# Fine-Grained Consistency Upgrades for Online Services

Filipe Freitas[‡†⋆], João Leitão[†], Nuno Preguiça[†], and Rodrigo Rodrigues[⋆*]
[‡]ISEL, Instituto Superior de Engenharia de Lisboa, Portugal; [†]NOVA-LINCS & FCT,
Universidade NOVA de Lisboa, Portugal; [⋆]INESC-ID; [*]IST, Universidade de Lisboa, Portugal

## Abstract

*Online services such as Facebook or Twitter have public APIs to enable an easy integration of these services with third party applications. However, the developers who design these applications have no information about the consistency provided by these services, which exacerbates the complexity of reasoning about the semantics of the applications they are developing. In this paper, we show that is possible to deploy a transparent middleware between the application and the service, which enables a fine-grained control over the session guarantees that comprise the consistency semantics provided by these APIs, without having to gain access to the implementation of the underlying services. We evaluated our middleware using the Facebook public API and the Redis datastore, and our results show that we are able to provide fine-grained control of the consistency semantics incurring in a small local storage and modest latency overhead.*

## 1 Introduction

Many computer systems and applications make use of stateful services that run in the cloud, with various types of interfaces mediating the access to these cloud services. For instance, an application may decide to store its persistent state in a Cassandra cluster running on Azure instances, or directly leverage a cloud storage service such as S3. At a higher level of abstraction, services such as Twitter or Facebook have not only attracted millions of users to their main websites, but have also enabled a myriad of popular applications that are layered on top of those services by leveraging the public APIs they provide.

An important challenge that arises from this layering is that the consistency semantics of these cloud services are almost always not clearly specified, with studies showing that in practice these services expose a number of consistency anomalies to applications [7]. Furthermore, even in the cases where precise specifications exist, it is difficult for programmers to reason about their impact, and this may lead to violations of application invariants that were meant to be preserved [2].

In this paper, we argue that it is possible to build a middleware layer mediating the access to cloud services in order to obtain fine-grained control over the consistency semantics that these services provide. The idea is that we can design a library that intercepts every call to the service or storage system running in the cloud, inserting relevant meta-data, calling the original API, and transforming the results that are obtained in a transparent way for the application. Through a combination of analyzing this meta-data and caching results that have been previously observed, this shim layer can then enforce fine-grained consistency guarantees.

In prior work, Bailis et al. [3] have proposed a similar approach, but with two main limitations compared to this work. First, their shim layer only provides a coarse-grained upgrade from eventual to causal consistency. In contrast, we allow programmers to turn on and off individual session guarantees, where different guarantees have been shown to be useful to different application scenarios [9]. Second, their work assumes the underlying $\langle$key,value$\rangle$ store is a NoSQL system with a read/write interface. Such an assumption simplifies the development of the shim layer, since (1) it gives the layer full access to the data stored in the system, and (2) it provides an interface with simple semantics.

In this work, we propose a shim layer that allows for a fine-grained control over the session guarantees that applications should perceive when accessing online services. These services typically enforce rate limits for operations issued by client applications. For guaranteeing that this limit is the same when using our shim layer, a single service operation should be executed for each application operation. Furthermore, our layer is not limited to using online storage services with a read/write interface, since it is designed to operate with services that offer a messaging interface such as online social networks. The combination of these three requirements raises interesting challenges from the perspective of the algorithms that our shim layer implements, e.g., to handle the fact that online social networks only return a subset of recent messages, which raises the question of whether a message does not appear because of a lack of a session guarantee or because of being truncated out of the list of recent messages.

We implemented our shim layer and integrated it with the Facebook API and the Redis storage system. Our evaluation shows that our layer allows for fine-grained consistency upgrades at a modest latency overhead.

The remainder of this paper is organized as follows. Section 2 discusses our target systems and the assumptions made regarding the centralized system over which third-party applications are developed. Section 3 discusses the architecture and high level view of our system, while Section 4 details the algorithms employed in our solution to enforce each of the session guarantees. Section 5 discusses our prototype implementation and presents experimental results obtained over two different services. Finally, Section 6 discusses relevant related work and Section 7 concludes the paper with some final remarks.

## 2 Target systems

Our goal is to provide particular consistency guarantees to third-party applications using popular online web services that expose public APIs. In particular, the application developer may choose to have individual session guarantees (read your write, monotonic reads, monotonic writes, and writes follows reads) as well as combinations of these properties (in particular, all four session guarantees corresponds to causality [5]). To achieve this, we provide a library that can be easily attached to the third-party client application, allowing us to enrich the semantics exposed through the system public API. There are multiple popular systems that provide such public APIs, with various differences in terms of the interface they expose. As such, we needed to focus on a group of APIs with a similar service interface that we can easily adapt to, and we chose to focus on a particular class of services, namely social networks, such as Facebook, Twitter, or Instagram. Our choice is based on the relevance and popularity of these services and also on the large number of third-party applications that are developed for them. In particular, we target services that expose a data model based on key-value stores, where data objects can be accessed through a key, and that associate a list of objects to each key. We observe that this data model is prevalent in online social network services, particularly since they share concepts such as user feeds and comment lists. In particular, we target services where the API provides two fundamental operations to manipulate the list of objects associated with a given key: an insert operation to append a new object to the first position of the list, and a get operation that exposes the first $N$ elements of the list (i.e., the most recent $N$ elements).

Since we access these services through their public APIs, we need to view the service implementation as a black box, meaning that no assumptions are made regarding their internal operation. Furthermore, we design our protocols without making any assumption regarding the consistency guarantees provided through the public service API. The importance of not assuming any guarantees from existing services is justified by our own previous measurement study [7], which showed a high prevalence of violations of multiple session guarantees in public APIs provided by services of this class.

Our algorithms require storing meta-data alongside the data, which can be difficult to do when accessing services as black boxes, namely when the service has no support for including user managed meta-data (this is the case of Facebook, which we explore in the context of our prototype experimental evaluation). In this case, we need to encode this meta-data as part of the data itself. As a consequence, when the service is accessed by native clients (i.e., web applications or third party applications that do not resort to our Middleware) the user might see this meta-data. However, we believe that this is not a crucial issue, since many third party applications only access lists that are used exclusively by that application.

In order to arbitrate an ordering among operations issued by the local client and other remote clients, our Middleware has the need to have an approximate estimate of the current time. To achieve this, two options are available. If the service has a specific call in its public API that exposes the time in the server, such call can directly be used by our system. Otherwise, if the service exposes a REST API (which is typical in many services) a simple REST call can be performed to the service, and the server time can be extracted from a standard HTTP response header (called *Date*). Note that, even though it is desirable that this estimate is synchronized across clients, we do not require either clock or clock rate synchronization for correctness. In particular, the only negative effect of clocks being out of synch is a reordering of concurrent events from different sessions that is incoherent with their real time occurrence; this can imply, in the case of a service that outputs a sliding window of recent events, that more recent messages may be considered eligible for being truncated (i.e., considered older than the lower end of the window). However, we guarantee that such ordering never violates the correctness conditions we are enforcing.

Finally, we observe that, in practice, the public API exposed by these services often imposes rate limits for operations issued by client applications. These rate limits are exposed under the form of a maximum number of operations that can be executed within a given time window. In particular, we have experimentally observed that violating these rate limits can lead

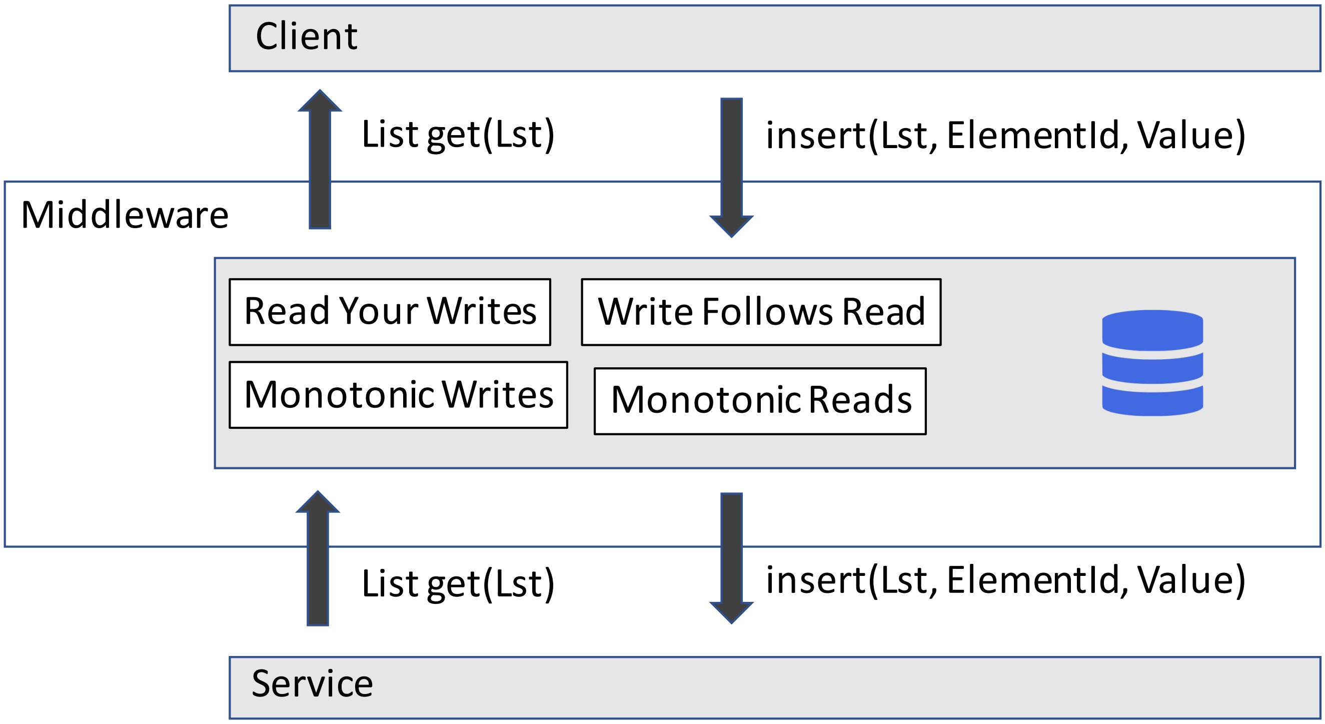


**Figure 1. Middleware**

the service to either block further access by the application, or introduce noticeable delays in processing requests issued by the application. The existence of operation rate limits imposes a requirement on our protocols: for each application operation, a single service operation can be issued. This is important to guarantee that an application using our middleware faces the same rate limits as an application using directly the service.

## 3 System overview

In this section we discuss the general architecture of our solution, which is materialized in a library implementing a middleware layer. We then provide an overview of the operation of our protocols, explaining how they enforce the consistency guarantees of session properties in a transparent way for the client applications.

### 3.1 Architecture

Our system consists of a thin layer that runs on the client side and intercepts every call made by the third-party client application on the service, mediating access to the service. In particular, our layer is responsible for contacting the service on behalf of the client application, process the responses returned by the service and generate responses to the client applications according with the session guarantees being enforced. Figure 1 provides a simple representation of this architecture.

Our system can be configured by the third-party application developer to enforce any combination of the individual session guarantees (as defined by Terry et. al [9]), namely: $i$) read your writes, $ii$) monotonic reads, $iii$) monotonic writes, and $iv$) writes follows reads. In order to enforce these guarantees, our system is required to maintain information regarding previous operations executed by the client application, namely previous writes that were issued or previous values that were observed by the client. In addition, our layer can also insert meta-data that is stored alongside the data in the original system, but stripped by the library before the final response is conveyed to the client.

**Algorithm 1:** Initialization of local state

```
1: upon init($Lst$) do
2:     lstState ← init()
3:     lstState.insertSet ← {}
4:     lstState.localView ← {}
5:     lstState.lastTimestamp ← 0
6:     lstState.insertCounter ← 0
7:     listStates[$Lst$] ← lstState
```

### 3.2 Overview

As mentioned, our system intercepts each request performed by the client application, executes the request in the service, and then processes the answer generated by the service to provide a (potentially different) answer to the client application. This answer is computed based on a combination of the internal state that records the previous operations that were run by that particular client, and the actual response that was returned by the service.

**Tracking application activity.** In order to keep track of user activity, our system maintains in memory a set of data structures for each part of the service state that is accessed by the application. These data structures are updated according both to the activity of the applications (i.e., the operations that were invoked) and the state that is returned by the service. These data structures are: $i)$ the *insertSet*, which stores the elements inserted by the client and $ii)$ the *localView*, which stores the elements returned to the client.

**Enforcing session guarantees.** Enforcing session guarantees entails achieving two complementary aspects. First, and depending on the session guarantees being enforced, some additional meta-data must be added when inserting operations. As mentioned, this meta-data can be either added to a specialized meta-data field (if the API exposed by the service allows this) or directly encoded within the body of the element being added to the list. Such meta-data has to be extracted by our library when retrieving the elements of a list, thus ensuring transparency towards client applications. Second, our system might be required to either remove or add elements to the list that is returned by the service when the application issues an operation to obtain the current service state, in order to ensure that the intended session guarantees are not violated.

In the next section, we discuss the concrete algorithms executed by our system upon receiving an insert or get operation for a particular list, in order to ensure that the values observed by the client application adhere to the semantics of each of the session guarantees that are intended to be provided.

## 4 Algorithms

We now discuss in more detail the algorithms that are employed by our Middleware layer to enforce session guarantees, and the rationale for their design. To this end, we briefly remind what each of the four session guarantees entails (we extend the definitions previously introduced in [7]), and then explain why our algorithms ensure that the anomalies associated with each of the session guarantees are prevented by it.

We explain our algorithms assuming that the service offers an interface with the following two functions, which are in practice easily mapped to functions that are supported by the various services that we analyzed: the insertion of an element in a given list $Lst$, denoted by the execution of function `insert`($Lst$,$ElementID$,$Value$), where $Lst$ identifies the list being accessed, $ElementID$ denotes the identifier of the element being added (which can be an identifier generated by the centralized service or a unique identifier generated by our Middleware), and $Value$ stands for the value of the element being added to the list; and the access to the contents of a list, denoted by the execution of function `get`($Lst$), where $Lst$ identifies the list being read by the client.

When the client accesses a list $Lst$ for the first time, a special initialization procedure is triggered internally by our Middleware (Alg. 1), which initializes the local state regarding the accesses to $Lst$. The initialization is straightforward: it creates the object *lstState* that maintains all relevant information to manage the accesses to $Lst$ (line 2). This state is composed by the sets **insertSet** and **localView** that were discussed previously, and that are initially empty (lines $3-4$). Furthermore, two other variables are initialized, **lastTimestamp**, which is used to maintain information regarding elements that were removed from the previously discussed sets, and **insertCounter**, which tracks the number of inserts performed by the local client in the context of the current session. Both of these variables have an initial value of zero (lines $5-6$). Finally, the *lstState* variable is stored in a local map, associated to the list $Lst$ (line 7). Next, we explain how this local state is leveraged by our algorithms to enforce the various session guarantees.

**Algorithm 2:** Read Your Writes

```
1:  function insert($Lst$, $ElementId$, $Value$) do
2:      lstState ← listStates[$Lst$]
3:      Element e ← init()
4:      e.v ← $Value$
5:      e.id ← $ElementId$
6:      e.timestamp ← obtainServiceTimeStamp()
7:      SERVICE.insert($Lst$, $ElementId$, e)
8:      lstState.insertSet ← e ∪ lstState.insertSet

9:  function get($Lst$) do
10:     lstState ← listStates[$Lst$]
11:     sl ← SERVICE.get($Lst$)
12:     sl ← orderByTimestamp(sl)
13:     sl ← addMissingElementsToSL(sl, lstState.insertSet, lstState.lastTimestamp)
14:     sl ← purgeOldElementFromSL(sl, lstState.insertSet, lstState.lastTimestamp)
15:     lstState.lastTimestamp ← getLastTimestamp(sl)
16:     return removeMetadata(subList(sl, 0, $N$))
```

## 4.1 Read Your Writes

The Read Your Writes (RYW) session guarantee requires that, in a session, any read observes all writes previously executed by the same client. More precisely, for every set of insert operations $W$ made by a client $c$ over a list $L$ in a given session, and set $S$ of elements from list $L$ returned by a subsequent get operation of $c$ over $L$, we say that RYW is violated if and only if $\exists\, x \in W : x \notin S$.

This definition, however, does not consider the case where only the $N$ most recent elements of a list are returned by a get operation. In this case, some writes of a given client may not be present in the result if more than $N$ other insert operations have been performed (by client $c$ or any other client). Considering that the list must hold the most recent writes, a RYW anomaly happens when a get operation returns an older write performed by the client but misses a more recent one. More formally, given two writes $x$, $y$ over list $L$ executed in the same client session, where $x$ was executed before $y$, an anomaly of RYW happens in a get that returns $S$ when $\exists\, x, y \in W : x \prec y \wedge y \notin S \wedge x \in S$.

Alg. 2 presents our algorithm for providing RYW. To avoid the anomaly described above, the idea is to store, locally at the client, all elements that are inserted by the local client in the list and add them to the result of get operations. In the insert operation, the inserted element is stored locally by the client (line 8). Additionally, our algorithm stores some meta-data in the object before performing the insert operation over the centralized service (lines $5-6$). This information represents, respectively, the identifier of the element and a timestamp for the insert operation (from the perspective of the service, and retrieved as described in Section 2). The element identifier is used to uniquely identify the writes. The timestamp and element identifier allow for totally ordering all entries in the **insertSet**, with the order being approximately that of the real-time order of execution. Note that the operation in line 12 also checks if the timestamps retrieved from the service in the same session are monotonically increasing, and, if not, enforces that property by overwriting the returned timestamp with an increment of the most recent one; this is important to avoid reordering events from the same session in case the timestamp provided by the server does not increase monotonically for some reason.

For executing a get operation (line 9) our algorithm starts by executing the get operation over the service (line 11). Then, the returned list (*sl*) is ordered (line 12) and all elements of the local **insertSet** that are missing in the list are added to the list, keeping it ordered (line 13). Before returning the most recent $N$ elements (with no meta-data) (line 16), our algorithm removes old session elements from the *sl* list and updates the **lastTimestamp** variable with the timestamp of the oldest element of the client session returned to the client (lines $14-16$).

A limitation of this algorithm is that it causes the **insertset** to grow indefinitely. To avoid this, we use the timestamp of each element to remove from the **insertset** any element older than **lastTimestamp**. We also need to include the session id in the metadata of each element to avoid old elements of the session to reaper. We omit this from Alg. 2 for readability.

## 4.2 Monotonic Reads

This session guarantee requires that all writes reflected in a read are also reflected in all subsequent reads performed by the same client. To define this in our scenario where a truncated list of $N$ recent elements is returned, we say that Monotonic Reads (MR) is violated when a client $c$ issues two read operations that return sequences $S_1$ and $S_2$ (in that order) and the

**Algorithm 3:** Monotonic reads

```
1:  function insert($Lst$, $ElementID$, $Value$) do
2:      Element e ← init()
3:      e.id ← $ElementID$
4:      e.v ← $Value$
5:      SERVICE.insert($Lst$, $ElementID$, e)

6:  function get($Lst$) do
7:      lstState ← listStates[$Lst$]
8:      sl ← SERVICE.get($Lst$)
9:      lstState.localView ← appendNewElementsToTop(sl, lstState.localView)
10:     return removeMetadata(subList(lstState.localView, 0, $N$))
```

**Algorithm 4:** Monotonic Writes

```
1:  function insert($Lst$, $ElementID$, $Value$) do
2:      lstState ← listStates[$Lst$]
3:      Element e ← init()
4:      e.id ← $ElementID$
5:      e.v ← $Value$
6:      e.clientSession ← getClientSessionID()
7:      e.sessionCounter ← lstState.insertCounter++
8:      SERVICE.insert($Lst$, $ElementID$, e)

9:  function get($Lst$) do
10:     lstState ← listStates[$Lst$]
11:     sl ← SERVICE.get($Lst$)
12:     sl ← sortElementsBySessionCounters(sl)
13:     sl ← removeElementsWithMissingDependencies(sl)
14:     return removeMetadata(sl)
```

following property holds: $\exists\, x, y \in S_1 : x \prec y$ *in* $S_1 \wedge y \notin S_2 \wedge x \in S_2$, where $x \prec y$ means that element $x$ appears in $S_1$ before $y$.

To avoid this anomaly, our algorithm (presented in Alg. 3) resorts to the **localView** variable to maintain information regarding the elements (and their respective order) observed by the client in previous get operations. Therefore, when the client issues a get operation, our Middleware issues the get command over the centralized service (line 8) and then updates the contents of its **localView** with any elements that are returned by the service and that were not yet within the **localView** (line 9). These new elements are appended to the start of the list, as they are assumed to be more recent than those of the current **localView**.

The algorithm terminates by returning to the client the $N$ most recent elements in the **localView**. These elements are exposed to the client without any of the meta-data added by our algorithms (line 10). Note that in this case the insert operation only issues the corresponding insert command with additional meta-data on the centralized service (lines $1-5$).

Similar to the previously discussed algorithm, this approach has a limitation regarding the growth of local state, in this case the **localView** can grow indefinitely. To avoid this, we associate with each element inserted in the list a timestamp. This timestamp allows us to remove from the **localView** any element with a timestamp smaller than the timestamp of the oldest element that was in the last return to the client. We omit this from Alg. 3 for readability.

### 4.3 Monotonic Writes

This session guarantee requires that writes issued by a given client are observed in the order in which they were issued by all clients. More precisely, if $W$ is a sequence of write operations issued by client $c$ up to a given instant, and $S$ is a sequence of write operations returned in a read operation by any client, a Monotonic Writes (MW) anomaly happens when the following property holds, where $W(x) \prec W(y)$ denotes $x$ precedes $y$ in sequence $W$: $\exists x, y \in W : W(x) \prec W(y) \wedge y \in S \wedge (x \notin S \vee S(y) \prec S(x))$.

However, this definition needs to be adapted for the case where only $N$ elements of a list are returned by a get operation. In this case, some session sequences may be incomplete, because older elements of the sequence may be left out of the truncated list of $N$ returned elements. Thus, we consider that older elements are eligible to be dropped from the output, provided that we ensure that there are no gaps in the session subsequences and that the write order is respected, before returning to the

**Algorithm 5:** Write Follows Read

```
1:  function insert($Lst$, $ElementID$, $Value$) do
2:      lstState ← listStates[$Lst$]
3:      Element e ← init()
4:      e.id ← $ElementID$
5:      e.v ← $Value$
6:      e.cutTimestamp ← obtainCutTimestamp(lstState.localView)
7:      e.dependencies ← projectElementIdentifiers(lstState.localView)
8:      e.timestamp ← obtainIncreasingServiceTimeStamp(lstState.localView)
9:      SERVICE.insert($Lst$, $ElementID$, e)

10: function get($Lst$) do
11:     lstState ← listStates[$Lst$]
12:     sl ← SERVICE.get($Lst$)
13:     sl ← removeElementsWithMissingDependencies(sl)
14:     cutTimestamp ← highestCutTimestamp(sl)
15:     sl ← removeElementsBelowCutTimestamp(sl, cutTimestamp)
16:     lstState.localView ← appendNewElementsByTimestamp(ls, lstState.localView)
17:     lstState.localView ← purgeOldElements(lstState.localView)
18:     return removeMetadata(sl)
```

client. Formally, we can redefine MW anomalies as follows, given a sequence of writes $W$ in the same session, and a sequence $S$ returned by a read: $(\exists x, y, z \in W : W(x) \prec W(y) \prec W(z) \wedge x \in S \wedge y \notin S \wedge z \in S) \vee (\exists x, y \in W : W(x) \prec W(y) \wedge S(y) \prec S(x))$.

Alg. 4 presents the algorithm employed by our Middleware to enforce the MW session guarantee. We avoid the anomaly described above by adding meta-data to each insert operation (lines $1-8$) in the form of a unique client session id (*clientSession* – line 6) and a counter (local to each client and session) that grows monotonically (*sessionCounter* – line 7). This information allows us to establish a total order of inserts for each client session.

This meta-data is then leveraged during the execution of a get operation (lines $9-14$) in the following way. After reading the current list from the service (line 11), we simply order the elements in the read list (*sl*) to ensure that all elements respect the partial orders for each client session (line 12). Finally, an additional step is required to ensure that no element is missing in any of these partial orders. To ensure this, whenever a gap is found within the elements of a given client session, we remove all elements whose *sessionCounter* is above the one of any of the missing elements.

The get operation returns the contents that are left in the list *sl* without the meta-data added by our algorithms (line 14). Note that in this case we might return to the client a list of elements with a size below $N$. We could mitigate this behavior by resorting to the contents of the **localView** as we did in the algorithm to enforce MR. However, we decided to provide the minimal behavior to enforce each of the session guarantees in isolation.

## 4.4 Write Follows Read

This session guarantee requires that the effects of a write observed in a read by a given client always precede the writes that the same client subsequently performs. (Note that although this anomaly has been used to exemplify causality violations [1, 8], any of the previous anomalies represent a different form of a causality violation [9].) To formalize this definition, and considering that the service only returns at most $N$ elements in a list, if $S_1$ is a sequence returned by a read invoked by client $c$, $w$ a write performed by $c$ after observing $S_1$, and $S_2$ is a sequence returned by a read issued by any client in the system; a violation of the Write Follows Read (WFR) anomaly happens when: $w \in S_2 \wedge \exists x, y \in S_1 : x \prec y \textit{ in } S_1 \wedge y \notin S_2 \wedge x \in S_2$.

Our algorithm to enforce this session guarantee is depicted in Alg. 5. The key idea to avoid this anomaly is to associate with each insert the direct list of dependencies of that insert, i.e, all elements previously observed by the client performing the insert (line 7). Evidently, this solution is not practical, since this list could easily grow to include all previous inserts performed during the lifetime of the system. To overcome this limitation, we associate with each insert a timestamp based on the clock of the service, but with the restriction of being strictly greater than the timestamp of any of its direct dependencies (line 8). Furthermore, we also associate with each insert a cut timestamp, that defines the timestamp of its last explicit dependency, i.e, the dependencies registered in the dependency list (line 6). The cut timestamp implicitly defines every element with a lower timestamp to be a dependency of that insert operation. By combining these different techniques, we ensure that the explicit dependency list associated with an insert has at most a value around $N$ elements (which is the size of the **localView** maintained by our Middleware).

Since only $N$ elements of a list are returned by a get operation, the older dependencies may be left out of the sequence that is returned. When this happens, it is safe to consider that these dependencies were dropped from the window that is returned, provided that we ensure that, for each element that is returned, all dependencies that are more recent than the oldest element are also returned.

In the get operation we leverage this meta-data to do the following: we start by reading the contents of the list from the service (line 12) and then over this list we remove any insert whose dependencies are missing. Thus, we only remove inserts whose missing dependencies have a timestamp above the insert cut timestamp. We then compute a cut timestamp for the obtained list *sl* (line 13) that is the highest cut timestamp among all elements in *sl*. We use this timestamp to remove from *sl* any element whose creation timestamp falls below the computed cut timestamp. Finally, before returning to the client the elements that remain in *sl* without the additional meta-data (line 18) we update and garbage collect old entries from the **localView** (lines $16-17$).

Similarly to the previous algorithm, the service might return a number of elements that is lower than $N$. In this case, to ensure that we always return $N$ elements, we need to obtain the missing dependencies using a get operation that returns a single element (if supported by the service). In our implementation, we avoided this solution because it is prone to triggering a violation of the API rate limits. Again, an alternative way to address this is by, after reading the list from the service, merging its contents with those in the **localStore** and enforcing an order that is compatible with the timestamp of each element. However, for simplicity in exposition, we omit the details of this alternative.

### 4.5 Combining multiple session guarantees

Considering the algorithms to enforce each of the session guarantees discussed above, we can now summarize how to combine them. In a nutshell, it suffices for our Middleware to, on insert operations, add the meta-data used by each of the individual algorithms according to the guarantees configured by the application developer. Correspondingly, upon the execution of a get operation, our Middleware must perform the transformations over the list obtained from the service (*sl*) prescribed by each of the individual algorithms. Furthermore, all meta-data added to each element must also be removed before exposing data to the client application.

## 5 Evaluation

In this section we present the experimental evaluation of our Middleware, which compares the client-perceived performance obtained when using our Middleware to provide each of the session guarantees in isolation and their combination (i.e, enforcing all four session guarantees). In our experiments we used a prototype of our Middleware whose implementation we briefly discuss below. Our evaluation was made using two different geo-replicated online services. First, to illustrate the benefit of our Middleware when designing third-party applications that interact with online social networks we have used Facebook's public API. Then, to illustrate the operation of our Middleware when interacting with a service that imposes fewer restrictions on the number and timing of client operations, we experimented with a geo-replicated deployment of the Redis datastore managed by ourselves.

Our evaluation focuses on asserting the overhead that results from the use of our middleware, in terms of client perceived latency (for insert and get operations), the communication overhead due to the inclusion of additional meta-data, and the storage overhead, namely due to the need for our Middleware to locally maintain some information about previous operations performed by the client.

### 5.1 Implementation

Our prototype of the Middleware layer proposed in the paper was implemented in the Java language. To interact with the two services that we explore in this work, we resorted to the *restFB* library for Facebook[1], and the Jedis library for interacting with Redis[2].

Since our prototype was designed to interact with any Internet service with a public API, it requires two adapter layers to be written and provided to its runtime upon execution (see Figure 2). These layers capture the API calls performed by the client application and translate them to a standard API exposed by our Middleware, and translate the calls to the

[1] `http://restfb.com`

[2] `https://github.com/xetorthio/jedis`

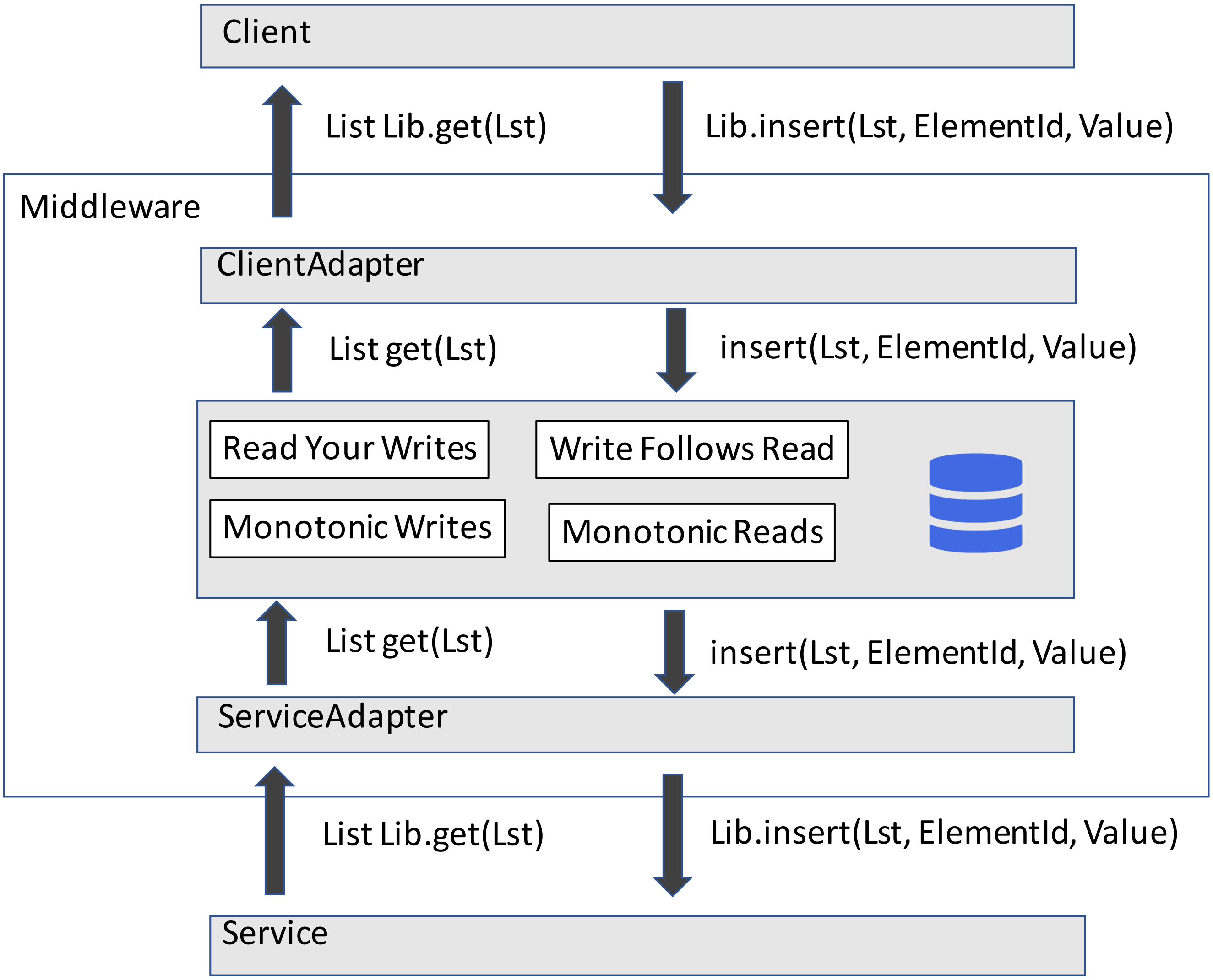


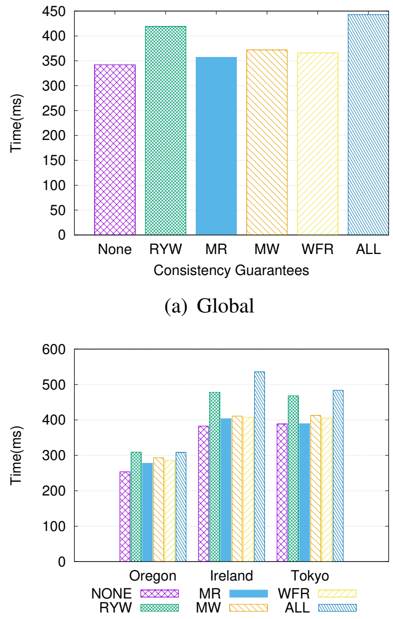
**Figure 2. Middleware with adapters**

centralized service performed by our Middleware into API calls to the library used to interact with the service, respectively. We have implemented these adapters for the two case studies employed in this evaluation. The adapters themselves are quite straightforward to write, and we believe most developers will be able to easily write new adapters to use our Middleware in combination with different libraries for accessing other online services.

### 5.2 Facebook Results

We have conducted our experiments with Facebook by using YCSB [6] to emulate clients using Facebook to post messages to a group feed and reading the contents of that group feed. To emulate such clients spread across the World, we run three independent YCSB instances in three different locations using Amazon EC2 instances in Oregon, Ireland, and Tokyo. Each YCSB instance uses 10 threads, emulating a total of 30 independent clients, for a total of 90 clients across the World. Each emulated client has an independent instance of our Middleware. To accommodate the rate limits of Facebook's public API, we impose a maximum of 15 requests per second per YCSB instance.

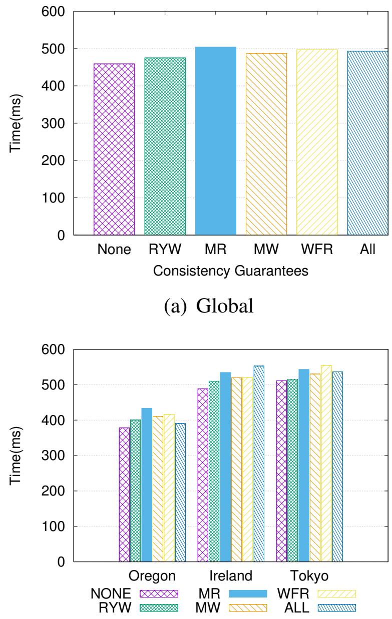
Each experiment reported in this section was executed 7 times, and different consistency guarantees were rotated along experiments, such that each different consistency guarantee had experiments running on different time periods of the day. This was done to remove experimental noise due to contention on the Facebook servers, e.g., due to other user activity. The



(a) Global

(b) Per location

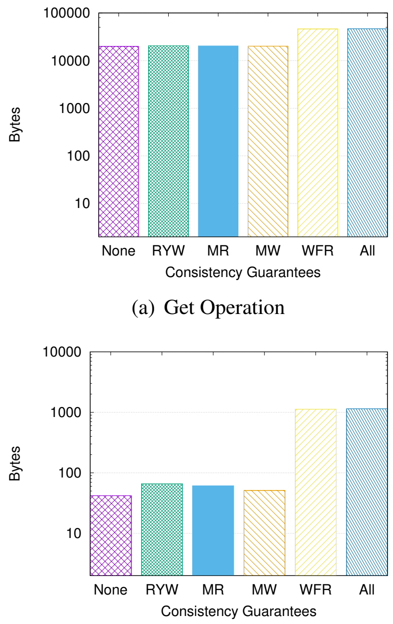
**Figure 3. Latency of Get Operation in Facebook**



(a) Global

(b) Per location

**Figure 4. Latency of Insert Operation in Facebook**



(a) Get Operation

(b) Insert Operation

**Figure 5. Communication overhead in Facebook**

workload executed by clients was a mix of 50% inserts and 50% gets. The Middleware was configured to have $N = 25$ meaning that each get retrieves at most 25 elements from the feed. Experiments reported in this section report the aggregated observations of $53,119$ insert and get operations.

### 5.2.1 Latency

We start by observing the latency of operations in Facebook when accessing the service directly through the library (labeled in the plots as NONE) and when using our Middleware to enforce each of the session guarantees in isolation and all of the session guarantees (labeled in the plots as ALL).

Figure 3 reports the latency observed for get operations, for all clients and per location of the client. Figure 3(a) shows that our Middleware introduces a small increase in the latency of get operations with a maximum increase of approximately one hundred milliseconds. Not surprisingly the overhead is at its maximum when all session guarantees are being enforced by our Middleware which is explained by a combination of the additional meta-data carried in each element, and the processing cost of the Middleware to perform the enforcement of each individual session guarantee.

When observing the distribution of latency for requests according to the region where the client is located (Figure 3(b)), we note the same relative distribution in the results, with overall lower latency values for the clients in Oregon. This is explained by the latency of those clients towards the Facebook servers, which is notoriously smaller according to the latency when using the client library directly. Another noteworthy aspect of Figure 3(b) is that the observed latency has a visible variation, both across and even within different client locations. This suggests that the latency overhead in these cases may suffer from a noticeable variability due to external factors which are related with the architecture and deployment of such a large-scale real World application.

Figure 4 reports average latency results for the insert operation for all clients and per client location. The results reported in Figure 4(a) show that globally the latency penalty incurred by the use of our Middleware is again modest, with a maximum increase of at most 50 milliseconds. The individual session guarantee with the largest increase in latency is monotonic reads. Considering the latency values observed in different locations reported in Figure 4(b), we note the same pattern previously observed, where the latency experienced by clients in Oregon is lower compared with the remaining locations. This is expected, since this can be explained by the latency experienced by the client to contact the Facebook service in that concrete

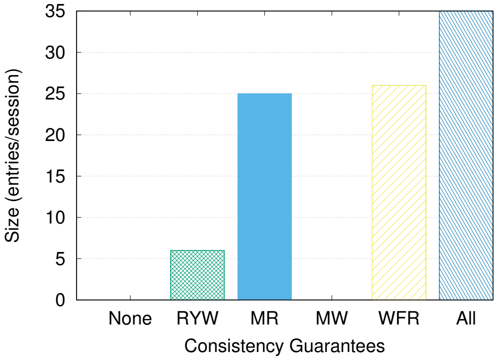


**Figure 6. Local storage overhead for Facebook**

location when compared with the remaining locations used in our experimental work.

#### 5.2.2 Communication Overhead

We now study the communication overhead imposed by our Middleware by observing the average size of messages exchanged between clients and the service. Figure 5 reports these results for each of the session guarantees and for their combination, compared with the use of the library without our Middleware, for both get and insert operations. The results show that the overhead introduced by our Middleware is low for the get operations. This happens because most of the payload in these messages are the multiple elements of the list that are returned. Since our algorithms use small meta-data objects, the communication cost remains dominated by the contents of the elements that are read, as can be observed in Figure 5(a).

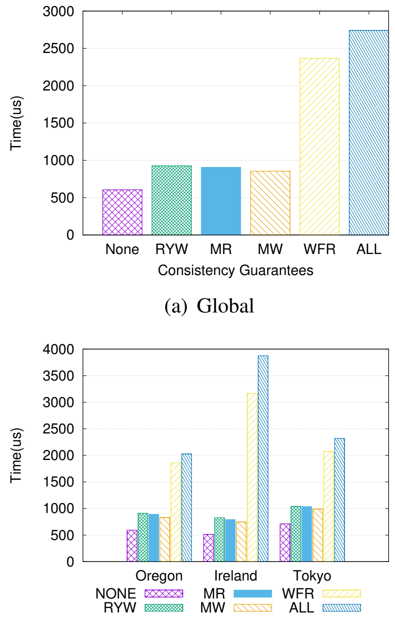
The same is not true for insert operations, as reported in Figure 5(b). In this case, since each message contains only a single element to be added, the increase in message size is quite noticeable when the Middleware is enforcing Writes Follows Reads and the combination of all session guarantees. This happens due to the cost of sending the explicit dependencies of each inserted element, which can account to 25 unique element identifiers and their timestamps. The remaining session guarantees, in contrast, have a modest overhead of only a few tens of bytes.

#### 5.2.3 Local Storage Size

Finally, Figure 6 reports the storage cost in terms of elements stored locally by our Middleware for enforcing each of the session guarantees and their combination. For completeness, we also provide the results for the NONE configuration, which, as expected, is zero. This is used as a sanity check for our results. Monotonic writes do not require any form of local storage, and therefore have no local storage overhead. In contrast, the remaining session guarantees resort to elements stored in the insertSet and localView data structures. As expected, when providing all of the session guarantees the local storage has more entries, this happens because the number of entries is the sum of the elements in the insertSet and in the localView.

### 5.3 Redis Results

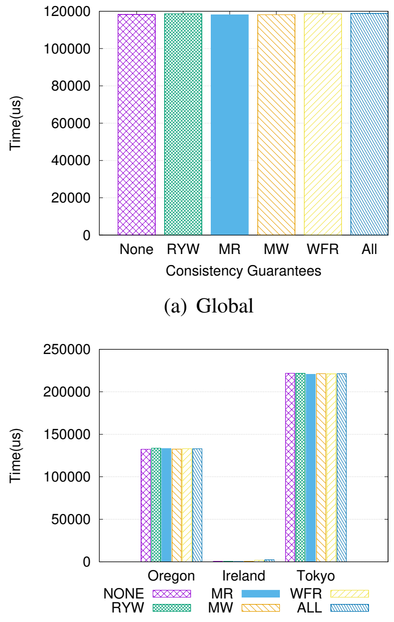
We also conducted experiments using the Redis data storage system. To this end, we have deployed Redis with its replication enabled across machines scattered in three Amazon EC2 regions: Oregon, Tokyo, and Ireland. Redis uses a master-slave replication model, and we have deployed the master in Ireland and two slaves in each region, for a total of 7 replicas. YCSB was executed in the same three regions of Amazon EC2 used in the previously reported experiments, with each YCSB instance running 10 threads that execute operation in a closed loop. Each thread has its own instance of the Middleware. All operations access the same list object stored in Redis, with the read operation being executed in one of the slave replicas, selected randomly. For each algorithm, we run our experiments 6 times for 60 seconds with an interval of four minutes between runs. Similar to the experiments conducted with Facebook, YCSB was configured to execute a workload composed of $50\%$ inserts and $50\%$ of reads. Again, we set $N$ to be equal to 25. The experiments reported in this section aggregate the results from executing a total of $21,285,291$ insert and get operations.



(a) Global

(b) Per location

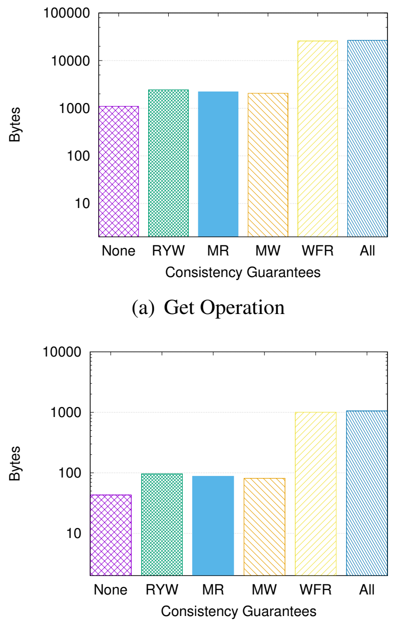
**Figure 7. Latency of Get Operation in Redis**



(a) Global

(b) Per Location

**Figure 8. Latency of Insert Operation in Redis**



(a) Get Operation

(b) Insert Operation

**Figure 9. Communication overhead in Redis**

### 5.3.1 Latency

Figure 7(a) presents the average latency of get operations. The results shows that our middleware introduces a very small overhead, on the order of microseconds, for Read Your Writes, Monotonic Reads, and Monotonic Writes. In Write Follows Read and when all session guarantees are enforced, there is an increase of approximately one to two milliseconds because the algorithms have to check the dependencies and process the meta-data. The results of Figure 7(b), which details the values observed in each region, show the same pattern across all regions, namely that the latency for reading data using a client in Ireland is higher than in other locations (due to the proximity to the master replica). This can be explained by the fact that writes in Ireland are much faster that in other locations, which causes the number of read operations that are executed to be higher in Ireland than in other locations, thus leading to a higher load, which results in a higher latency for executing operations.

In contrast to the experiments for the Facebook service, the observed latencies are much more predictable in this deployment. This confirms the expectation that a real-world service leads to qualitatively different results from a controlled experiment.

Figure 8(a) shows the latencies of the insert operation, in this case the latency is almost the same across all cases, but if we look to Figure 8(b) we see that in Ireland latency values are much smaller, this is again justified by the location of the master replica in Ireland and the fact that all clients are issuing their write operations to the (same) master replica. Figure 10 reports the latencies in Ireland, again a similar pattern to the one observed for Get operations.

### 5.3.2 Communication Overhead

In terms of communication overhead imposed by our Middleware, the results in Figure 9(a) and Figure 9(b) show that in the Get and Insert operations the overhead is more noticeable when enforcing Write Follows Read and in when employing the combination of all algorithms. This happens due to the overhead associated with managing and communicating the information stored in dependency lists.

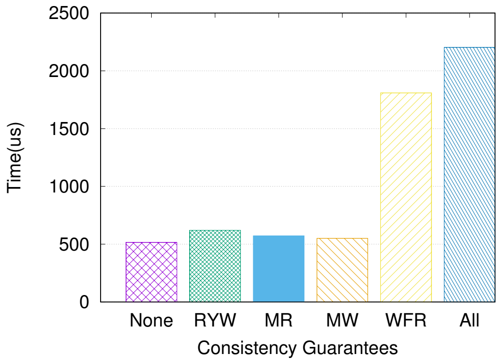


**Figure 10. Latency of Insert Operation in Redis in Ireland**

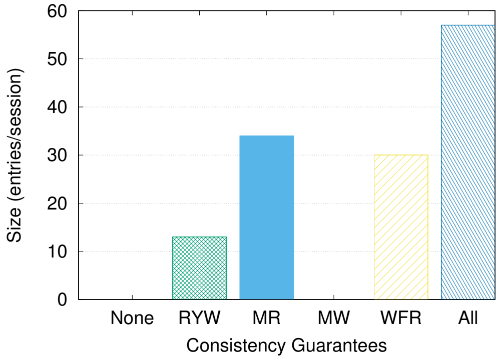


**Figure 11. Local storage overhead for Redis**

### 5.3.3 Local Storage Size

To conclude, Figure 11 shows that in Monotonic Reads and Write Follows Read the number of elements in the localView is around $30$, which is higher than $N = 25$. This happens because of the high write throughput, which causes several elements to be assigned the same timestamp. In this case, our truncation algorithm allows for the limit to be exceeded in the case of ties. The combinations of all algorithms is also affected by this situation, leading to a higher value around $55$. Note that we are showing the average of the highest value registered for each independent client session at any time during its execution.

## 6 Related Work

The closest related work can be found in the recent proposals that also leverage a middleware layer that can mediate access to a storage system in order to upgrade the respective consistency guarantees [3, 4].

In particular, Bailis et al. [3] proposed a system called “bolt-on causal consistency” to offer causal consistency on top of eventually consistent data stores. There are two main distinctions between bolt-on causal consistency and our proposal: first, we provide a fine-grained choice of which session guarantees the programmer intends the system to provide, and only pay a performance penalty that is associated with enforcing those guarantees. Second, they assume the underlying system offers a general read/write storage interface, which gives significant more flexibility in terms of the system design than in our proposal, which is restricted to the APIs provided by social networking services.

The other closely related system is the one proposed by Bermbach et al. [4], which is also based on a generic storage interface, namely that provided by S3, DynamoDB, or SimpleDB, in contrast to our focus on high level service APIs. While they also provide fine-grained session guarantees chosen by the programmer, they limit these to Monotonic Reads and Read Your Writes.

Our own prior work provides a measurement study of the violations of session guarantees that are observed when accessing real services [7]. However, the focus of that prior work is on understanding the prevalence of occurrences of lack of session guarantees, whereas this proposal is about fixing those problems through a middleware layer implementing a series of novel algorithms.

## 7 Conclusions

In this paper we have shown that it is possible to enforce different consistency properties, in particular session guarantees for third party applications that access online services through their public APIs. We do so without explicit support from the service architecture, and without assuming that the service itself provides any of these guarantees. Our solution relies on a thin Middleware layer that executes on the client side, and intercepts all interactions of the client with the online service. We have presented different algorithms to enforce each of the well known session guarantees. Furthermore, our algorithms follow a simple structure that allows to combine then easily.

We have developed a prototype in Java that we used to evaluate our approach using two centralized services: Facebook, and a geo-replicated deployment of Redis. Our experiments show that we can enforce session guarantees with a modest overhead both in terms of user-perceived latency and communication with the centralized service.

### Acknowledgments

This work was partially supported by the EU project LightKone (grant agreement n. 732505) and by FCT (UID/CEC/04516/2013). The research of R. Rodrigues is funded by the European Research Council (ERC-2012-StG-307732) and by FCT (UID/CEC/50021/2013). Part of the computing resources used for this work were supported by an AWS in Education Research Grant.

## References

[1] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, New York, NY, USA, 2015. ACM.

[3] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[4] David Bermbach, Jorn Kuhlenkamp, Bugra Derre, Markus Klems, and Stefan Tai. A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores. In *Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, IC2E '13, pages 114–123, Washington, DC, USA, 2013. IEEE Computer Society.

[5] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 152–158, Feb 2004.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[7] F. Freitas, J. Leitao, N. Preguia, and R. Rodrigues. Characterizing the Consistency of Online Services (Practical Experience Report). In *Proceedings of the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 638–645. IEEE, June 2016.

[8] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[9] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.