RW-Through: A Data Replication Protocol Suitable for Geo Distributed and Read-Intensive Workloads

Wagner R. M. Barretto, Ana Cristina B. Kochem Vendramin, Mauro Sérigo Pereira Fonseca

Informatics Department (DAINF) - Graduate Program in Applied Computing (PPGCA)
Federal University of Technology - Paraná (UTFPR)

wagner.rezende@gmail.com, {criskochem, maurofonseca}@utfpr.edu.br

Abstract. *In large-scale modern applications, users consume much more data than they create. To guarantee that data storage systems appropriately handle read-intensive workloads, application designers have been adopting data replication and caching techniques. These techniques face many challenges regarding data consistency, specially in faulty network conditions such as wide-area networks. In this work we use recent distributed consistency frameworks to identify the consistency models of the main solutions for read-intensive workloads used today in the industry. Additionally, we present RW-Through, a new data replication protocol suited for large-scale read-intensive workloads. Our new protocol provides stronger consistency guarantees than current solutions and is designed to tolerate unreliable network conditions, making it suitable for geo-distributed deployments.*

1. Introduction

Modern large scale applications manage high volumes of data. In these applications, users consume more data than they produce [Nishtala et al. 2013]. This behavior results in workloads dominated by reads and searches to the various storage systems that hold the application's data. Additionally, usage patterns of social networks indicates that the majority of the reads are concentrated on a small subset of the data [Bruner 2013].

Cloud providers and the industry have been adopting solutions such as database replication and caching in order to tackle read-intensive workloads scenarios. Database replication solutions for read intensive workloads generally consist of *Lazy Primary Copy Replication* (LPCR). LPCR is a common database replication technique [Gray et al. 1996, Pacitti et al. , Cecchet et al. 2008] where one node of a system is responsible for the write operations whereas a different set of nodes are responsible for the read operations. Caching solutions often consist of *Demman-filled read-aside caching* or, *Cache-aside*. Cache-aside is a common technique in web applications [Nishtala et al. 2013] where a read-optimized (often in-memory) data store is used to "cache" a subset of a system's data with a high chance of being accessed.

These solutions were born in single datacenter environments and face challenges when applied to geo-distributed cloud deployments. Specifically, both solutions are known to face distributed consistency problems. Such problems become more frequent when unreliable communication links are used.

Distributed consistency is an area that faces several advancements due to the emergence of large-scale distributed systems. Works in the area revolve around limitations inherent from distributed systems such as the CAP Theorem [Brewer 2000, Gilbert and Lynch 2002]. These limitations motivated the study of less restrictive consistency models such as eventual or causal consistency. These consistency models have relaxed requirements when compared to *Linearizability* [Herlihy and Wing 1990], the most restrictive of the models. Motivated by a large number of works introducing different consistency models, a work [Burckhardt 2014] introduced a mathematical framework capable of precisely defining the whole spectrum of consistency in distributed systems. This framework enabled the precise and formal identification of which consistency model a certain system is capable to provide.

In this paper we present two main contributions: we derive the general algorithms for LPCR and *Cache-aside* and formally analyze its consistency properties; and we present *RW-Through*, a new data replication protocol capable of addressing the same problems as the mentioned solutions while being more consistent, efficient in network usage and more suitable for geo-replicated deployments. The rest of the paper is organized as follows. Section 2 summarizes distributed consistency models and frameworks. Section 3 presents an analysis of the consistency model offered by LPCR and *Cache-aside*. Section 4 presents the *RW-Through* algorithms and their consistency analysis. Section 5 discusses how *RW-Through* compares to both LPCR and cache-aside. Section 6 presents future works. Finally, Section 7 presents concluding remarks.

2. Rationale and Background

This section briefly summarizes the main concepts behind consistency in distributed systems required in order to comprehend our analyses. The main focus is on the works done by [Burckhardt 2014] and [Viotti and Vukolić 2016]. It is important to note that the concept we are interested here is *Distributed Systems Consistency*, which is different from the concept of consistency commonly explored on the database literature. The concept of consistency in databases (i.e. the "C" in ACID) refers to the non violation of constraints the system's data may have. In distributed systems, consistency refers to how clients see a system's state when the system replicates such state among a group of different processes.

2.1. Distributed Consistency Specifications

We consider a distributed storage system as a finite set of processes connected through an asynchronous network. The system's purpose is to manage a collection of *Objects*. Clients are a separate set of processes that manipulate the objects by means of *Operations* submitted to the system's processes. A set of operations invoked during a given execution of the system is called a *History* (often referred as H). To represent notions of order or equivalence among the operations of a history, binary relations over the history are used. The notation $a \xrightarrow{\mathsf{rel}} b$ denotes that $(a, b) \in \mathsf{rel}$. The following relations are used in consistency models specifications [Burckhardt 2014].

- rb (*returns-before*): is an order relation based on real-time precedence. Denotes that an operation returned before other started. Formally: $\mathsf{rb} \stackrel{\text{def}}{=} \{(a, b) \mid a, b \in H \wedge a.rtime < b.stime\}$ where stime and rtime are operations start and retur times respectively.

Ordering Guarantees	
EVENTUALVISIBILITY	$\forall a \in H, \forall [f] \in H/\approx_{ss}: \lvert\{b' \in [f] \lvert (a \xrightarrow{\mathsf{rb}} b) \wedge (a \not\xrightarrow{\mathsf{vis}} b)\}\rvert < \infty$
READMYWRITES	$\mathsf{so} \subseteq \mathsf{vis}$
MONOTONICREADS	$\forall a \in H, \forall b, c \in H\lvert_{rd} : a \xrightarrow{\mathsf{vis}} b \wedge b \xrightarrow{\mathsf{so}} c \Rightarrow a \xrightarrow{\mathsf{vis}} c$
CAUSALVISIBILITY	$\mathsf{hb} \subseteq \mathsf{vis}$
CAUSALARBITRATION	$\mathsf{hb} \subseteq \mathsf{ar}$
SINGLEORDER	$\exists H' \subseteq \{op \in H \mid op.oval = \nabla\} : \mathsf{vis} = \mathsf{ar} \setminus (H' \times H)$
REALTIME	$\mathsf{rb} \subseteq \mathsf{ar}$
Consistency Models	
BASICEVENTUALCONSISTENCY	EVENTUALVISIBILITY ∧ NOCIRCULARCAUSALITY
CAUSALCONSISTENCY	CAUSALVISIBILITY ∧ CAUSALARBITRATION
SEQUENTIALCONSISTENCY	READMYWRITES ∧ SINGLEORDER
LINEARIZABILITY	SINGLEORDER ∧ REALTIME

Table 1. Ordering guarantees and consistency models [Burckhardt 2014]

- ss (*same-session*): is an equivalence relation based on the invoking process. Two operations invoked by the same process are said to be in the same session. Formally: $\mathsf{ss} \stackrel{\text{def}}{=} \{(a, b) \mid a, b \in H \wedge a.proc = b.proc\}$ where proc is the client proccess that invoked the operation.
- so (*session-order*): is an order relation between operations of the same session. It can be expressed as the intersection of the two aforementioned relations. Formally: $\mathsf{so} \stackrel{\text{def}}{=} \mathsf{rb} \cap \mathsf{ss}$

Histories and their relations enable the representation of the observable behavior of a system. However they don't justify this behavior. Two other relations are used to explain why a history is the way it is. These relations can explain situations caused by non deterministic components of a system such as asynchronous networks and implementation-specific details. A history with these added relations is called an *Abstract Execution*. The relations are [Burckhardt 2014]:

- vis (visibility): an operation a is visible to an operation b ($a \xrightarrow{\mathsf{vis}} b$) if the effects of a are visible to the process that invokes b. For example, b reads a value written by a.
- ar (arbitration): denotes conflict resolution. Given two non-related operations a and b, $a \xrightarrow{\mathsf{ar}} b$ means that the system considers that a occurred before b.

An *Ordering Guarantee* is a logic predicate over an abstract execution. An abstract execution is said to be justified by a given ordering guarantee if the predicate is true for the execution. A *Consistency Model* is a collection of ordering guarantees. A system is said to provide a given consistency model if all of its histories can be justified by the ordering guarantees that compose the model.

With the notions of relations and ordering guarantees, the consistency models can be specified. Table 1 presents the formal definitions of the main ordering guarantees and consistency models. Next we describe the main characteristics for each model.

Informally, BASICEVENTUALCONSISTENCY can be defined as a guarantee that if additional operations stop being invoked on a given object, the state of the object on all

replicas will eventually be the same [Bailis and Ghodsi 2013]. This definition describes the *convergence* property of an eventually consistent system. Formally, convergence is expressed by the guarantee EVENTUALVISIBILITY. The guarantee states that a completed operation must eventually become visible to all other sessions. The guarantee EVENTUALVISIBILITY in conjunction with the guarantee NOCIRCULARCAUSALITY form BASICEVENTUALCONSISTENCY.

BASICEVENTUALCONSISTENCY offers very few guarantees and therefore allows a number of anomalies to be observed. The anomalies more easily observed by users are those that affect the sessions. To enforce that users can read objects that they wrote the guarantee READMYWRITES is defined. The guarantee can be formally defined by requiring that the session order be a subset of visibility. Another form of session anomaly is observed when users that read an object can not read it again in a future moment. The guarantee MONOTONICREADS prevents this.

Session guarantees do not rule out causality related anomalies. Causality in distributed systems is a notion that was explored by many past works [Lamport 1978, Schwarz and Mattern 1994, Ahamad et al. 1995]. Two operations are said to be causally related if the operations were issued in the same session or if one operation is visible by another. Using relations, causality can be expressed as the *happens-before* relation, which is the transitive union between session order and visibility. Formally: $\mathsf{hb} \stackrel{\text{def}}{=} \mathsf{so} \cup \mathsf{vis}$ [Burckhardt 2014]

The happens-before relation serve as a basis for two ordering guarantees that prevent cause-related anomalies: CAUSALVISIBILITY and CAUSALARBITRATION. The first one is when the system orders operations in a way that violates the happens-before relation. This can lead to situations where users see "answers" before "questions". The second anomaly is when causally related operations are not visible to one another. This can lead to situations where users see chains of events with "holes" in them. The combination between the two causality guarantees form the consistency model CAUSALCONSISTENCY.

CAUSALCONSISTENCY fails to enforce that a single global order of operations exist within a system. The *Dekker Test* is a test designed to verify if a system enforces a single global order of operations. In the test, two processes write an object and right after read the value that the other process wrote. The pseudocode for the test is described in Table 2. If a system permits that both "A wins" and "B wins" get printed, the system does not pass the test and, consequently, does not guarantee a single order of operations.

To express the notion of a single global order of operations, the ordering guarantee SINGLEORDER is defined. A system that passes the Dekker test provide this guarantee. The guarantee SINGLEORDER combined with the guarantee READMYWRITES form SEQUENTIALCONSISTENCY. A system that provides this model, implicitly provides CAUSALCONSISTENCY [Burckhardt 2014].

SEQUENTIALCONSISTENCY still do not capture single copy semantics as it permits stale reads to occur. To capture single copy semantics, arbitration has to comply with real time. The ordering guarantee REALTIME defines that. Guarantees SINGLEORDER and REALTIME combined form the most restrictive consistency model, LINEARIZABILITY [Burckhardt 2014].

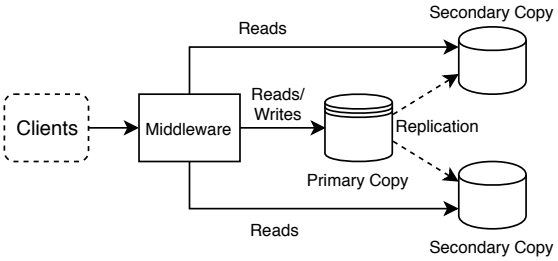

Figure 1. Lazy Primary Copy Replication

Program A	Program B

$x \leftarrow$ "active";
if $y = \varnothing$ **then**
  print "A wins";
end
	$y \leftarrow$ "active";
	if $x = \varnothing$ **then**
	  print "B wins";
	end

Table 2. Dekker test pseudocode [Burckhardt 2014]

With the implications among the consistency models, an objective "strength" order can be derived. It is said that a model A is "stronger" than B if every valid history of A is also valid over B. With this definition, the following hierarchy among the consistency models holds: LINEARIZABILITY $>$ SEQUENTIALCONSISTENCY $>$ CAUSALCONSISTENCY $>$ BASICEVENTUALCONSISTENCY [Burckhardt 2014]

3. Existing Solutions Consistency Analysis

In this section we use the distributed consistency definitions described previously to analyze the consistency models of LPCR and cache-aside.

3.1. Lazy Primary Copy Replication

Grey et. al. [Gray et al. 1996] categorizes database replication into four categories based on the combination of two properties: *Propagation* and *Ownership*. Propagation can be either *Lazy* (i.e. replication happens after the operation is returned to the client) or *Eager* (i.e. replication occurs before the operation is returned to the client). Ownership can be either *Primary Copy* (i.e. writes are centralized in one process) or *Update-Anywhere* (i.e. writes can happen in every process).

In this work we are interested in *Lazy Primary Copy Replication* since this is the replication strategy that suits better read-intensive workloads. A general architecture [Cecchet et al. 2008] of a system using LPCR is illustrated by Figure 1. The storage layer is composed by one *Primary Copy* (sometimes called *master*) and multiple *Secondary Copies* (sometimes called *slaves*). Clients submit operations to a *Middleware* layer responsible for sending write operations to the primary copy and read operations to the secondary copies. The primary copy is responsible for sending the write operations to the secondary copies. This process happens asynchronously, i.e. after the operations are returned to clients. Secondary copies hold a complete copy of the objects that the system manages, hence the strategy is said to be *full-replication*.

In LPCR, strong models can be immediately discarded. The reason is demonstrated by the execution of the Dekker test illustrated by Figure 2. I can be observed that a delay in the replication of the two writes from the primary copy to the secondary copy caused both reads to return $\varnothing$, which means a failure to provide the guarantee SINGLEORDER. If the execution is modified so that each program reads the value that it just wrote, the results would still be $\varnothing$. This shows that session order is not a subset of visibility ($\mathsf{so} \not\subset \mathsf{vis}$), which violates READMYWRITES. We can also infer that if $\mathsf{so} \not\subset \mathsf{vis}$,

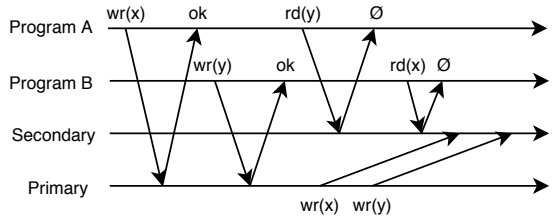

Figure 2. Possible Dekker test execution in an LPCR system

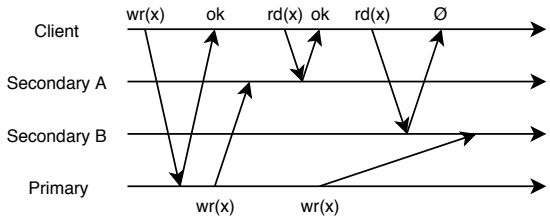

Figure 3. Execution with replication delay in an LPCR system

than $(\mathsf{so} \cup \mathsf{vis}) \not\subset \mathsf{vis}$, which violates CAUSALVISIBILITY. Since writes are centralized, the guarantee CAUSALARBITRATION is provided.

Figure 3 illustrates an execution that demonstrates that the guarantee MONOTONICREADS is not provided. The figure shows a client communicating with an LPCR system with two secondary copies A and B. After a write, two reads are routed to different secondary copies. A delay in the replication causes the second read to return an earlier version of the object. This situation violates the guarantee MONOTONICREADS.

The convergence property is achieved by LPCR since the primary copy needs to be always available for the system operation. Therefore replication to secondary copies never stops. Given the ordering guarantees provided by the solution, its consistency model can be defined as BASICEVENTUALCONSISTENCY plus the guarantee CAUSALARBITRATION.

3.2. Cache-aside

A common practice to increase performance of storage systems that handle read-intensive workloads is to use an in-memory data store alongside it. The in-memory data store acts as a cache to the system's objects and is managed by an application that uses the data. This is a popular technique in web applications where the web server is the main client of the storage system.

The solution is composed by two data stores, a "main" store that is used for general purpose and guarantees durability of the data (e.g. a relational database); and a "cache" store that is highly optimized for reads and often uses volatile memory to store the data (e.g. memcached). An application layer handles both stores and is responsible to fill the cache in an on-demand fashion. To handle possible out of space problems, an eviction mechanism is employed to free space as new objects are inserted into the store. When processing reads, the application first checks if the object exists in cache, if yes it is returned to the client, otherwise the application reads the object from the main storage, writes it in the cache, and returns it to the client. On writes the application writes the object in the main storage and cache sequentially.

As long as only one instance of application and cache exists and no other process writes to the main storage, no consistency anomalies occur. This happens because the application essentially serves as a single client to the storage system. This configuration is very limited and not suitable for geo-replicated scenarios. For the consistency analysis, we assume that the main storage can have multiple writers, e.g. multiple instances of applications and caches.

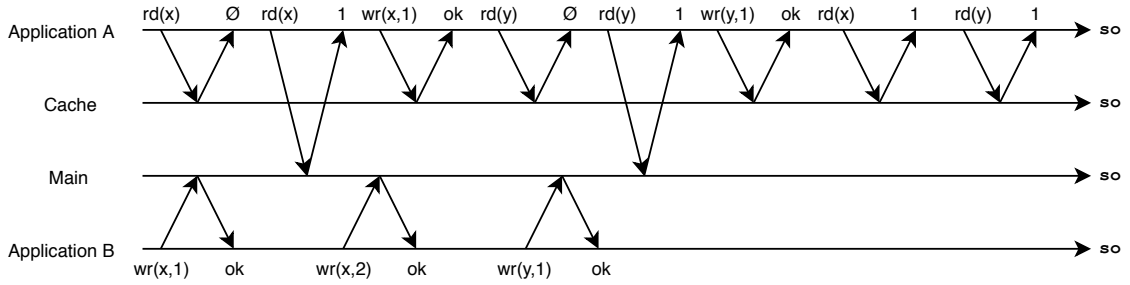

Figure 4. Cache-aside execution with two processes writing to the main storage

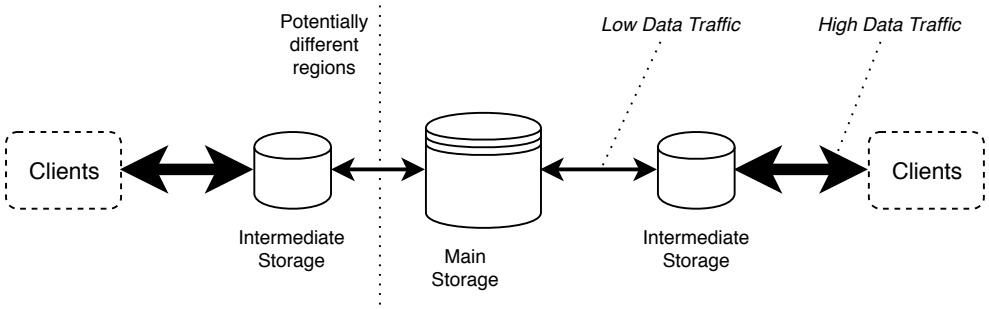

Figure 5. *RW-Through* general architecture

In configurations with multiple writers, strong models are not provided as illustrated by Figure 4. A second writer "Application B" issues three writes for two objects to the main storage. According to the happens-before relation, we have $wr(x,1) \xrightarrow{\text{hb}} wr(x,2) \xrightarrow{\text{hb}} wr(y,1)$. The process "Application A" keeps reading the key x with a stale value until an eviction occurs. This execution demonstrates that the guarantees SINGLEORDER and CAUSALVISIBILITY are not provided.

As the application accesses only a single cache and updates the cache after each write, both session guarantees are provided by the solution. As in LPCR, the centralized writes on cache-aside guarantee that CAUSALARBITRATION is provided. Convergence happens because the eviction mechanism prevents the cache from keeping the same objects forever. Given the ordering guarantees presented, we can conclude that the cache-aside solutions offers BASICEVENTUALCONSISTENCY plus the guarantees READMYWRITES, MONOTONICREADS, and CAUSALARBITRATION.

4. RW-Through Specification

In this section we present a new algorithm for data replication suitable for read-intensive workloads and large scale applications. Similarly to LPCR and *Cache-aside*, our algorithm uses a centralized process for writes and a set of processes for reads. The main difference between our algorithm and LPCR and *Cache-aside* is the way clients interact with the system. In our algorithm, clients never communicate directly with the process responsible for writes. Instead all operations go to the set of processes responsible for reads. These processes act as an intermediate layer, processing some of the operations and making others pass *through* them as illustrated by Figure 5. For this reason we call the algorithm *Read Write Through* or simply *RW-Through*.

We present two variations of *RW-Through*: *full replication* and *cache*. The full replication version has similar properties to LPCR, where every process keeps a full copy of the data managed by the system. The cache version is similar to *Cache-aside* where the intermediate layer store a subset of the objects and employs an eviction mechanism. We refer to the two types of processes as *Main Storage* and *Intermediate*.

4.1. Full Replication Mode

The full replication mode of *RW-Through* are described by Algorithms 1 and 2. Inter-process communication is modeled as Remote Procedure Calls (RPC) exposed by the *Main Storage* process.

Data:
LS: local storage;
log: write log;
$time$: process current logical time;

RPC MAINGET(t)**:**
  return $log[t \mapsto]$;
end

RPC MAINPUT(key, val, t)**:**
  $LS[key] \leftarrow val$;
  $time++$;
  $log[time] \leftarrow (key, val, time)$;
  return $log[t \mapsto]$;
end

Algorithm 1: Pseudocode of the Main Storage Process in Full Replication Mode

Data:
LS: local storage;
$time$: process current logical time;

Operation GET(key)**:**
  return $LS[key]$;
end

Operation PUT(key, val)**:**
  $log \leftarrow$ ORIGINPUT($key, val, time$);
  APPLY(log);
  return;
end

Procedure APPLY(log)**:**
  while log is not empty **do**
    $(k, v, t) \leftarrow$ dequeue log;
    if $t > time$ **then**
      $LS[k] \leftarrow v$;
      $time \leftarrow t$;
    end
  end
end

Algorithm 2: Pseudocode of the Intermediate Process in Full Replication Mode

The *Main Storage* process keeps three data structures in its internal state: LS, log, and $time$. LS is a map of keys and values and act as the main storage for the system objects. log is a record of all write operations performed on the system. Each operation is represented by an array of tuples (k, v, t), where k and v represent the key and value of object written and t the logical time associated with the operation. $time$ is the current logical time of the last write operation.

The *Main Storage* process exposes two RPCs: MAINGET and MAINPUT. MAINGET receives a logical time t as input and returns a subset of log containing the operations with logical $time > t$ (indicated with the notation $log[t \mapsto]$. MAINPUT receives as input key and val, which represent the object being written and t, which is the logical time of process that invoked the RPC. Upon receiving a MAINPUT request, the *Main Storage* process updates its local storage with the object received, increments its local variable $time$, appends the operation to its log, and returns a subset of log with the operations $time > t$.

The *Intermediate* process, similarly to MAIN STORAGE, keeps LS and $time$ in its local state. The process exposes two operations to clients: GET and PUT. GET receives as input a key and returns the value associated with this key in its LS. PUT receives as input a key and value pair and send it to the *Main Storage* process with its current logical time. With the write log resulted from the RPC call, the process invokes the auxiliary procedure APPLY. The procedure takes a write log as input and iterates over it similarly to a *First-In-First-Out* (FIFO) queue. For each dequeued object, the procedure checks if the object's logical time is higher than the process's current logical time; if it is, the key and value pair is applied in the local storage and process current time becomes the object's logical time. The process also executes a second auxiliary procedure GETLOG periodically. This procedure simply invokes the GETLOG RPC and applies the resulting log.

The algotithm has two main drawbacks: (1) high usage of space as the log grows on *Main Storage*. This can be solved by keeping track of current times for the *Intermediate* processes and implementing a cleanup routine that evitcs from the log every operation with logical time lower than the lowest of *Intermediate*'s logical time. (2) the high occurrence of stale reads when writes are not issued frequently on a particular *Intermediate*. This can be mitigated by periodically invoking MAINGET from *Intermediates*.

4.2. Cache Mode

RW-Through can also operates in cache mode supporting a partial replication scheme similar to *Cache-aside* where the *Intermediate* processes act as a demmand-filled cache. The general architecture is still the same, i.e. clients communicate only with the intermediate layer and contact the same *Intermediate* process instance for the duration of a session.

The algorithms for the *Main Storage* and *Intermediate* processes are described by Algorithms 3 and 4, respectively. The main idea of this mode is to keep on *Main Storage* state one write log for each *Intermediate* process instance. These write logs contain only operations on the objects stored by each *Intermediate*.

The cache mode of *RW-Through* is capable of using storage space and network more efficiently. Storage space efficiency is achieved by storing on the intermediate layer only the subset of data that has the highest chance of being required. Network usage efficiency is achieved by sending write logs with only the elements relevant to a particular *Intermediate* instance.

The *Main Storage* process needs to employ a mechanism to identify which objects are stored on each *Intermediate* instance. This can be achieved in a variety of ways such as hash tables or binary trees. In our implementation we choose *Cuckoo Filters* [Fan et al. 2014]. A cuckoo filter is a recent probabilistic data structure that enables set membership verification using space many times smaller than the set itself. The drawback is that the data structure allows a small number of false-positive results. The cuckoo filter is very similar to *Bloom Filters* [Bloom 1970]. The main difference between them is that cuckoo filter supports set member deletion while bloom filter does not.

We use the notation $\mathbb{I}$ to denote the set of *Intermediate* process instances. The *Main Storage* process keeps the following data structures in its internal state: A local storage LS; a set of *Cuckoo Filters* $F = \{f_i \mid i \in \mathbb{I}\}$ that maintains a compact registry of which key is currently stored in each *Intermediate* instance; a set of write logs $L = \{l_i \mid$

$i \in \mathbb{I}\}$ that represents the modifications to be executed in each *Intermediate* instance; a vector clock $T = \{t_i \in \mathbb{N} \mid i \in \mathbb{I}\}$ that keeps the highest logical time reported by each *Intermediate* instance.

```
Data:
LS: local storage;
L: log with all write operations performed;
T: set of logical times, one for each intermediate;

RPC MainGet(key, t, i):
    val ← LS[key];
    if val ≠ ∅ then
        add key to F_i;
        T_i++;
        enqueue (key, val, T_i) in L_i;
    return L_i;
end

RPC MainPut(key, val, t, i):
    add k to F_i;
    LS[key] ← val;
    foreach j ∈ 𝕀 do
        if key exists in F_j then
            T_i++;
            enqueue (key, val, T_j) in L_j;
        end
    end
    CleanLog(L_i, t);
    return L_i;
end

RPC Evict(i, key):
    remove key from F_i;
end

Procedure CleanLog(log, t):
    while top of log has position ≤ t do
        dequeue from log;
    end
end
```

Algorithm 3: Pseudocode of the Main Storage Process in Cache Mode

```
Data:
LS: local storage;
i: process identifier;
time: process logical time;

Operation Get(key):
    if LS[key] = ∅ then
        log ← MainGet(key, time, i);
        Apply(log);
    end
    return LS[key];
end

Operation Put(key, val):
    log ← MainPut(key, val, time, i);
    Apply(log);
    return;
end

Procedure Apply(log):
    while exist elements in log do
        (k, v, t) ← dequeue from b;
        if t > time then
            e ← LS[k] ← v;
            time ← t;
            if e ≠ ∅ then
                Evict(k, i);
            end
        end
    end
end
```

Algorithm 4: Pseudocode of the Intermediate Process in Cache Mode

On cache mode, the *Main Storage* process follows a similar behavior to the full replication mode. The main difference lies in the *Main Storage* process returning the correspondent write log for each *Intermediate* instance. To keep track of which object is stored in each *Intermediate*, the *Main Storage* adds every key sent as input to MAINGET or MAINPUT RPCs to the *cuckoo filter* relative to the *Intermediate* instance that invoked the RPC. Whenever an object is evicted from an *Intermediate* local storage, an EVICT RPC is invoked. This RPC simply removes the object's key from the respective *cuckoo filter*. Before sending the write logs, they are submitted to a cleanup procedure CLEANLOG that removes every entry with logical time lower than the *Intermediate*'s current.

The *Intermediate* process in cache mode has two main differences from the full

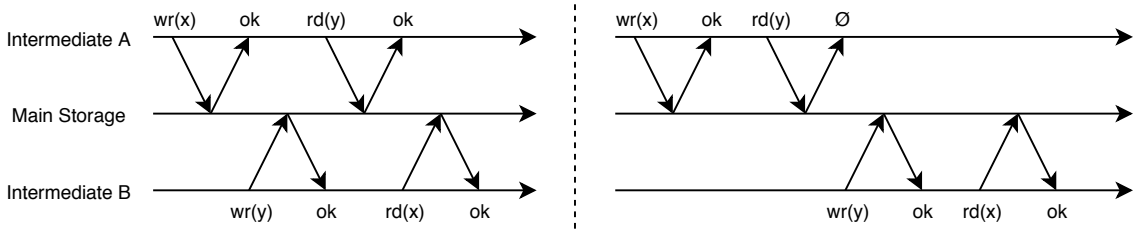

Figure 6. Dekker test executions on *RW-Through*

replication mode. The first is its GET operation. When a key is not found on local storage, the process submits a MAINGET RPC to fetch the requested object. The second is in the local storage that evicts items when its capacity is full.

4.3. Consistency Analysis

Since the full replication mode is a simplified version of cache mode, we only present an analysis for the cache mode. All properties observed in this mode are also valid on full replication mode. Initially, we proceed by defining the arbitration and visibility relations. Since writes are centralized in the *Origin* process, arbitration can be defined as the session order of that process: $\mathsf{ar} = \mathsf{eo}|_{orig}$. An *Intermediate* process instance has visibility to every operation that happened up until the process last communication with the *Main Storage* process. Formally, visibility can be defined as $a \xrightarrow{\mathsf{vis}} b \stackrel{\text{def}}{=} a \xrightarrow{\mathsf{ar}} \mathsf{lc}(b)$ where $\mathsf{lc}(op) : last\_op$ (*last communication*) is a function that receives as input an operation op and returns the last operation that communicates with the *Main Storage* that precedes op.

To verify the ordering guarantees provided by *RW-Through*, we begin by analyzing the Dekker test execution. Figure 6 shows two executions of the test in a *RW-Through* system with two *Intermediate* processes, one for each test program. These two executions can be generalized to all possible executions of the test in an *RW-Through* system. For this, one only have to symmetrically invert the two programs. It can be seen that the solution passes the test, which means that *RW-Through* provides SINGLEORDER.

In a given session, writes will always be visible to subsequent operations in the same session. To demonstrate this, let a write operation a and an operation b that reads the object written by a such that $a \xrightarrow{\mathsf{so}} b$. After a completes, the *Intermediate* local storage will immediately contain the written object. The outcome of b can be either a cache-hit or a cache-miss. If the outcome is a cache-hit, the object is successfully retrieved, thus $a \xrightarrow{\mathsf{vis}} b$. If the outcome is a cache-miss, a communication with the *Main Storage* is required, thus $a \xrightarrow{\mathsf{ar}} b$. Given the visibility definition for *RW-Through*, $a \xrightarrow{\mathsf{ar}} b$ implies that $a \xrightarrow{\mathsf{vis}} b$. With these definitions we can conclude that the guarantee READMYWRITES is provided. This guarantee together with SINGLEORDER is enough to conclude that *RW-Through* provides SEQUENTIALCONSISTENCY.

5. RW-Through Compared to LPCR and Cache-aside

In this section we discuss how *RW-Through* compares to LPCR and cache-aside. Initially we analyze the ordering guarantees and consistency models provided by the three solutions. After, we discuss network usage and the applicability of the solutions in large-scale geo-replicated scenarios.

	LPCR	Cache-aside	RW-Through
EVENTUALVISIBILITY	✓	✓	✓
READMYWRITES		✓	✓
MONOTONICREADS		✓	✓
CAUSALVISIBILITY			✓
CAUSALARBITRATION	✓	✓	✓
SINGLEORDER			✓
REALTIME			

Table 3. Ordering guarantees provided by each solution

Table 3 summarizes the ordering guarantees provided by the three solutions. Based on Table 1, we can determine the consistency model for each solution. According to distributed consistency specifications, *RW-Through* offers SEQUENTIALCONSISTENCY which is stronger than the BASICEVENTUALCONSISTENCY offered by both LPCR and *Cache-aside*.

In practice, not offering session guarantees (i.e. READMYWRITES and MONOTONICREADS) can result in unintuitive situations to users. For example, a user writes a comment in a social network, refreshes the page, and can not see his comment. This is a possible situation on a system that does not provide READMYWRITES. Systems that do not provide causal guarantees may result in scenarios where answers are ordered before questions. These anomalies are possible in both LPCR and cache-aside. On the other hand, *RW-Through* provides all session and causal guarantees.

Due to its replication characteristics, *RW-Through* full replication mode can be an alternative to LPCR. In single datacenter scenarios where network partitions and delays are rare, LPCR consistency anomalies are also rare. *RW-Through* on the other hand, offers SEQUENTIALCONSISTENCY regardless of the network conditions, making it a viable option for geo-replicated deployments. LPCR can also be used as a fault tolerance solution for node crashes (e.g. a secondary can assume the primary role when the primary fails). *RW-Through* does not have this property. Therefore, we conclude that *RW-Through* is a solution for read-intensive workloads and unreliable network problems while LPCR is a solution for read-intensive workloads and unreliable nodes problems.

RW-Through cache mode can offer advantages when compared to cache-aside. Besides offering a stronger consistency model, *RW-Through* cache mode is more efficient in network usage and is less susceptible to stale-reads than cache-aside. Network efficiency is higher due to cache-misses being more efficient in *RW-Through*. While cache aside requires 6 messages to be exchanged in a cache-miss, *RW-Through* completes the operation with 4 messages exchanged. The reduction in stale reads occurs due to the write log mechanism employed by *RW-Through*. In cache aside an item is only refreshed in the cache when it expires. In *RW-Through*, every modified item in the main storage is refreshed on the *Intermediate* whenever a communication is made between *Intermediate* and *Main Storage* process. This reduces the window that an item already modified on the main storage remains active in the cache.

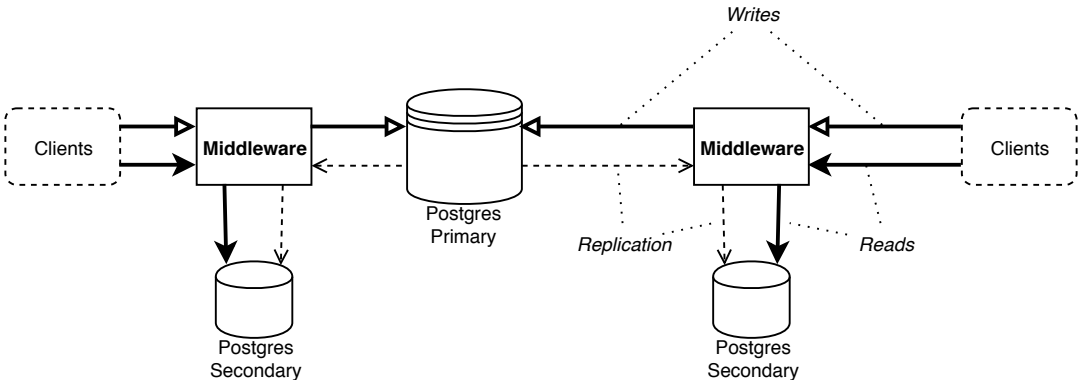

Figure 7. PostgreSQL middleware based solution based on *RW-Through*

6. Future Work

RW-Through can serve as a base for different types of projects involving data replication. In this paper we presented the algorithms and demonstrated its consistency and network usage properties. To validate its applicability in real world scenarios, we are currently developing projects that use *RW-Through*'s algorithms in different settings.

One such project is a middleware-based distributed PostgreSQL solution that implements *RW-Through*'s full replication mode. Figure 7 illustrates the general architecture implemented by the project. The main idea is to place a middleware between clients, a PostgreSQL instance configured as a primary copy, and a PostgreSQL instance configured as a secondary copy. One middleware exists for each secondary copy. All PostgreSQL instances are configured to use PostgreSQL's streaming replication. In this replication mode the primary sends write logs to secondaries in a way very similar to *RW-Through*. When a read operation is received, the middleware simply routes the query to the secondary. When writes are received, the middleware routes the query to the primary and waits for the write log containing the write operation. As soon as the log is applied to the secondary the operation is returned to the client. This scheme certainly adds an additional overhead to write operations when compared to other solutions. However, this solution is intended to be used when read throughput needs to be maximized and has the benefit of providing a higher consistency than LPCR.

7. Conclusion

In this work we conducted a formal consistency analysis on the main data storage solutions used by the industry for read-intensive workloads. We demonstrated that these solutions offer weak consistency models susceptible to many different consistency anomalies. The occurrence of such anomalies increases as the systems are exposed to faulty network conditions such as wide-area networks.

We presented *RW-Through*, a design for a data replication protocol capable of solving many of the problems faced by current solutions. *RW-Through* is capable of addressing the same issues as current solutions while offering a stronger consistency model. By using formal consistency verification frameworks, we demonstrated that our new protocol can maintain its consistency regardless of the network conditions. This makes *RW-Through* a suitable solution for large-scale geo-distributed deployments such as global

cloud applications.

In future works, we intend to use *RW-Through* as a basis for a PostgreSQL plugin. We aim to demonstrate that existing replication plugins can present consistency anomalies and compare the performance with our plugin on geo-distributed cloud deployments.

References

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

Bailis, P. and Ghodsi, A. (2013). Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

Brewer, E. A. (2000). Towards robust distributed systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00.

Bruner, J. (2013). Tweets loud and quiet.

Burckhardt, S. (2014). *Principles of Eventual Consistency*, volume 1. now publishers.

Cecchet, E., Candea, G., and Ailamaki, A. (2008). Middleware-based database replication: The gaps between theory and practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08.

Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. D. (2014). Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14.

Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59.

Gray, J., Helland, P., O'Neil, P., and Shasha, D. (1996). The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA. ACM.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

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

Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. (2013). Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*.

Pacitti, E. p., Minet, P., and Simon, E. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. Technical report.

Schwarz, R. and Mattern, F. (1994). Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174.

Viotti, P. and Vukolić, M. (2016). Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34.