T_i)$
         return $x_j$
$remoteRead(T_i, x, q)$
    pre: $received\ \langle \textsc{req}, T_i, x\rangle\ from\ q$
         $\underline{choose}(x,\ T_i) \neq \varnothing$
    eff: let $x_j \in \underline{choose}(x,\ T_i)$
         $send\ \langle \textsc{reply}, T_i, x_j\rangle\ \ to\ q$
```

- *Submitted*: Once all the read and write operations of T_i have executed, the coordinator submits it for termination. This includes synchronizing with the concerned replicas, and a *certification* check to satisfy the safety conditions of the implemented consistency criterion.
- *Committed/Aborted*: If certification is successful, T_i enters the *Committed* state, and every process $q \in \mathit{replicas}(T_i)$ applies the transaction's after-values (if any) to its copy of the datastore. Otherwise, T_i aborts, and enters the *Aborted* state. Consequently, its after-values are discarded.

Building upon the work of Wiesmann et al. [21], our key insight in the design of G-DUR is that *all* the DUR protocols satisfy the above description. In the next sections, we give additional detail on our generic execution and termination protocols. These protocols are explained as a set of atomic actions guarded by pre-conditions. The customizable points (called *realization points*) appear as functions whose names are underlined and in blue in the algorithms, e.g., $\underline{choose}()$. Concretely, a *realized* protocol will define the set of plug-ins to be called in lieu of the realization points.

4. EXECUTION

Algorithm 1 shows the pseudo-code of the execution protocol from the perspective of a G-DUR instance (replica p). The description refers to the following variables: We note *ds* the local (partial) copy of the datastore. Variable *committed*, *aborted*, *executing* and *submitted* refer to four sets that serve to log the transactions the replica executes,

A transaction T_i starts when action $\mathit{execute}(\textsc{begin}, T_i)$ is invoked at process p. In such a case, we say that p is the coordinator of T_i, denoted $\mathit{coord}(T_i)$. Action $\mathit{execute}(\textsc{read}, T_i, x)$ describes how T_i reads some object x. First, $\mathit{coord}(T_i)$ checks against the buffer $ws(T_i)$ in case T_i previously updated x. Otherwise, if the local database contains a copy of x, $coord(T_i)$ reads it (line 11). If none of the previous cases hold, $coord(T_i)$ sends an (asynchronous) read request to some replica holding x (line 13). Such a request is re-iterated to another replica, in case no answer is returned after some time (not covered in Algorithm 1).

Local (i.e., the coordinator is p) and remote reads (when the coordinator has requested a read from p) are handled by the actions *localRead* and *remoteRead* respectively. In both cases, the plug-in for *choose* selects a version that complies with the consistency criterion's versioning rules. We shall detail shortly how.

Action $execute(\text{WRITE}, T_i, x)$ describes the processing of a write request by T_i at the coordinator $coord(T_i)$. The middleware buffers the update value in $ws(T_i)$ (line 18).

When the execution reaches the end of the transaction, action $execute(\text{COMMIT}, T_i)$ submits T_i to the termination protocol line 21). The execution algorithm then waits until T_i either commits or aborts, and returns the outcome.

When the termination protocol commits a transaction, it stores the modifications in the datastore. Depending on the realized protocol, one or more versions of an object may exist simultaneously in ds. The realization point *choose* (line 23 in Algorithm 1) abstracts which version is selected when the replica resolves a read request. G-DUR provides a convenient and generic support for tracking and choosing versions. We detail it in the next sections.

4.1 Version Tracking

The choice of a version depends on the *versioning mechanism* at work in the datastore. Recall that when a transaction T_i writes to object x, we say that it creates *version* x_i. Given some history h, we abstract a versioning mechanism Θ as a mapping that associates to each version $x_i \in h$, a *version number* $\Theta(x_i)$ taken from some partially ordered set $(\mathcal{V}, <)$.

Several versioning mechanisms have been proposed in the past. Their implementations usually rely on timestamps (TS) [32], vector clocks (VC) [33] or version vectors (VV) [34, 35]. More recently, Sovran et al. [3] and Sciascia and Pedone [12, Section E] discovered independently the concept of vector timestamps (VTS), which support the computation of partially consistent snapshots, at the cost of communicating with all replicas in the background. The GMU vectors (GMV) of Peluso et al. [4] do not have this drawback, but they do not guarantee the monotonicity of snapshots. Partitioned dependence vectors (PDV) offer a mechanism close to GMV. In case their size equals $O(m)$, with $m = |Objects|$, they allow a permissive computation of all the partial consistent snapshots [15, Theorem 1].[2] In the current state of the implementation, G-DUR supports the TS, VC, VTS, GMV and PDV versioning mechanisms.

Workload contention, liveness of read-only transactions and storage cost all influence the choice of a versioning mechanism, For instance, a DUR protocol may favor a central sequencer assigning timestamps for its simplicity despite that it creates a potential bottleneck in the system. The dimension of $\mathcal{V}$ usually varies from one to the size of the data set or the number of storage nodes. Recently, Peluso et al. [37] prove an $\Omega(min(m, n))$ lower bound on the dimension of $\mathcal{V}$ with $n = |\Pi|$ when the datastore is strictly disjoint access parallel.[3] A similar result was previously conjectured in [39].

Algorithm 2 Termination protocol - code at process p

```
Variables:
    $ds$, $committed$, $aborted$, $executing$, $submitted$, $\mathcal{Q}$, $\mathcal{AC}$

$initialize()$
    eff:  if $\mathcal{AC} = \text{GC}$ then start Algorithm 3
          else if $\mathcal{AC} = \text{2PC}$ then start Algorithm 4
$submit(T_i)$
    pre:  $T_i \in executing$
    eff:  $executing \leftarrow executing \setminus \{T_i\}$
          $submitted \leftarrow submitted \cup \{T_i\}$
          $obj \leftarrow \underline{certifying\_obj}(T_i)$
          if $obj = \varnothing$ then
              $committed \leftarrow committed \cup \{T_i\}$
          else
              $\underline{xcast}(\text{TERM}, T_i)$ to $replicas(obj)$
$xdeliver(T_i)$
    pre:  $received\ \langle \text{TERM}, T_i \rangle$
    eff:  $\mathcal{Q} \leftarrow \mathcal{Q} \circ T_i$
$commit(T_i)$
    pre:  $decide(T_i) = \text{COMMIT}$
    eff:  $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{T_i\}$
          $committed \leftarrow committed \cup \{T_i\}$
          $ds \leftarrow ds \cup \{x_i \in ws(T_i) : x_0 \in ds\}$
          $\underline{post\_commit}(T_i)$
$abort(T_i)$
    pre:  $decide(T_i) = \text{ABORT}$
    eff:  $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{T_i\}$
          $aborted \leftarrow aborted \cup \{T_i\}$
          $\underline{post\_abort}(T_i)$
```

4.2 Picking a Version

The realization of function *choose* by a DUR protocol fits in two categories: $choose_{last}$ returns the latest version of the object in the sense of $<$, whereas $choose_{cons}$ returns a version consistent with the previous reads. The first mechanism is straightforward but requires to abort queries that did not read a consistent snapshot. In the second case, G-DUR abstracts the behavior of protocols that construct consistent snapshots on the course of the execution with a *versions compatibility test*. This test takes as input two version numbers $\Theta(x_i)$ and $\Theta(y_j)$, and it outputs *true* iff $\{x_i, y_j\}$ forms a consistent snapshot according to the versioning mechanism Θ. Upon executing a read request from transaction T_i on some object x, $choose_{cons}$ returns the latest version of x that is compatible with all the versions read previously by T_i.

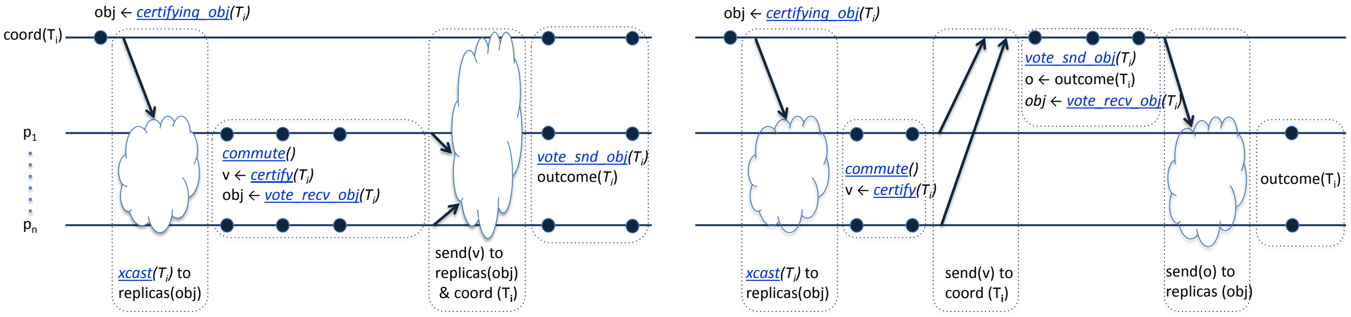
5. TERMINATION

Algorithm 2 depicts the pseudo-code of the termination protocol in G-DUR It accesses the same variables as the execution protocol along with a FIFO queue named $\mathcal{Q}$ and a variable called $\mathcal{AC}$. This last variable specifies a particular atomic commitment algorithm; we shall detail its role shortly.

When a transaction T_i is submitted for termination, the coordinator first computes $certifying\_obj(T_i)$, the set of objects required to *certify* T_i. Depending on the realized protocol,

[2] Following Guerraoui et al. [36], a transactional system is permissive to some property P iff for every history in $h \in P$, there exists a run that simulates h.

[3] A datastore is strictly disjoint access parallel when two non-conflicting transactions never contend on the same base object (i.e., on any object in use at the implementation level) [38]. This definition generalizes the concept of genuine partial replication Schiper et al. [11].



(a) Using group communication

(b) Using two-phase commit

Figure 2: Atomic Commitment Timeline

certifying_obj(T_i) returns one of the following sets of objects:

- $\varnothing$: When returning an empty set, the realized protocol allows transaction T_i to commit without synchronization. A typical use case is to ensure that read-only transactions are wait-free.
- $ws(T_i)$: In this case, only the objects modified by transaction T_i are certified.
- $ws(T_i) \cup rs(T_i)$: Both the readset and the writeset of T_i are involved in the certification. Protocols ensuring serializability (or above) usually return this value.
- *Objects*: This last case represents a scenario where all replicas should participate in the certification.

In the case where *certifying_obj*(T_i) returns an empty set, T_i commits locally (line 12). Otherwise, T_i is sent to all the replicas *concerned* by T_i, that is holding some object in *certifying_obj*(T_i). This is up to the realized protocol to chose an appropriate *xcast* primitive in order to propagate the submitted transaction (line 15). Some protocols [8, 10] employ atomic broadcast, while others [11, 15] use atomic or reliable multicast. (The reader may refer to Défago et al. [40] for complete specifications and explanations of these communication primitives.)

Upon delivery of T_i for termination (action *xdeliver*(T_i)), T_i is added to queue $\mathcal{Q}$. The atomic commitment algorithm (variable $\mathcal{AC}$) then *decides* upon T_i, and eventually commit or abort it. Once the atomic commitment algorithm has taken a decision upon T_i, the transaction is flagged either COMMIT or ABORT. If it is COMMIT, the transaction is removed from $\mathcal{Q}$, added to *committed*, then its updates are applied to the local database. Otherwise, it is added to *aborted*. In both cases, and once a transaction is terminated, an event (*post_commit*() or *post_abort*()) is triggered through a function call. These events can be used to perform operations off the critical path (e.g., garbage collection).

From the above description, we notice that atomic commitment plays a central role in termination. This plug-in decides upon the outcome of transactions by certifying them. The most widely employed realizations of this plug-in are *(i)* atomic commitment with group communication ensuring a total or partial ordering of transactions [8, 10, 11], *(ii)* atomic commitment using two phase commit [3, 4], and *(iii)* atomic commitment with Paxos Commit [41]. In what follows, we cover in detail the first two realizations. The third one is omitted due to space limitations.

5.1 Group Communication

Figure 2-a presents an overview of atomic commitment with group communication (GC). Conceptually, this approach is divided into the following steps: Transaction T_i is first sent to a set of voting replicas which deliver it in the same order. These replicas certify T_i, then send the results of their certification tests to another group of replicas. This last group of replicas contains at least the coordinator and the replicas of $ws(T_i)$, but it might be larger in certain cases. After receiving enough votes, these processes decide locally upon the outcome of T_i.

Algorithm 3 details the internals of the approach. This realization requires that function *xcast* ensures a partial order property over the set of submitted conflicting transactions. Hence, depending on the realized protocol, *xcast* can be replaced with (uniform) atomic broadcast or multicast. When transaction T_i is added to $\mathcal{Q}$, a certification vote is cast upon it. This vote occurs when T_i commutes with all the transactions that precede it in $\mathcal{Q}$ (line 3). The definition of commutativity is a function of the consistency model implemented by the realized protocol. Commutativity is crucial to minimize convoy effect during certification [11, 42], that is when the certification of one transaction slows down the certification of another one.

Once a replica locally certifies T_i (line 4), it sends the result of its vote to the coordinator and to the processes in *replicas*(*vote_recv_obj*(T_i)). In most cases, *vote_recv_obj*(T_i) equals $ws(T_i)$, that is the objects updated by the transaction. However, for some protocols, all replicas must receive the certification votes.

A process can safely decide upon the outcome of T_i once it has received votes from a *voting quorum* for T_i. A voting quorum Q for T_i is a set of replicas such that for every object $x \in$ *vote_snd_obj*(T_i), the set Q contains at least one replica of x. Formally, $vquorum(T_i)$ equals $\{Q \subseteq \Pi \mid \forall x \in$ *vote_snd_obj*$(T_i) : \exists j \in Q \cap replicas(x)\}$. A process uses the following (three-values) predicate *outcome*(T_i) to determine whether some transaction T_i commits, or not:

```
$outcome(T_i) \equiv$
    **if** $vote\_snd\_obj(T_i) = \varnothing$
        **then** $true$
    **else if** $\forall Q \in vquorum(T_i), \exists q \in Q,$
        $\neg received\ \langle \text{VOTE}, T, \_ \rangle\ from\ q$ **then** $\bot$
    **else if** $\exists Q \in vquorum(T_i), \forall q \in Q,$
        $received\ \langle \text{VOTE}, T, true \rangle\ from\ q$ **then** $true$
    **else** $false$
```

Once the result of *outcome*(T_i) equals *true*, and T_i is at the beginning of $\mathcal{Q}$, T_i is flagged COMMIT (lines 9 to 11). Other-

Algorithm 3 Atomic Commitment with GC - code at p

```
1:  $vote(T_i)$
2:      pre:  $T_i \in \mathcal{Q}$
3:            $\forall T_j \leq_{\mathcal{Q}} T_i : commute(T_i, T_j)$
4:      eff:  $v \leftarrow certify(T_i)$
5:            $send$ $\langle \text{VOTE}, T_i, v \rangle$ to
6:                $replicas(vote\_recv\_obj(T_i)) \cup \{coord(T_i)\}$
7:  $decide(T_i)$
8:      pre:  $outcome(T_i) \neq \bot$
9:      eff:  if $outcome(T_i)$ then
10:               wait until $T_i = head(\mathcal{Q})$
11:               return COMMIT
12:           else
13:               return ABORT
```

Algorithm 4 Atomic Commitment with 2PC - code at p

```
1:  $vote(T_i)$
2:      pre:  $T_i \in \mathcal{Q}$
3:      eff:  if $\exists T_j \in \mathcal{Q} : \neg commute(T_i, T_j)$ then
4:                $send$ $\langle \text{VOTE}, T_i, false \rangle$ $to$ $coord(T_i)$
5:            else
6:                $v \leftarrow certify(T_i)$
7:                $send$ $\langle \text{VOTE}, T_i, v \rangle$ to $coord(T_i)$
8:  $vote\_coordinator(T_i)$
9:      pre:  $p = coord(T_i)$
10:     eff:  $send$ $\langle \text{VOTE}, T_i, outcome(T_i) \rangle$ to $vote\_recv\_obj(T_i)$
11: $decide(T_i)$
12:     pre:  $outcome(T_i) \neq \bot$
13:     eff:  if $outcome(T_i)$ then return COMMIT
14:           else return ABORT
```

wise, if $outcome(T_i)$ equals *false*, it is flagged ABORT (line 13). Algorithm 3 waits that T_i reaches the head of the queue before committing it to ensure replicas apply updates in the same order. This property is mandatory for every criterion equal or stronger than Serializability [18]. For weaker consistency criteria (e.g., Read Committed), we can suppress this constraint. However, in our experience, such a modification has a small impact on performance.

5.2 Two-Phase Commit

The timeline of termination with two-phase commit (2PC) for some transaction T_i is given at Figure 2-b. The core difference between 2PC and GC lies in the way 2PC employs the coordinator. In Algorithm 3, all processes receive the certification votes and locally decide to commit (or abort) the transaction. In 2PC, the coordinator receives all votes, decides the outcome of the transaction, then notifies other participants about its decision.

We provide a complete view of 2PC in Algorithm 4. This realization of atomic commitment overrides function *xcast* with a multicast primitive. When a replica delivers a transaction T_i, it aborts T_i in case a concurrent conflicting transaction precedes it in $\mathcal{Q}$. Otherwise, the transaction is certified. In both cases, the outcome is sent to $coord(T_i)$ which will decide upon the outcome of T_i; other replicas need only to receive the final vote from $coord(T_i)$. We reflect this by modifying the definition of a voting quorum. More precisely, $vquorum(T_i)$ equals $\{Q \subseteq \Pi \mid \forall x \in vote\_snd\_obj(T_i) : replicas(x) \subseteq Q\}$ at the coordinator, and $\{coord(T_i)\}$ at other replicas.

Algorithm 3 orders a priori conflicting transactions, whereas Algorithm 4 relies on the spontaneous ordering of the network. We shall see in Section 8.5 that this last choice increases abort rate under contention.

We pointed out previously that Algorithms 3 and 4 differ on their use of the coordinator. The next section details how this key difference impacts the fault-tolerance of the two approaches.

5.3 Fault-Tolerance

The approach based on two-phase commit works either when perfect failure detectors are available [43], or in a crash-recovery model. In the first case, the coordinator preemptively aborts the transaction when a replicas fails in the middle of the termination phase. In the later, *(i)* every time the state of Algorithm 4 changes, the modification must be logged, and *(ii)* when a replica crashes, Algorithm 4 has to wait that it comes back online to pursue the execution.

A commitment protocol based on group communication can cope more easily with failures if it internally relies on a dependable consensus protocol. In more details, if Algorithm 3 employs atomic broadcast to order transactions, it needs inaccurate failure detection and tolerates up to $f < n/2$ replica crashes, where n is the total number of replicas. Now, if only replicas concerned by the transaction make steps to commit it, Algorithm 3 should use a genuine atomic multicast primitive [44]. In the general case, this requires perfect failure detection [45]. Nevertheless, with an appropriate replicas placement, e.g., in a geo-replicated scenario, inaccurate failure detectors can implement a non disaster-tolerant atomic multicast (see Schiper et al. [46] for more detail).

The difference in terms of dependability between the two approaches translates into a difference in time and message complexity. Let us note r the average cardinality of $replicas(ws(T) \cup rs(T))$. Algorithm 4 requires $\Omega(r)$ messages and its message delay is 2. On the other hand, an optimal atomic broadcast protocol [47, 48] costs 3 message delay with $\Omega(n)$ messages, and the best genuine fault-tolerant atomic multicast known to date needs 6 message delays with $\Omega(r^2)$ messages [45]. This makes fault-tolerance expensive at first glance. In Section 8.5, we further investigates this cost in the context of geo-replicated data.

6. REALIZING PROTOCOLS

This section illustrates how to specify a protocol in the G-DUR middleware. To that end, we chose five consistency criteria: Serializability, Snapshot Isolation, Update Serializability, Parallel Snapshot Isolation and Non-Monotonic Snapshot Isolation. For each criterion, we pick at least one state-of-the-art DUR protocol, and explain how to implement it with G-DUR. As we shall see, we can express the core aspects of a protocol in less than 10 lines of pseudo-code.

In all the implementations we cover next, and otherwise specified, $vote\_recv\_obj(T_i)$ returns $ws(T_i)$, and the realization of $vote\_snd\_obj()$ is the same as the realization of $certifying\_obj()$.

6.1 Serializability

Serializability (SER) is the classical consistency criterion implemented by transactional systems. A transactional system satisfies SER when every concurrent execution of committed transactions is equivalent to some serial execution of the same transactions. While early replication protocols under SER have some scalability limitations, recent works overcome these problems by focusing on two properties: (*Genuine Partial Replication.*) While partial replication increases scalability, some protocols [9] still require to communicate

Algorithm 5 P-Store [11]

```
1: $\Theta \equiv TS$
2: $choose \equiv choose_{last}$
3: $\mathcal{AC} \equiv$ GC
4: $xcast \equiv$ AM-Cast
5: $certifying\_obj(T_i) \equiv ws(T_i) \cup rs(T_i)$
6: $commute(T_i, T_j) \equiv rs(T_i) \cap ws(T_j) = \varnothing \wedge rs(T_j) \cap ws(T_i) = \varnothing$
7: $certify(T_i) \equiv \forall (x_i, x_j) \in rs(T_i) \times db : \Theta(x_j) \leq \Theta(x_i)$
```

Algorithm 6 S-DUR [12]

```
1: $\Theta \equiv VTS$
2: $choose \equiv choose_{cons}$
3: $\mathcal{AC} \equiv$ GC
4: $xcast \equiv$ AM$_{pw}$-Cast
5: $certifying\_obj(T_i) \equiv$ if $|ws(T_i)| = 0$ then $\varnothing$
                               else $ws(T_i) \cup rs(T_i)$
6: $commute(T_i, T_j) \equiv rs(T_i) \cap ws(T_j) = \varnothing \wedge rs(T_j) \cap ws(T_i) = \varnothing$
7: $certify(T_i) \equiv \forall T_j \parallel T_i \in committed$ :
                 $ws(T_i) \cap rs(T_j) = \varnothing \wedge rs(T_i) \cap ws(T_j) = \varnothing$
8: $post\_commit(T_i) \equiv$ M-Cast($\Theta(T_i)$)
                       to $(\Pi \setminus replicas(certifying\_obj(T_i)))$
```

Algorithm 7 GMU [4]

```
1: $\Theta \equiv GMV$
2: $choose \equiv choose_{cons}$
3: $\mathcal{AC} \equiv$ 2PC
4: $certifying\_obj(T_i) \equiv$ if $|ws(T_i)| = 0$ then $\varnothing$
                               else $rs(T_i) \cup ws(T_i)$
5: $commute(T_i, T_j) \equiv rs(T_i) \cap ws(T_j) = \varnothing \wedge rs(T_j) \cap ws(T_i) = \varnothing$
6: $certify(T_i) \equiv \forall x_i, x_j \in rs(T_i) \times db : \Theta(x_j) \leq \Theta(x_i)$
```

with all replicas in the system. This decreases the parallelism in the system. On the contrary, P-Store [11] ensures genuine partial replication (GPR). This property states that a transaction only communicates with the replicas holding some object read or written by the transaction. (*Wait-Free Queries.*) In early serializable solutions, both queries (read-only) and update transactions need to certify and go through a synchronization phase. The S-DUR protocol of Sciascia and Pedone [12] solves this issue by ensuring that queries are wait-free (WFQ), that is they do not wait for concurrent transactions and always commit. In what follows, we explain how to realize both P-Store and S-DUR in our middleware.

P-Store: Algorithm 5 depicts our realization of P-Store. This protocol relies on a timestamping mechanism to version objects (line 1). Every read operation retrieves asynchronously the latest version of the corresponding object (line 2). A transaction commits iff no new versions of the objects it read were created concurrently (line 7). Such a certification test is typical of DUR protocols that ensures SER. Another classical approach [49] is to rely on cycles detection in the serializability graph. However, as pointed by Guerraoui et al. [36], this is expensive.

S-DUR: Unlike P-Store, S-DUR (Algorithm 6) ensures that every read operates on a consistent snapshot (line 2). This implies that queries are wait-free (line 5). Upon the termination of an update transaction T_i, S-DUR atomic multicasts T_i to the replicas holding an object in $ws(T_i) \cup rs(T_i)$. In that case, however, the group communication primitive only ensures a pair-wise ordering of the transactions [40], i.e., two processes only deliver transactions they have in common in the same order (line 4). This design tends to increase the scalability of the multicast primitive, but comes at the price of the following drawbacks: *(i)* More aborts because an update transaction commits only if there is no concurrent conflicting committed transaction (line 7), and *(ii)* Saeida Ardekani et al. [50] proved that no GPR system under SER can ensure WFQ. Thus, S-DUR needs to perform some background propagation to all replicas (line 8). This propagation consists in sending $\Theta(T_i) = max\ \{\Theta(x_i) : x_i \in rs(T_i) \cup ws(T_j)\}$ to advance the vector clock maintained at each process.

6.2 Update Serializability

Update serializability (US) was introduced by Garcia-Molina and Wiederhold [51], then later extended to aborted transactions by Hansdah and Patnaik [52]. US guarantees that update transactions are serialized, and that read-only transactions see consistent but non-monotonic snapshots. This last property weakens SER such that the impossibility of ensuring WFQ in a GPR system is circumvented.

GMU: The GMU transactional system of Peluso et al. [4] (see Algorithm 7) commits read-only transactions locally (line 4), and in the case of update transactions, it makes use of 2PC (line 3). All replicas holding an object read or written by the transaction participate to 2PC. We note here that the certification test of GMU (line 6) is the same as P-Store while this system ensures both GPR and WFQ. This comes from the fact that, as pointed out previously, US is more permissive to conflicting transactions than SER.

6.3 Snapshot Isolation

Snapshot isolation (SI) is defined by Berenson et al. [26], then later generalize by Elnikety et al. [53]. A transaction executing under SI observes the state of the database at some point in time. Thus, SI implements *de facto* WFQ. This feature, as well as the conceptual simplicity of SI, are appealing enough that most database vendors implement SI in their products (e.g., Microsoft SQL-Server, Oracle and PostgreSQL). However, it has been shown recently that it is impossible to ensure SI in a GPR system [50].

Serrano: The protocol of Serrano et al. [10] offers non-genuine partial replication under SI; we depict its pseudo-code in Algorithm 8. In this protocol, queries commit locally and update transactions are atomic broadcast to all replicas (lines 4 and 5). Upon delivering an update transaction, every replica performs a certification test to avoid concurrent updates to both commit (line 7). Serrano maintains at each replica the latest version number of all objects, thus allowing the protocol to skip the distributed voting phase. We capture this behavior by the fact that both $vote\_snd\_obj()$ and $vote\_recv\_obj()$ equal the local objects (line 8).

6.4 Parallel Snapshot Isolation

With the emergence of new Internet based applications, like social networks, the need for highly scalable and geo-replicated transactional systems has increased substantially over the past few years. To target this need, Sovran et al. [3] propose Parallel Snapshot Isolation (PSI), a new consistency criterion suitable for geo-replicated systems. PSI is close to SI, however unlike this criterion, PSI does not enforce monotonic snapshots of transactions. Hence queries might observe non-monotonic snapshots (as in US). Sovran et al. [3] identify monotonic snapshots as the main bottlenecks of

Algorithm 8 Serrano [10]

```
1: $choose \equiv choose_{cons}$
2: $\Theta \equiv TS$
3: $\mathcal{AC} \equiv$ GC
4: $xcast \equiv$ AB-Cast
5: $certifying\_obj(T_i) \equiv$ if $|ws(T_i)| = 0$ then $\varnothing$
                             else $Objects$
6: $commute(T_i, T_j) \equiv ws(T_i) \cap ws(T_j)$
7: $certify(T_i) \equiv \forall x_i, x_j \in ws(T_i) \times db : \Theta(x_j) \leq \Theta(x_i)$
8: $vote\_snd\_obj(T_i) = vote\_recv\_obj(T_i) \equiv LocalObjects$
```

Algorithm 9 Walter [3]

```
1: $choose \equiv choose_{cons}$
2: $\Theta \equiv VTS$
3: $\mathcal{AC} \equiv$ 2PC
4: $certifying\_obj(T_i) \equiv ws(T_i)$
5: $commute(T_i, T_j) \equiv ws(T_i) \cap ws(T_j) = \varnothing$
6: $certify(T_i) \equiv \forall x_i, x_j \in ws(T_i) \times db : \Theta(x_j) \leq \Theta(x_i)$
7: $post\_commit(T_i) \equiv$ M-Cast($\Theta(T_i)$)
                    to $(\Pi \setminus replicas(certifying\_obj(T_i)))$
```

Algorithm 10 Jessy$_{2pc}$

```
1: $choose \equiv choose_{cons}$
2: $\Theta \equiv PDV$
3: $\mathcal{AC} \equiv$ 2PC
4: $certifying\_obj(T_i) \equiv ws(T_i)$
5: $commute(T_i, T_j) \equiv ws(T_i) \cap ws(T_j) = \varnothing$
6: $certify(T_i) \equiv \forall x_i, x_j \in ws(T_i) \times db : \Theta(x_j) \leq \Theta(x_i)$
```

SI regarding scalability.

Walter: Walter (shown in Algorithm 9) is the transactional system proposed by Sovran et al. [3] to implement PSI. This protocol relies on a two-phase commit among the replicas holding an object written by the transaction (lines 3 and 4). To satisfy PSI, the certification of Walter ensures that no two concurrent write-conflicting transactions both commit (line 6). Once a transaction is committed, Walter propagates $\Theta(T_i)$ in the background to all the replicas in the system (line 7). As in the case of S-DUR, this background propagation is crucial to ensure progress.

6.5 Non-Monotonic Snapshot Isolation

As pointed out in the previous section, PSI addresses some of the scalability issues of SI. Nevertheless, Saeida Ardekani et al. [50] shows that no PSI system can ensure GPR. To sidestep this problem, Saeida Ardekani et al. [15] introduced recently a novel criterion named Non-monotonic Snapshot Isolation (NMSI). The core difference between PSI and NMSI is that NMSI allows transactions to take *any* consistent snapshot. In particular, a transaction can observe the effects of transactions that committed after it starts.

Jessy$_{2pc}$: The Jessy protocol [15] guarantees NMSI. In this paper, we shall be considering a 2PC-based variation of Jessy. This protocol, denoted Jessy$_{2pc}$, is presented in Algorithm 10. Jessy$_{2pc}$ relies on the PDV versioning mechanism to compute consistent snapshots (line 2), and it employs two-phase commit during termination (line 3). Like Serrano and Walter, Jessy$_{2pc}$ prevents two concurrent write-conflicting transactions to commit both (line 6). Notice that because Jessy$_{2pc}$ is GPR, there is no background propagation in this protocol after the commitment of a transaction.

Protocol	Source Lines of Code			
			Total	
	Exec.†	*Term.*†	G-DUR†	Original
P-Store‡	45	134	179	6000
S-DUR	199	288	397	N/A
GMU‡	184	292	476	6000
Serrano	104	247	351	N/A
Walter	322	277	599	30000
Jessy$_{2pc}$ ‡	155	197	352	6000

Table 2: Source lines of code
†: excluding comments
‡: open source

	Key Selec.	Operations	
Workload	Dist.	R-O .	Update Tran.
A	Uniform	2 Read	1 Read, 1 Update
B	Uniform	4 Read	2 Read, 2 Update
C	Zipfian	2 Read	1 Read, 1 Update

Table 3: Experimental Settings

7. IMPLEMENTATION

We implemented G-DUR, and the realized transactional protocols in Java. Our implementations closely follow the published specification of each protocol, and are highly optimized. We also implemented a DUR read-committed consistency criterion (RC). RC is a weak consistency criterion ensuring that a transaction reads a committed version of an object without any additional guarantee. It is also default consistency in many databases (e.g., Postgres 9.2.2, MS SQL Server 2012, and SAP HANA [54]). We use RC to show the maximum achievable performance in our experiments.

G-DUR can work either with a data persistence layer (i.e., BerkeleyDB), or without (i.e., an in-memory concurrent hashmap). To minimize noise, and to focus on scalability and synchronization, our experiments in this paper are done using the latter case. Should the user decide to use the data persistence layer, she can easily implement an interface, and attach any other data store.

The implementation of G-DUR and the realization of the six protocols takes approximately 10^4 source lines of code (SLOC). The communication layer also takes an additional 10^4 source lines of code. Table 2 details the amount of SLOC for each protocol. Observe that the G-DUR implementations take an order of magnitude fewer LOC than the monolithic originals. This code, the benchmarks, as well as the scripts we used in our experiments are publicly available [55].

8. CASE STUDY

This section shows the practical usages of our G-DUR middleware. We divide our study into the following four parts: *(i)* a comparison of the protocols realized in Section 6, *(ii)* an analysis of the bottlenecks of the GMU protocol, *(iii)* a demonstration of the pluggability capabilities of G-DUR, and *(iv)* an assessment of the cost of dependability.

8.1 Setup and Benchmark

All our experiments are run on the sites of the Grid'5000 experimental testbed [56]. Latencies between the sites are between 10 to 20 ms. We always use 4-core machines running between 2.2 and 2.6 GHz with a maximum heap size of 4 GB.

We performed our experiments under different configurations and using various numbers of sites. Although G-DUR can take care of intra-site replication as well, our experiments are considering a single replica per site (in a similar spirit to Sovran et al. [3]). For each replica, two additional client machines generate the workload; hence, there is no shared memory between clients and replicas. We perform our experiments in either a *Disaster Prone* (DP), or *Disaster Tolerant* (DT) configuration. In case of DP, objects are stored at a single site, while DT replicates objects at two sites.

Every replica contains 10^5 objects, and each object has a payload size of 1 KB. We employ the Yahoo! Cloud Serving Benchmark [57], modified to generate transactions. Table 3 lists the workloads we use during our experiments. These workloads have already been employed in several previous research papers [12, 15]. For a workload, a client machine emulates multiple client threads in parallel, each running in closed loop. During an experiment, a client machine executes at least 10^6 transactions. Every transaction is *interactive*, that is none of the objects it accesses is known in advance, and *global*, no replica holds all the objects read or written by the transaction. We chose this last setting to emphasize the geo-replicated performance of the protocols.

8.2 Comparing Transactional Protocols

This section compare the realizations of Section 6 in order to bring out differences between the protocols, as well as between the consistency criteria they implement.

In Figure 3-a, we depict the performance of each protocol under Workload A, when using four sites in a DP configurations. We use either 90% (top) or 70% (bottom) of read-only transactions. Each point plots the *termination latency of update transactions*, that is, the average time between the termination request of an update transaction and the reception of the response by the client, as a function of the throughput. The termination latency of the update transactions is the most meaningful metric to observe differences between the realized protocols since*(i)* all protocols (except P-Store) implement wait-free queries, and *(ii)* they all follow the DUR approach with the same execution phase (except Serrano).

Jessy$_{2pc}$ is the fastest protocol by being a genuine protocol, and requiring minimal synchronization: only replicas holding modified objects are included in transaction certification. Although Walter also enjoys minimal synchronization, being non-genuine results in smaller throughput compared to Jessy$_{2pc}$. In GMU, the replicas of both read and modified objects are involved in the certification. Yet the performance of GMU and Walter are the same with 90% read-only transactions. With 70% of read-only transactions, Walter requires more global propagation (due to its non-genuineness), and starts degrading before GMU.

In P-Store, queries are not wait-free and they have to go through AM-Cast. This design choice greatly impacts performance and explains that the throughput P-Store is the worst among the studied protocols with 90% of read-only transactions. On the other hand, P-Store compensates this gap with 70% of read-only transaction, and overtakes Serrano. This observation underlines again the importance of supporting GPR for a transactional system. This also shows that as pointed out by Lin et al. [58], the protocol of Serrano et al. is more oriented for LAN environments.

S-DUR always delivers a better throughput than Serrano. This difference points out that, unlike the general credence, SER can be faster than SI, provided the protocol implementing SER ensures wait-free queries. Finally, the performance gap between Serrano and Walter clearly motivates the use of PSI over SI in a geo-replicated setting. This assesses empirically the original argument of Sovran et al. [3].

We note that while PSI is weaker than SI [3] and US is weaker than SER [27], neither US and PSI nor SI and SER are mutually comparable [15, 27]. Thus, an anomalistic comparison between the two criteria would not explain their performance differences. On the contrary, the realization of the protocols inside G-DUR allows us to fairly compare them in terms of throughput and latency.

To better understand the differences among the protocols ensuring weaker consistency criteria, we evaluate them in Figure 3-b using Workload B and in a disaster tolerant configuration. Under 90% of read-only transactions, the performance of Walter, and Jessy$_{2pc}$ are similar. This is due to the fact that transactions contain more operations in Workload B, hence the non-genuineness of Walter does not impact performance in comparison to Jessy$_{2pc}$. The performance of GMU also degrades with Workload B. This is mainly due to the abort rate. With 1024 client threads, and 90% read-only transactions, the abort rate of GMU reaches 12% while the abort rates of Walter and Jessy$_{2pc}$ stay below 0.1%. With 30% update transactions, the abort rate of GMU deteriorates to 48%, and the abort rate of Jessy$_{2pc}$ and Walter reach around 1%.

8.3 Understanding Bottlenecks

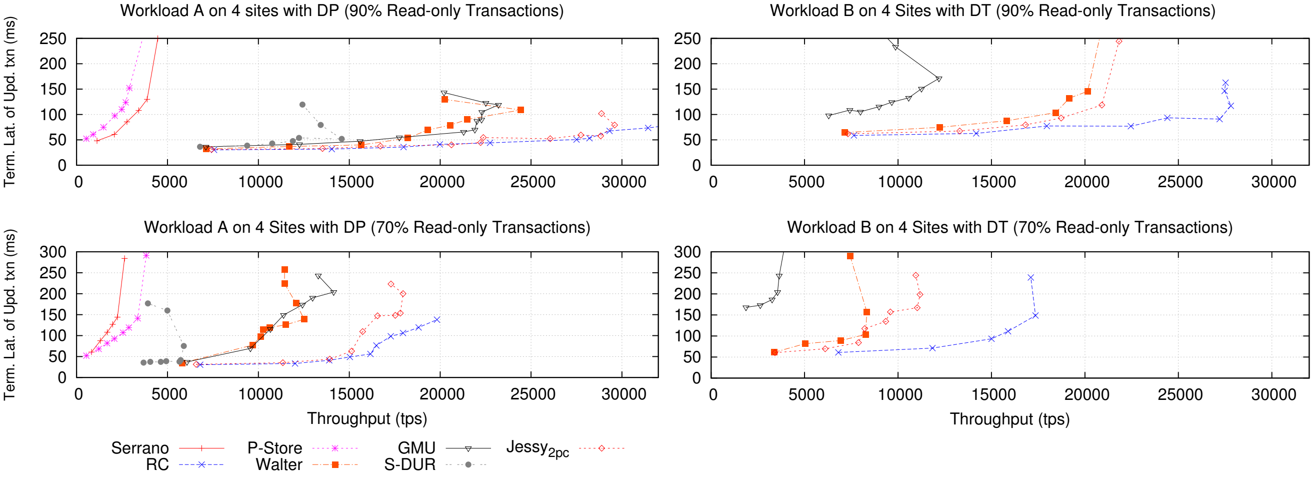
In this section, we explain how a developer can study the costs of the different components implementing a transactional protocol in order to locate its bottlenecks. Our approach consists in a careful substitution of the versioning and certification plug-ins with trivial ones. We plot our results for the GMU protocol in Figure 4.

As depicted in Algorithm 7, GMU takes consistent and fresh snapshots during the execution phase. In GMU*, we turned off this versioning component, replacing it with $choose_{last}$. However, metadata required for taking consistent snapshots is still sent during the execution phase. We observe in Figure 4 that both GMU and GMU* follow the same trend, and that the overhead of taking consistent snapshots in GMU is around 5%. With GMU**, we turn off the certification test, and all the transactions now pass the certification test. The resulting protocol follows the trend of RC, while still exhibiting a small performance gap. This difference is explained by the overhead of marshaling and sending metadata related to snapshots that GMU** inherits from the original protocol. Thus, at the light of these results, we can conclude that the certification test is the main bottleneck of the algorithm of Peluso et al. [4].

8.4 Pluggability Capabilities

G-DUR allows a developer to replace the plug-ins that compose a transactional protocol. Such a feature helps to finely understand a protocol and in turn paves the way to improve it. In this section, we demonstrate this usage with P-Store [11].

In P-Store, read-only transactions have to go through a certification test. Following an analysis similar to the one we conducted in the previous section for GMU, we can show that this mechanism is an important bottleneck of the protocol. In general and as explained in Section 6.1, this bottleneck cannot be overcome since P-Store is a genuine algorithm. However, P-Store can safely commit a read-only transaction without



(a) Disaster Prone Configuration

(b) Disaster Tolerant Configuration

Figure 3: Performance Comparison

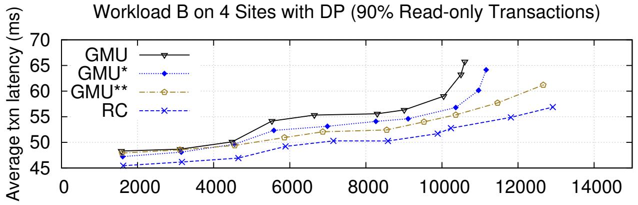


Figure 4: Study of Bottlenecks in GMU -*GMU: Consistent Snapshot & Certification, GMU*: Trivial Snapshot & Certification, GMU**: Trivial Snapshot & Trivial Certification*

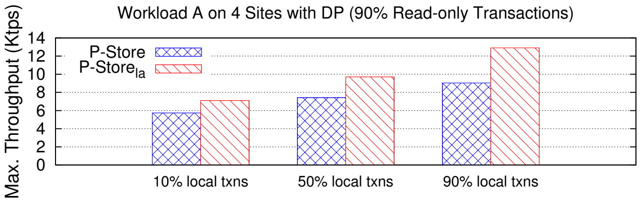


Figure 5: Throughput improvement of P-Store

certifying it when the transaction accesses a single data partition (typically a single site). We implement this feature as follows:Instead of reading the latest committed value during the execution phase, we take a consistent snapshot. We achieve this in G-DUR by using the $choose_{cons}$ component implemented with partitioned dependency vectors (PDV). We change the realization of $certifying\_obj(T_i)$ such that it returns $\varnothing$ in the case where T_i is a query accessing a single partition. Figure 5 plots the throughput of our locality aware P-Store, denoted *P-Store*$_{la}$, in comparison to the original algorithm of Schiper et al. [11]. We can observe that, depending on the ratio of local read-only transactions, *P-Store*$_{la}$ is 20 to 70% faster than the original protocol.

8.5 Dependability

As pointed out in Section 5, termination based on group communication primitives orders a priori conflicting transactions, whereas the use of 2PC relies on a spontaneous ordering of the network. The two approaches also differ in terms of fault-tolerance. The former requires either a crash-recovery model or perfect failure detection to ensure liveness, whereas the later can accommodate with faults. In this section, we compare them empirically in our geo-replicated environment.

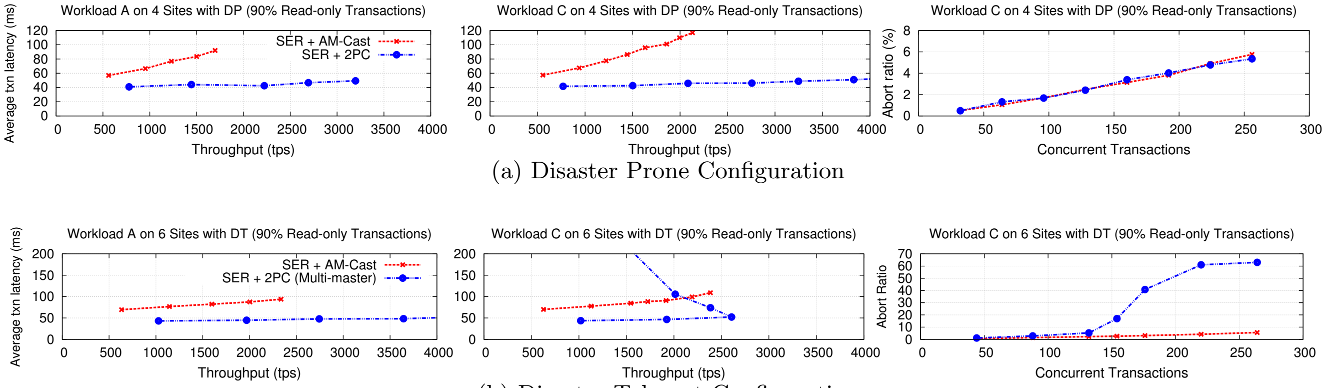
To this goal, we picked P-Store and changed its atomic commitment protocol from AM-Cast to 2PC. The rationale of this choice is that the versioning mechanism of P-Store has the smallest overhead compared to other protocols, and that this protocol certifies both read-only and updates transactions. Both features reduce noise during our measurements since we limit the amount metadata used by the system, and all the transactions go through the termination phase.

8.5.1 *Disaster Prone*

In a disaster-prone scenario, every object is replicated at a single site. Hence, when a site goes down, the system has to wait that it becomes available again. Figure 6-a compares the 2PC and AM-Cast variations of P-Store in this configuration. With Workload A, the abort ratio of both protocols is almost null and 2PC outperforms AM-Cast by a factor of at least two. Under a highly contended workload (Workload C), the abort ratio of both protocols increases similarly, and 2PC still outperforms AM-Cast. As a consequence, in such a scenario, ordering transactions a priori has a limited positive effect on the abort ratio. and it does not pay off.

8.5.2 *Disaster Tolerant*

In a disaster-tolerant setting, every object is replicated at two sites. Therefore, the system can tolerate a complete site failure. The results of this experiment are shown in Figure 6-b. Like the previous scenario, 2PC still outperforms AM-Cast with Workload A. However, under Workload C, once the sites become saturated, the abort ratio of 2PC increases drastically, due to the preemptive aborts (line 4 in Algorithm 4). Thus, in such a situation, pre-ordering the transactions in the commitment phase pays off.



(a) Disaster Prone Configuration

(b) Disaster Tolerant Configuration

Figure 6: 2PC vs. AM-Cast

9. CONCLUSION

Deferred update replication (DUR) is a classical technique to construct transactional datastores. Protocols that follow the DUR approach share a common algorithmic structure consisting in a speculative execution phase followed by a termination phase, and at core, they only differ by instantiating a few generic functions in each phase. This paper presents G-DUR, a generic deferred update replication middleware built upon this insight. G-DUR brings several benefits to practitioners and researchers in the field of transactional storage:

- It allows to easily fast prototype a transactional protocol following the DUR approach. Section 6 presented the implementation of six state-of-the-art replication protocols published in the past few years [3, 4, 10–12, 15]. Each protocol in our middleware requires less than 600 lines of code.
- G-DUR fosters apples-to-apples comparison of transactional protocols. We illustrated this in Section 8.2 by presenting an empirical evaluation in a geo-replicated environment. To the best of our knowledge, such a fair comparison never appears elsewhere in literature. The key reason is that it is either hard (or impossible) to be performed with the original implementations as source codes are generally not comparable, nor always publicly available. In addition, mastering each protocol requires a large amount of time.
- With G-DUR, a developer can study in details the limitations and overheads of her protocol. In Section 8.3, we illustrated this point with the protocols of Peluso et al. [4]. Then, we presented in Section 8.4 a variation of P-Store [11] that leverages workload locality. Our variation performs up to 70% faster than the original protocol.
- Finally, G-DUR allows us to study the cost of various degrees of dependability. In Section 8.5, we evaluated in practice the difference between commitment based on group communication primitives and 2PC in a disaster-prone and a disaster-tolerant setting.

The result of our empirical comparison in Figure 3 shows that each consistency criterion has a distinct performance domain. This illustrates in two real-word scenarios how the CAP theorem of Brewer [59] impacts transactional datastores. In a near future, we plan to refine this study by supporting additional criteria and protocols (e.g., read atomicity [2] or opacity [60]). Another direction of interest is the dynamic adaptation of consistency to the workload. To that regard, we believe that G-DUR can greatly improve our development and evaluation time thanks to its library of execution and termination plug-ins.

References

[1] J. C. Corbett *et al.*, "Spanner: Google's Globally-Distributed Database," in *OSDI*. Hollywood, CA, USA: usenix, 2012, pp. 251—-264.

[2] P. Bailis *et al.*, "Scalable atomic visibility with RAMP transactions," in *SIGMOD*, 2014.

[3] Y. Sovran *et al.*, "Transactional storage for geo-replicated systems," in *SOSP*. New York, NY, USA: ACM, 2011, pp. 385—-400.

[4] S. Peluso *et al.*, "When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication," in *ICDCS*. Macau, China: IEEE, 2012, Paper, pp. 455–465.

[5] M. Patiño Martínez *et al.*, "Scalable Replication in Database Clusters," in *DISC*. Springer, Oct. 2000, pp. 315–329.

[6] B. Kemme *et al.*, "A new approach to developing and implementing eager database replication protocols," *Trans. on Database Systems*, vol. 25, no. 3, pp. 333–379, Sep. 2000.

[7] C. Amir, Yair and Tutu, "From total order to database replication," in *ICDCS*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 494–503.

[8] F. Pedone *et al.*, "The Database State Machine Approach," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 71–98–98, Jul. 2003.

[9] N. Schiper *et al.*, "Optimistic algorithms for partial database replication," in *OPODIS*, M. M. A. A. Shvartsman, Ed., vol. 4305. Berlin, Heidelberg: Springer, Dec. 2006, pp. 81–93.

[10] D. Serrano *et al.*, "Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation," in *PRDC*. IEEE, Dec. 2007, pp. 290–297.

[11] N. Schiper *et al.*, "P-Store: Genuine Partial Replication in Wide Area Networks," in *SRDS*. IEEE, Oct. 2010, pp. 214–224.

[12] D. Sciascia *et al.*, "Scalable Deferred Update Replication," in *DSN*. IEEE Computer Society, 2012, pp. 1–12.

[13] S. Peluso *et al.*, "SCORe: a scalable one-copy serializable partial replication protocol," in *Middleware*. Springer, Dec. 2012, pp. 456–475.

[14] D. Sciascia *et al.*, "Geo-replicated storage with scalable deferred update replication," in *DSN*, 2013.

[15] M. Saeida Ardekani *et al.*, "Non-Monotonic Snapshot Iso-

lation: scalable and strong consistency for geo-replicated transactional systems," in *SRDS*, Braga, Portugal, Oct. 2013, pp. 163–172.

[16] G. Alonso, "Partial database replication and group communication primitives," Euro. Res. Seminar on Advances in Dist. Sys., 1997.

[17] M. Wiesmann *et al.*, "Comparison of database replication techniques based on total order broadcast," *IEEE Trans. on Knowlege and Data Eng.*, vol. 17, no. 4, pp. 551–566, 2005.

[18] R. Schmidt *et al.*, "A formal analysis of the deferred update technique," in *OPODIS*, 2007, pp. 16–30.

[19] A. Bieniusa *et al.*, "Consistency in hindsight: A fully decentralized STM algorithm," in *IPDPS*. IEEE, 2010, pp. 1–12.

[20] P. Felber *et al.*, "Time-based software transactional memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 12, pp. 1793–1807, 2010.

[21] M. Wiesmann *et al.*, "Understanding replication in databases and distributed systems," in *ICDCS*. IEEE Computer Society, 2000, pp. 464–474.

[22] M. Wiesmann *et al.*, "Database Replication Techniques: a Three Parameter Classification," in *SRDS*. IEEE Computer Society, 2000, pp. 206—-.

[23] M. Nicola *et al.*, "Performance modeling of distributed and replicated databases," *IEEE Trans. on Knowl. and Data Eng.*, vol. 12, no. 4, pp. 645–672, Jul. 2000.

[24] Oracle, *Getting Started with Berkeley DB*. Oracle.

[25] Microsoft Corporation, "SQL Server 2012," 2012.

[26] H. Berenson *et al.*, "A critique of ANSI SQL isolation levels," in *SIGMOD*. New York, New York, USA: ACM Press, 1995, pp. 1–10.

[27] A. Adya, "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions," Ph.D., MIT, Cambridge, MA, USA, Mar. 1999.

[28] A. Adya *et al.*, "Generalized isolation level definitions," in *ICDE*, no. March. IEEE Comput. Soc, 2000, pp. 67–78.

[29] M. Patiño Martinez *et al.*, "Middle-R: Consistent database replication at the middleware level," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 375–423, Nov. 2005.

[30] J. Correia, A. *et al.*, "Akara: A flexible clustering protocol for demanding transactional workloads," in *On the Move to Meaningful Internet Systems: OTM 2008*, ser. LNCS, R. Meersman *et al.*, Eds. Springer-Verlag, 2008, vol. 5331, pp. 691–708.

[31] J. Bernabé-Gisbert *et al.*, "Managing Multiple Isolation Levels in Middleware Database Replication Protocols," in *Parallel and Distributed Processing and Applications*, ser. Lec. Notes in Comp. Sc., M. Guo *et al.*, Eds. Springer, 2006, vol. 4330, pp. 511–523.

[32] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[33] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. Workshop on Parallel and Distributed Algorithms*, C. M. et al., Ed., North-Holland / Elsevier, 1989, pp. 215–226.

[34] D. S. Parker *et al.*, "Detection of mutual inconsistency in distributed systems," *IEEE Trans. Softw. Eng.*, vol. 9, no. 3, pp. 240–247, May 1983.

[35] J. Almeida *et al.*, "Bounded version vectors," in *Distributed Computing*, ser. LNCS, R. Guerraoui, Ed. Springer-Verlag, 2004, vol. 3274, pp. 102–116.

[36] R. Guerraoui *et al.*, "Permissiveness in transactional memories," in *DISC*, 2008, pp. 305–319.

[37] S. Peluso *et al.*, "On Breaching the Wall of Impossibility Results on Disjoint-Access Parallel STM," Virginia Tech, Tech. Rep., 2014.

[38] R. Guerraoui *et al.*, "On obstruction-free transactions," in *SPAA*. New York, NY, USA: ACM, 2008, pp. 304–313.

[39] M. Saeida Ardekani *et al.*, "The space complexity of transactional interactive reads," in *HotClouds*. New York, New York, USA: ACM Press, Apr. 2012, pp. 1–5.

[40] X. Défago *et al.*, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comp. Surveys*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[41] F. Schintke *et al.*, "Enhanced Paxos Commit for Transactions on DHTs," in *CCGrid*. Washington, DC, USA: IEEE, May 2010, pp. 448–454.

[42] M. Blasgen *et al.*, "The convoy phenomenon," *SIGOPS Oper. Syst. Rev.*, vol. 13, no. 2, pp. 20–25, Apr. 1979.

[43] T. D. Chandra *et al.*, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[44] R. Guerraoui *et al.*, "Genuine atomic multicast in asynchronous distributed systems," *Theoretical Computer Science*, vol. 254, no. 1-2, pp. 297–316, Mar. 2001.

[45] N. Schiper, "On multicast primitives in large networks and partial replication protocols," Ph.D. dissertation, U. of Lugano, 2009.

[46] N. Schiper *et al.*, "Genuine versus Non-Genuine Atomic Multicast Protocols for Wide Area Networks: An Empirical Study," in *SRDS*. IEEE, Sep. 2009, pp. 166–175.

[47] L. Lamport, "Lower bounds for asynchronous consensus," *Distributed Computing*, vol. 19, no. 2, pp. 104–125, October 2006.

[48] D. Dolev *et al.*, "Early-deciding consensus is expensive," in *PODC*. New York, New York, USA: ACM Press, Jul. 2013, p. 270.

[49] P. Sutra *et al.*, "Fault-tolerant partial replication in large-scale database systems," in *Euro-Par*, Las Palmas de Gran Canaria, Spain, Aug. 2008, pp. 404–413.

[50] M. Saeida Ardekani *et al.*, "On the Scalability of Snapshot Isolation," in *Euro-Par*, F. Wolf *et al.*, Eds., vol. 8097, Aachen, Germany, Aug. 2013, pp. 369–381.

[51] H. Garcia-Molina *et al.*, "Read-only transactions in a distributed database," *Trans. on Database Systems*, vol. 7, no. 2, pp. 209–234, Jun. 1982.

[52] R. C. Hansdah *et al.*, "Update serializability in locking," in *Lec. Notes in Comp. Sc.*, ser. Lec. Notes in Comp. Sc., G. Ausiello *et al.*, Eds. Springer, 1986, vol. 243, pp. 171–185.

[53] S. Elnikety *et al.*, "Database Replication Using Generalized Snapshot Isolation," in *SRDS*. IEEE Computer Society, Oct. 2005, pp. 73–84.

[54] P. Bailis *et al.*, "Highly Available Transactions : Virtues and Limitations," in *VLDB*, 2014.

[55] M. Saeida Ardekani *et al.*, "Jessy - *https://github.com/msaeida/jessy*," 2013.

[56] Grid'5000, "Grid'5000, a scientific instrument [. . .]," , retrieved April 2013.

[57] B. Cooper *et al.*, "Benchmarking Cloud Serving Systems with YCSB," in *SoCC*. New York, NY, USA: ACM, 2010, pp. 143—-154.

[58] Y. Lin *et al.*, "Consistent data replication: Is it feasible in wans?" in *Euro-Par 2005 Parallel Processing*, ser. LNCS, J. Cunha *et al.*, Eds. Springer Berlin Heidelberg, 2005, vol. 3648, pp. 633–643.

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

[60] R. Guerraoui *et al.*, "On the correctness of transactional memory," in *PPoPP*. New York, NY, USA: ACM, 2008, pp. 175–184.